Table of Contents

pgRouting extends the **PostGIS**/**PostgreSQL** geospatial database to provide geospatial routing and other network analysis functionality.

This is the manual for pgRouting v3.0.0.

General

Introduction

pgRouting is an extension of **PostGIS** and **PostgreSQL** geospatial database and adds routing and other network analysis functionality. A predecessor of pgRouting – pgDijkstra, written by Sylvain Pasche from **Camptocamp**, was later extended by **Orkney** and renamed to pgRouting. The project is now supported and maintained by **Georepublic**, **iMaptools** and a broad user community.

pgRouting is part of **OSGeo Community Projects** from the **OSGeo Foundation** and included on **OSGeo Live**.

Licensing

The following licenses can be found in pgRouting:

License	
GNU General Public License, version 2	Most features of pgRouting are available under **GNU General Public License, version 2**.
Boost Software License - Version 1.0	Some Boost extensions are available under **Boost Software License - Version 1.0**.
MIT-X License	Some code contributed by iMaptools.com is available under MIT-X license.
Creative Commons Attribution-Share Alike 3.0 License	The pgRouting Manual is licensed under a **Creative Commons Attribution-Share Alike 3.0 License**.

In general license information should be included in the header of each source file.

Contributors

This Release Contributors

Individuals (in alphabetical order)

Aasheesh Tiwari, Aditya Pratap Singh, Adrien Berchet, Cayetano Benavent, Gudesa Venkata Sai Akhil, Hang Wu, Maoguang Wang, Martha Vergara, Regina Obe, Rohith Reddy, Sourabh Garg, Virginia Vergara

And all the people that give us a little of their time making comments, finding issues, making pull requests etc. in any of our products: osm2pgrouting, pgRouting, pgRoutingLayer.

Corporate Sponsors (in alphabetical order)

These are corporate entities that have contributed developer time, hosting, or direct monetary funding to the pgRouting project:

- **Georepublic**
- **Google Summer of Code**
- **iMaptools**
- **Leopark**
- **Paragon Corporation**

Contributors Past & Present:

Individuals (in alphabetical order)

Aasheesh Tiwari, Aditya Pratap Singh, Adrien Berchet, Akio Takubo, Andrea Nardelli, Anthony Tasca, Anton Patrushev, Ashraf Hossain, Cayetano Benavent, Christian Gonzalez, Daniel Kastl, Dave Potts, David Techer, Denis Rykov, Ema Miyawaki, Florian Thurkow, Frederic Junod, Gerald Fenoy, Gudesa Venkata Sai Akhil, Hang Wu, Jay Mahadeokar, Jinfu Leng, Kai Behncke, Kishore Kumar, Ko Nagase, Manikata Kondeti, Mario Basa, Martin Wiesenhaan, Maxim Dubinin, Maoguang Wang, Mohamed Zia, Mukul Priya, Razequl Islam, Regina Obe, Rohith Reddy, Sarthak Agarwal, Sourabh Garg, Stephen Woodbridge, Sylvain Housseman, Sylvain Pasche, Vidhan Jain, Virginia Vergara

Corporate Sponsors (in alphabetical order)

These are corporate entities that have contributed developer time, hosting, or direct monetary funding to the pgRouting project:

- Camptocamp
- CSIS (University of Tokyo)
- Georepublic
- Google Summer of Code
- iMaptools
- Orkney
- Paragon Corporation
- Versaterm Inc.

More Information

- The latest software, documentation and news items are available at the pgRouting web site **https://pgrouting.org**.
- PostgreSQL database server at the PostgreSQL main site **https://www.postgresql.org**.
- PostGIS extension at the PostGIS project web site **https://postgis.net**.
- Boost C++ source libraries at **https://www.boost.org**.
- The Migration guide can be found at **https://github.com/pgRouting/pgrouting/wiki/Migration-Guide**.

Installation

Table of Contents

Instructions for downloading and installing binaries for different Operative systems instructions and additional notes and corrections not included in this documentation can be found in **Installation wiki**

To use pgRouting postGIS needs to be installed, please read the information about installation in this **Install Guide**

Short Version

Extracting the tar ball

```
tar xvfz pgrouting-3.0.0.tar.gz
cd pgrouting-3.0.0
```

To compile assuming you have all the dependencies in your search path:

```
mkdir build
cd build
cmake  ..
make
sudo make install
```

Once pgRouting is installed, it needs to be enabled in each individual database you want to use it in.

```
createdb routing
psql routing -c 'CREATE EXTENSION PostGIS'
psql routing -c 'CREATE EXTENSION pgRouting'
```

Get the sources

The pgRouting latest release can be found in **https://github.com/pgRouting/pgrouting/releases/latest**

wget

To download this release:

```
wget -O pgrouting-3.0.0.tar.gz https://github.com/pgRouting/pgrouting/archive/v3.0.0.tar.gz
```

Goto **Short Version** to the extract and compile instructions.

git

To download the repository

```
git clone git://github.com/pgRouting/pgrouting.git
cd pgrouting
git checkout v3.0.0
```

Goto **Short Version** to the compile instructions (there is no tar ball).

Enabling and upgrading in the database

Enabling the database

pgRouting is an extension and depends on postGIS. Enabling postGIS before enabling pgRouting in the database

```
CREATE EXTENSION postgis;
CREATE EXTENSION pgrouting;
```

Upgrading the database

To upgrade pgRouting in the database to version 3.0.0 use the following command:

```
ALTER EXTENSION pgrouting UPDATE TO "3.0.0";
```

More information can be found in **https://www.postgresql.org/docs/current/static/sql-createextension.html**

Dependencies

Compilation Dependencies

To be able to compile pgRouting, make sure that the following dependencies are met:

- C and C++0x compilers * g++ version >= 4.8
- Postgresql version >= 9.3
- The Boost Graph Library (BGL). Version >= 1.53
- CMake >= 3.2

optional dependencies

For user's documentation

- Sphinx >= 1.1
- Latex

For developer's documentation

- Doxygen >= 1.7

For testing

- pgtap
- pg_prove

For using:

- PostGIS version >= 2.2

Example: Installing dependencies on linux

Installing the compilation dependencies

Database dependencies

```
sudo apt-get install
    postgresql-10 \
    postgresql-server-dev-10 \
    postgresql-10-postgis
```

Build dependencies

```
sudo apt-get install
    cmake \
    g++ \
    libboost-graph-dev
```

Optional dependencies

For documentation and testing

```
sudo apt-get install -y python-sphinx \
    texlive \
    doxygen \
    libtap-parser-sourcehandler-pgtap-perl \
    postgresql-10-pgtap
```

Configuring

pgRouting uses the *cmake* system to do the configuration.

The build directory is different from the source directory

Create the build directory

```
$ mkdir build
```

Configurable variables

To see the variables that can be configured

```
$ cd build
$ cmake -L ..
```

Configuring The Documentation

Most of the effort of the documentation has being on the HTML files. Some variables for the documentation:

Variable	Default	Comment
WITH_DOC	BOOL=OFF	Turn on/off building the documentation
BUILD_HTML	BOOL=ON	If ON, turn on/off building HTML for user's documentation
BUILD_DOXY	BOOL=ON	If ON, turn on/off building HTML for developer's documentation
BUILD_LATEX	BOOL=OFF	If ON, turn on/off building PDF
BUILD_MAN	BOOL=OFF	If ON, turn on/off building MAN pages
DOC_USE_BOOTSTRAP	BOOL=OFF	If ON, use sphinx-bootstrap for HTML pages of the users documentation

Configuring with documentation

```
$ cmake -DWITH_DOC=ON ..
```

Note

Most of the effort of the documentation has being on the html files.

Building

Using `make` to build the code and the documentation

The following instructions start from *path/to/pgrouting/build*

```
$ make         # build the code but not the documentation
$ make doc     # build only the documentation
$ make all doc # build both the code and the documentation
```

We have tested on several platforms, For installing or reinstalling all the steps are needed.

Warning

The sql signatures are configured and build in the `cmake` command.

MinGW on Windows

```
$ mkdir build
$ cd build
$ cmake -G"MSYS Makefiles" ..
$ make
$ make install
```

Linux

The following instructions start from *path/to/pgrouting*

```
mkdir build
cd build
cmake  ..
make
sudo make install
```

When the configuration changes:

```
rm -rf build
```

and start the build process as mentioned above.

Testing

Currently there is no `make test` and testing is done as follows

The following instructions start from *path/to/pgrouting/*

```
tools/testers/doc_queries_generator.pl
createdb  -U <user> ___pgr___test___
sh ./tools/testers/pg_prove_tests.sh <user>
dropdb  -U <user> ___pgr___test___
```

See Also

Indices and tables

- **Index**
- **Search Page**

Support

pgRouting community support is available through the **pgRouting website**, **documentation**, tutorials, mailing lists and others. If you're looking for **commercial support**, find below a list of companies providing pgRouting development and consulting services.

Reporting Problems

Bugs are reported and managed in an **issue tracker**. Please follow these steps:

1. Search the tickets to see if your problem has already been reported. If so, add any extra context you might have found, or at least indicate that you too are having the problem. This will help us prioritize common issues.
2. If your problem is unreported, create a **new issue** for it.
3. In your report include explicit instructions to replicate your issue. The best tickets include the exact SQL necessary to

replicate a problem.
4. If you can test older versions of PostGIS for your problem, please do. On your ticket, note the earliest version the problem appears.
5. For the versions where you can replicate the problem, note the operating system and version of pgRouting, PostGIS and PostgreSQL.
6. It is recommended to use the following wrapper on the problem to pin point the step that is causing the problem.

```
SET client_min_messages TO debug;
  <your code>
SET client_min_messages TO notice;
```

Mailing List and GIS StackExchange

There are two mailing lists for pgRouting hosted on OSGeo mailing list server:

- User mailing list: **https://lists.osgeo.org/mailman/listinfo/pgrouting-users**
- Developer mailing list: **https://lists.osgeo.org/mailman/listinfo/pgrouting-dev**

For general questions and topics about how to use pgRouting, please write to the user mailing list.

You can also ask at **GIS StackExchange** and tag the question with `pgrouting`. Find all questions tagged with `pgrouting` under **http://gis.stackexchange.com/questions/tagged/pgrouting** or subscribe to the **pgRouting questions feed**.

Commercial Support

For users who require professional support, development and consulting services, consider contacting any of the following organizations, which have significantly contributed to the development of pgRouting:

Company	Offices in	Website
Georepublic	Germany, Japan	**https://georepublic.info**
iMaptools	United States	**https://imaptools.com**
Paragon Corporation	United States	**https://www.paragoncorporation.com**
Camptocamp	Switzerland, France	**https://www.camptocamp.com**
Netlab	Capranica, Italy	**https://www.osgeo.org/service-providers/netlab/**

- **Sample Data** that is used in the examples of this manual.

Sample Data

The documentation provides very simple example queries based on a small sample network. To be able to execute the sample queries, run the following SQL commands to create a table with a small network data set.

Create table

```
CREATE TABLE edge_table (
    id BIGSERIAL,
    dir character varying,
    source BIGINT,
    target BIGINT,
    cost FLOAT,
    reverse_cost FLOAT,
    capacity BIGINT,
    reverse_capacity BIGINT,
    category_id INTEGER,
    reverse_category_id INTEGER,
    x1 FLOAT,
    y1 FLOAT,
    x2 FLOAT,
    y2 FLOAT,
    the_geom geometry
);
```

Insert data

```
INSERT INTO edge_table (
    category_id, reverse_category_id,
    cost, reverse_cost,
    capacity, reverse_capacity,
    x1, y1,
    x2, y2) VALUES
(3, 1,    1,  1,  80, 130,   2,   0,    2, 1),
(3, 2,   -1,  1,  -1, 100,   2,   1,    3, 1),
(2, 1,   -1,  1,  -1, 130,   3,   1,    4, 1),
(2, 4,    1,  1, 100,  50,   2,   1,    2, 2),
(1, 4,    1, -1, 130,  -1,   3,   1,    3, 2),
(4, 2,    1,  1,  50, 100,   0,   2,    1, 2),
(4, 1,    1,  1,  50, 130,   1,   2,    2, 2),
(2, 1,    1,  1, 100, 130,   2,   2,    3, 2),
(1, 3,    1,  1, 130,  80,   3,   2,    4, 2),
(1, 4,    1,  1, 130,  50,   2,   2,    2, 3),
(1, 2,    1, -1, 130,  -1,   3,   2,    3, 3),
(2, 3,    1, -1, 100,  -1,   2,   3,    3, 3),
(2, 4,    1, -1, 100,  -1,   3,   3,    4, 3),
(3, 1,    1,  1,  80, 130,   2,   3,    2, 4),
(3, 4,    1,  1,  80,  50,   4,   2,    4, 3),
(3, 3,    1,  1,  80,  80,   4,   1,    4, 2),
(1, 2,    1,  1, 130, 100,   0.5, 3.5,  1.999999999999,3.5),
(4, 1,    1,  1,  50, 130,   3.5, 2.3,  3.5,4);
```

Updating geometry

```
UPDATE edge_table SET the_geom = st_makeline(st_point(x1,y1),st_point(x2,y2)),
dir = CASE WHEN (cost>0 AND reverse_cost>0) THEN 'B'   -- both ways
          WHEN (cost>0 AND reverse_cost<0) THEN 'FT'  -- direction of the LINESSTRING
          WHEN (cost<0 AND reverse_cost>0) THEN 'TF'  -- reverse direction of the LINESTRING
          ELSE '' END;                                -- unknown
```

Topology

- Before you test a routing function use this query to create a topology (fills the`source` and `target` columns).

```
SELECT pgr_createTopology('edge_table',0.001);
```

Points of interest

- When points outside of the graph.
- Used with the **withPoints - Family of functions** functions.

```
CREATE TABLE pointsOfInterest(
    pid BIGSERIAL,
    x FLOAT,
    y FLOAT,
    edge_id BIGINT,
    side CHAR,
    fraction FLOAT,
    the_geom geometry,
    newPoint geometry
);

INSERT INTO pointsOfInterest (x, y, edge_id, side, fraction) VALUES
(1.8, 0.4,   1, 'l', 0.4),
(4.2, 2.4,  15, 'r', 0.4),
(2.6, 3.2,  12, 'l', 0.6),
(0.3, 1.8,   6, 'r', 0.3),
(2.9, 1.8,   5, 'l', 0.8),
(2.2, 1.7,   4, 'b', 0.7);
UPDATE pointsOfInterest SET the_geom = st_makePoint(x,y);

UPDATE pointsOfInterest
    SET newPoint = ST_LineInterpolatePoint(e.the_geom, fraction)
    FROM edge_table AS e WHERE edge_id = id;
```

Restrictions

- Used with the **pgr_trsp - Turn Restriction Shortest Path (TRSP)** functions.

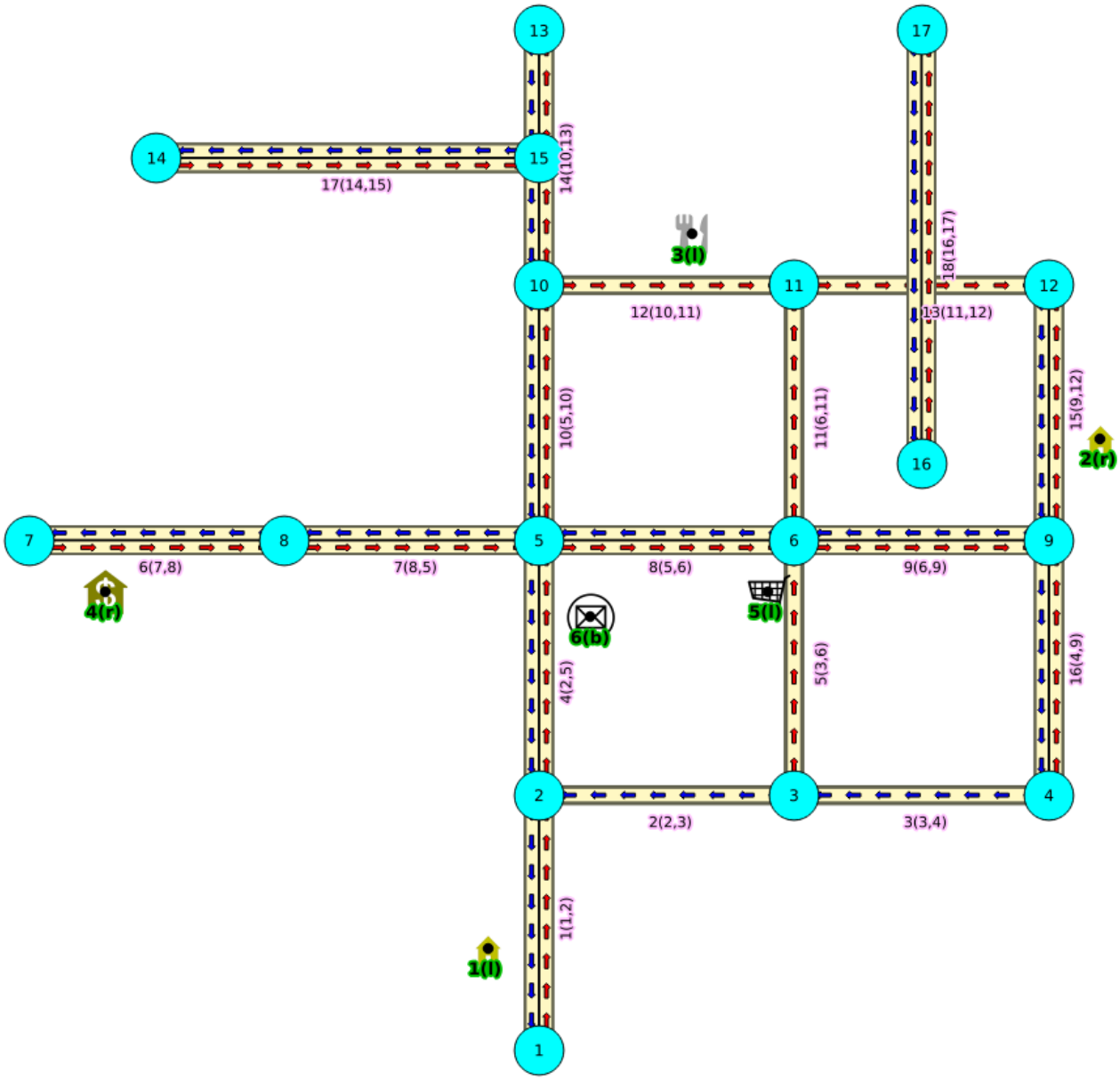
```
CREATE TABLE restrictions (
    rid BIGINT NOT NULL,
    to_cost FLOAT,
    target_id BIGINT,
    from_edge BIGINT,
    via_path TEXT
);

INSERT INTO restrictions (rid, to_cost, target_id, from_edge, via_path) VALUES
(1, 100,  7,  4, NULL),
(1, 100, 11,  8, NULL),
(1, 100, 10,  7, NULL),
(2,   4,  8,  3, 5),
(3, 100,  9, 16, NULL);

CREATE TABLE new_restrictions (
    id SERIAL PRIMARY KEY,
    path BIGINT[],
    cost float
);

INSERT INTO new_restrictions (path, cost) VALUES
(ARRAY[4, 7], 100),
(ARRAY[8, 11], 100),
(ARRAY[4, 8], 100),
(ARRAY[5, 9], 100),
(ARRAY[10, 12], 100),
(ARRAY[9, 15], 100),
(ARRAY[3, 5, 8], 100);
```

Images

- Red arrows correspond when `cost` > 0 in the edge table.
- Blue arrows correspond when `reverse_cost` > 0 in the edge table.
- Points are outside the graph.
- Click on the graph to enlarge.

Network for queries marked as `directed` and `cost` and `reverse_cost` columns are used

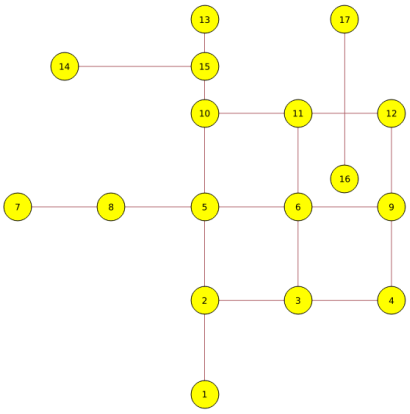
When working with city networks, this is recommended for point of view of vehicles.



Graph 1: Directed, with cost and reverse cost

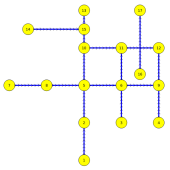
Network for queries marked as undirected **and** cost **and** reverse_cost **columns are used**

When working with city networks, this is recommended for point of view of pedestrians.



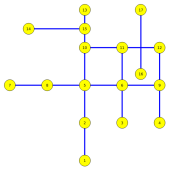
Graph 2: Undirected, with cost and reverse cost

Network for queries marked as directed and only cost column is used



Graph 3: Directed, with cost

Network for queries marked as undirected and only cost column is used



Graph 4: Undirected, with cost

Pick & Deliver Data

```
DROP TABLE IF EXISTS customer CASCADE;
CREATE table customer (
    id BIGINT not null primary key,
    x DOUBLE PRECISION,
    y DOUBLE PRECISION,
    demand INTEGER,
    opentime INTEGER,
    closetime INTEGER,
    servicetime INTEGER,
    pindex BIGINT,
    dindex BIGINT
);


INSERT INTO customer(
   id,     x,    y, demand, opentime, closetime, servicetime, pindex, dindex) VALUES
(  0,   40,   50,    0,    0,  1236,    0,    0,    0),
(  1,   45,   68,  -10,  912,  967,   90,   11,    0),
(  2,   45,   70,  -20,  825,  870,   90,    6,    0),
(  3,   42,   66,   10,   65,  146,   90,    0,   75),
(  4,   42,   68,  -10,  727,  782,   90,    9,    0),
(  5,   42,   65,   10,   15,   67,   90,    0,    7),
(  6,   40,   69,   20,  621,  702,   90,    0,    2),
(  7,   40,   66,  -10,  170,  225,   90,    5,    0),
(  8,   38,   68,   20,  255,  324,   90,    0,   10),
(  9,   38,   70,   10,  534,  605,   90,    0,    4),
( 10,   35,   66,  -20,  357,  410,   90,    8,    0),
( 11,   35,   69,   10,  448,  505,   90,    0,    1),
( 12,   25,   85,  -20,  652,  721,   90,   18,    0),
( 13,   22,   75,   30,   30,   92,   90,    0,   17),
( 14,   22,   85,  -40,  567,  620,   90,   16,    0),
( 15,   20,   80,  -10,  384,  429,   90,   19,    0),
( 16,   20,   85,   40,  475,  528,   90,    0,   14),
( 17,   18,   75,  -30,   99,  148,   90,   13,    0),
( 18,   15,   75,   20,  179,  254,   90,    0,   12),
( 19,   15,   80,   10,  278,  345,   90,    0,   15),
( 20,   30,   50,   10,   10,   73,   90,    0,   24),
( 21,   30,   52,  -10,  914,  965,   90,   30,    0),
( 22,   28,   52,  -20,  812,  883,   90,   28,    0),
( 23,   28,   55,   10,  732,  777,    0,    0,  103),
( 24,   25,   50,  -10,   65,  144,   90,   20,    0),
( 25,   25,   52,   40,  169,  224,   90,    0,   27),
( 26,   25,   55,  -10,  622,  701,   90,   29,    0),
( 27,   23,   52,  -40,  261,  316,   90,   25,    0),
( 28,   23,   55,   20,  546,  593,   90,    0,   22),
( 29,   20,   50,   10,  358,  405,   90,    0,   26),
( 30,   20,   55,   10,  449,  504,   90,    0,   21),
( 31,   10,   35,  -30,  200,  237,   90,   32,    0),
( 32,   10,   40,   30,   31,  100,   90,    0,   31),
( 33,    8,   40,   40,   87,  158,   90,    0,   37),
( 34,    8,   45,  -30,  751,  816,   90,   38,    0),
( 35,    5,   35,   10,  283,  344,   90,    0,   39),
( 36,    5,   45,   10,  665,  716,    0,    0,  105),
( 37,    2,   40,  -40,  383,  434,   90,   33,    0),
( 38,    0,   40,   30,  479,  522,   90,    0,   34),
( 39,    0,   45,  -10,  567,  624,   90,   35,    0),
( 40,   35,   30,   30,  264,  321,   90,   43,    0)
```

```
( 40,  35,  30,  -20,  264,  321,  90,  42,  0),
( 41,  35,  32,  -10,  166,  235,  90,  43,  0),
( 42,  33,  32,  20,  68,  149,  90,  0,  40),
( 43,  33,  35,  10,  16,  80,  90,  0,  41),
( 44,  32,  30,  10,  359,  412,  90,  0,  46),
( 45,  30,  30,  10,  541,  600,  90,  0,  48),
( 46,  30,  32,  -10,  448,  509,  90,  44,  0),
( 47,  30,  35,  -10,  1054,  1127,  90,  49,  0),
( 48,  28,  30,  -10,  632,  693,  90,  45,  0),
( 49,  28,  35,  10,  1001,  1066,  90,  0,  47),
( 50,  26,  32,  10,  815,  880,  90,  0,  52),
( 51,  25,  30,  10,  725,  786,  0,  0,  101),
( 52,  25,  35,  -10,  912,  969,  90,  50,  0),
( 53,  44,  5,  20,  286,  347,  90,  0,  58),
( 54,  42,  10,  40,  186,  257,  90,  0,  60),
( 55,  42,  15,  -40,  95,  158,  90,  57,  0),
( 56,  40,  5,  30,  385,  436,  90,  0,  59),
( 57,  40,  15,  40,  35,  87,  90,  0,  55),
( 58,  38,  5,  -20,  471,  534,  90,  53,  0),
( 59,  38,  15,  -30,  651,  740,  90,  56,  0),
( 60,  35,  5,  -40,  562,  629,  90,  54,  0),
( 61,  50,  30,  -10,  531,  610,  90,  67,  0),
( 62,  50,  35,  20,  262,  317,  90,  0,  68),
( 63,  50,  40,  50,  171,  218,  90,  0,  74),
( 64,  48,  30,  10,  632,  693,  0,  0,  102),
( 65,  48,  40,  10,  76,  129,  90,  0,  72),
( 66,  47,  35,  10,  826,  875,  90,  0,  69),
( 67,  47,  40,  10,  12,  77,  90,  0,  61),
( 68,  45,  30,  -20,  734,  777,  90,  62,  0),
( 69,  45,  35,  -10,  916,  969,  90,  66,  0),
( 70,  95,  30,  -30,  387,  456,  90,  81,  0),
( 71,  95,  35,  20,  293,  360,  90,  0,  77),
( 72,  53,  30,  -10,  450,  505,  90,  65,  0),
( 73,  92,  30,  -10,  478,  551,  90,  76,  0),
( 74,  53,  35,  -50,  353,  412,  90,  63,  0),
( 75,  45,  65,  -10,  997,  1068,  90,  3,  0),
( 76,  90,  35,  10,  203,  260,  90,  0,  73),
( 77,  88,  30,  -20,  574,  643,  90,  71,  0),
( 78,  88,  35,  20,  109,  170,  0,  0,  104),
( 79,  87,  30,  10,  668,  731,  90,  0,  80),
( 80,  85,  25,  -10,  769,  820,  90,  79,  0),
( 81,  85,  35,  30,  47,  124,  90,  0,  70),
( 82,  75,  55,  20,  369,  420,  90,  0,  85),
( 83,  72,  55,  -20,  265,  338,  90,  87,  0),
( 84,  70,  58,  20,  458,  523,  90,  0,  89),
( 85,  68,  60,  -20,  555,  612,  90,  82,  0),
( 86,  66,  55,  10,  173,  238,  90,  0,  91),
( 87,  65,  55,  20,  85,  144,  90,  0,  83),
( 88,  65,  60,  -10,  645,  708,  90,  90,  0),
( 89,  63,  58,  -20,  737,  802,  90,  84,  0),
( 90,  60,  55,  10,  20,  84,  90,  0,  88),
( 91,  60,  60,  -10,  836,  889,  90,  86,  0),
( 92,  67,  85,  20,  368,  441,  90,  0,  93),
( 93,  65,  85,  -20,  475,  518,  90,  92,  0),
( 94,  65,  82,  -10,  285,  336,  90,  96,  0),
( 95,  62,  80,  -20,  196,  239,  90,  98,  0),
( 96,  60,  80,  10,  95,  156,  90,  0,  94),
( 97,  60,  85,  30,  561,  622,  0,  0,  106),
( 98,  58,  75,  20,  30,  84,  90,  0,  95),
( 99,  55,  80,  -20,  743,  820,  90,  100,  0),
( 100,  55,  85,  20,  647,  726,  90,  0,  99),
( 101,  25,  30,  -10,  725,  786,  90,  51,  0),
( 102,  48,  30,  -10,  632,  693,  90,  64,  0),
( 103,  28,  55,  -10,  732,  777,  90,  23,  0),
( 104,  88,  35,  -20,  109,  170,  90,  78,  0),
( 105,  5,  45,  -10,  665,  716,  90,  36,  0),
( 106,  60,  85,  -30,  561,  622,  90,  97,  0);
```

Pgrouting Concepts

pgRouting Concepts

Contents

Getting Started

This is a simple guide to walk you through the steps of getting started with pgRouting. In this guide we will cover:

- Create a routing Database
- Load Data
- Build a Routing Topology
- Check the Routing Topology
- Compute a Path

Create a routing Database

The first thing we need to do is create a database and load pgrouting in the database. Typically you will create a database for each project. Once you have a database to work in, your can load your data and build your application in that database. This makes it easy to move your project later if you want to to say a production server.

For Postgresql 9.2 and later versions

```
createdb mydatabase
psql mydatabase -c "create extension postgis"
psql mydatabase -c "create extension pgrouting"
```

Load Data

How you load your data will depend in what form it comes it. There are various OpenSource tools that can help you, like:

osm2pgrouting:
- this is a tool for loading OSM data into postgresql with pgRouting requirements

shp2pgsql:
- this is the postgresql shapefile loader

ogr2ogr:
- this is a vector data conversion utility

osm2pgsql:
- this is a tool for loading OSM data into postgresql

So these tools and probably others will allow you to read vector data so that you may then load that data into your database as a table of some kind. At this point you need to know a little about your data structure and content. One easy way to browse your new data table is with pgAdmin3 or phpPgAdmin.

Build a Routing Topology

Next we need to build a topology for our street data. What this means is that for any given edge in your street data the ends of that edge will be connected to a unique node and to other edges that are also connected to that same unique node. Once all the edges are connected to nodes we have a graph that can be used for routing with pgrouting. We provide a tool that will help with this:

Note

this step is not needed if data is loaded with *osm2pgrouting*

```
select pgr_createTopology('myroads', 0.000001);
```

- **pgr_createTopology**

Check the Routing Topology

There are lots of possible sources for errors in a graph. The data that you started with may not have been designed with routing in mind. A graph has some very specific requirements. One is that it is *NODED*, this means that except for some very specific use cases, each road segment starts and ends at a node and that in general is does not cross another road segment that it should be connected to.

There can be other errors like the direction of a one-way street being entered in the wrong direction. We do not have tools to search for all possible errors but we have some basic tools that might help.

```
select pgr_analyzegraph('myroads', 0.000001);
select pgr_analyzeoneway('myroads',  s_in_rules, s_out_rules,
                         t_in_rules, t_out_rules
                         direction)
select pgr_nodeNetwork('myroads', 0.001);
```

- **pgr_analyzeGraph**
- **pgr_analyzeOneWay**
- **pgr_nodeNetwork**

Compute a Path

Once you have all the preparation work done above, computing a route is fairly easy. We have a lot of different algorithms that can work with your prepared road network. The general form of a route query is:

```
select pgr_dijkstra(`SELECT * FROM myroads', 1, 2)
```

As you can see this is fairly straight forward and you can look and the specific algorithms for the details of the signatures and how to use them. These results have information like edge id and/or the node id along with the cost or geometry for the step in the path from *start* to *end*. Using the ids you can join these result back to your edge table to get more information about each step in the path.

- **pgr_dijkstra**

Group of Functions

A function might have different overloads. Across this documentation, to indicate which overload we use the following terms:

- **One to One**
- **One to Many**
- **Many to One**
- **Many to Many**

Depending on the overload are the parameters used, keeping consistency across all functions.

One to One

When routing from:

- From **one** starting vertex
- to **one** ending vertex

One to Many

When routing from:

- From **one** starting vertex
- to **many** ending vertices

Many to One

When routing from:

- From **many** starting vertices
- to **one** ending vertex

Many to Many

When routing from:

- From **many** starting vertices
- to **many** ending vertices

Inner Queries

Description of the edges_sql query for dijkstra like functions

There are several kinds of valid inner queries and also the columns returned are depending of the function. Which kind of inner query will depend on the function(s) requirements. To simplify variety of types, `ANY-INTEGER` and `ANY-NUMERICAL` is used.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the edges_sql query for dijkstra like functions

Column	Type	Default	Description
id	`ANY-INTEGER`		Identifier of the edge.
source	`ANY-INTEGER`		Identifier of the first end point vertex of the edge.
target	`ANY-INTEGER`		Identifier of the second end point vertex of the edge.
cost	`ANY-NUMERICAL`		Weight of the edge *(source, target)* - When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	`ANY-NUMERICAL`	-1	Weight of the edge *(target, source)*, - When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the edges_sql query (id is not necessary)

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
source	`ANY-INTEGER`		Identifier of the first end point vertex of the edge.
target	`ANY-INTEGER`		Identifier of the second end point vertex of the edge.
cost	`ANY-NUMERICAL`		Weight of the edge *(source, target)* - When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	`ANY-NUMERICAL`	-1	Weight of the edge *(target, source)*, - When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Parameters

Parameter	Type	Default	Description
edges_sql	`TEXT`		SQL query as described above.

Parameter	Type	Default	Description
via_vertices	ARRAY[ANY-INTEGER]		Array of ordered vertices identifiers that are going to be visited.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as Undirected.
strict	BOOLEAN	false	• When false ignores missing paths returning all paths found • When true if a path is missing stops and returns *EMPTY SET*
U_turn_on_edge	BOOLEAN	true	• When true departing from a visited vertex will not try to avoid using the edge used to reach it. In other words, U turn using the edge with same *id* is allowed. • When false when a departing from a visited vertex tries to avoid using the edge used to reach it. In other words, U turn using the edge with same *id* is used when no other path is found.

edges_sql query for aStar - Family of functions and aStar - Family of functions functions

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

For pgr_pushRelabel, pgr_edmondsKarp, pgr_boykovKolmogorov :

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

For pgr_maxFlowMinCost - Experimental and pgr_maxFlowMinCost_Cost - Experimental:

Edges SQL:

an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Capacity of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Capacity of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* if it exists.
reverse_cost	ANY-NUMERICAL	0	Weight of the edge *(target, source)* if it exists.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
smallint, int, bigint, real, float

Description of the Points SQL query

points_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
pid	ANY-INTEGER	(optional) Identifier of the point. • If column present, it can not be NULL. • If column not present, a sequential identifier will be given automatically.
edge_id	ANY-INTEGER	Identifier of the "closest" edge to the point.
fraction	ANY-NUMERICAL	Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: • In the right, left of the edge or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.

Where:

ANY-INTEGER:
smallint, int, bigint
ANY-NUMERICAL:
smallint, int, bigint, real, float

Return columns & values

- **Return values for a path**
- **Return values for multiple paths from the same source and destination**
- **Description of the return values for a Cost Matrix - Category** function
- **Description of the Return Values**

There are several kinds of columns returned are depending of the function.

Return values for a path

Returns set of (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

Return values for multiple paths from the same source and destination

Returns set of `(seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)`

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same `start_vid` to `end_vid` combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

Description of the return values for a Cost Matrix - Category function

Returns SET OF `(start_vid, end_vid, agg_cost)`

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `end_vid`.

Description of the Return Values

For pgr_pushRelabel, pgr_edmondsKarp, pgr_boykovKolmogorov :

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
start_vid	BIGINT	Identifier of the first end point vertex of the edge.
end_vid	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction `start_vid`, `end_vid`).
residual_capacity	BIGINT	Residual capacity of the edge in the direction `start_vid`, `end_vid`).

For pgr_maxFlowMinCost - Experimental

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).

Column	Type	Description
source	BIGINT	Identifier of the first end point vertex of the edge.
target	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (source, target).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (source, target).
cost	FLOAT	The cost of sending this flow through the edge in the direction (source, target).
agg_cost	FLOAT	The aggregate cost.

Advanced Topics

Routing Topology

Overview

Typically when GIS files are loaded into the data database for use with pgRouting they do not have topology information associated with them. To create a useful topology the data needs to be "noded". This means that where two or more roads form an intersection there it needs to be a node at the intersection and all the road segments need to be broken at the intersection, assuming that you can navigate from any of these segments to any other segment via that intersection.

You can use the **graph analysis functions** to help you see where you might have topology problems in your data. If you need to node your data, we also have a function **pgr_nodeNetwork()** that might work for you. This function splits ALL crossing segments and nodes them. There are some cases where this might NOT be the right thing to do.

For example, when you have an overpass and underpass intersection, you do not want these noded, but pgr_nodeNetwork does not know that is the case and will node them which is not good because then the router will be able to turn off the overpass onto the underpass like it was a flat 2D intersection. To deal with this problem some data sets use z-levels at these types of intersections and other data might not node these intersection which would be ok.

For those cases where topology needs to be added the following functions may be useful. One way to prep the data for pgRouting is to add the following columns to your table and then populate them as appropriate. This example makes a lot of assumption like that you original data tables already has certain columns in it like `one_way`, `fcc`, and possibly others and that they contain specific data values. This is only to give you an idea of what you can do with your data.

```
ALTER TABLE edge_table
   ADD COLUMN source integer,
   ADD COLUMN target integer,
   ADD COLUMN cost_len double precision,
   ADD COLUMN cost_time double precision,
   ADD COLUMN rcost_len double precision,
   ADD COLUMN rcost_time double precision,
   ADD COLUMN x1 double precision,
   ADD COLUMN y1 double precision,
   ADD COLUMN x2 double precision,
   ADD COLUMN y2 double precision,
   ADD COLUMN to_cost double precision,
   ADD COLUMN rule text,
   ADD COLUMN isolated integer;

SELECT pgr_createTopology('edge_table', 0.000001, 'the_geom', 'id');
```

The function **pgr_createTopology** will create the `vertices_tmp` table and populate the `source` and `target` columns. The following example populated the remaining columns. In this example, the `fcc` column contains feature class code and the `CASE` statements converts it to an average speed.

```
UPDATE edge_table SET x1 = st_x(st_startpoint(the_geom)),
                      y1 = st_y(st_startpoint(the_geom)),
                      x2 = st_x(st_endpoint(the_geom)),
                      y2 = st_y(st_endpoint(the_geom)),
  cost_len  = st_length_spheroid(the_geom, 'SPHEROID["WGS84",6378137,298.25728]'),
  rcost_len = st_length_spheroid(the_geom, 'SPHEROID["WGS84",6378137,298.25728]'),
  len_km = st_length_spheroid(the_geom, 'SPHEROID["WGS84",6378137,298.25728]')/1000.0,
  len_miles = st_length_spheroid(the_geom, 'SPHEROID["WGS84",6378137,298.25728]')
            / 1000.0 * 0.6213712,
  speed_mph = CASE WHEN fcc='A10' THEN 65
                   WHEN fcc='A15' THEN 65
                   WHEN fcc='A20' THEN 55
                   WHEN fcc='A25' THEN 55
                   WHEN fcc='A30' THEN 45
                   WHEN fcc='A35' THEN 45
                   WHEN fcc='A40' THEN 35
                   WHEN fcc='A45' THEN 35
                   WHEN fcc='A50' THEN 25
                   WHEN fcc='A60' THEN 25
                   WHEN fcc='A61' THEN 25
                   WHEN fcc='A62' THEN 25
                   WHEN fcc='A64' THEN 25
                   WHEN fcc='A70' THEN 15
                   WHEN fcc='A69' THEN 10
                   ELSE null END,
  speed_kmh = CASE WHEN fcc='A10' THEN 104
                   WHEN fcc='A15' THEN 104
                   WHEN fcc='A20' THEN 88
                   WHEN fcc='A25' THEN 88
                   WHEN fcc='A30' THEN 72
                   WHEN fcc='A35' THEN 72
                   WHEN fcc='A40' THEN 56
                   WHEN fcc='A45' THEN 56
                   WHEN fcc='A50' THEN 40
                   WHEN fcc='A60' THEN 50
                   WHEN fcc='A61' THEN 40
                   WHEN fcc='A62' THEN 40
                   WHEN fcc='A64' THEN 40
                   WHEN fcc='A70' THEN 25
                   WHEN fcc='A69' THEN 15
                   ELSE null END;

-- UPDATE the cost information based on oneway streets

UPDATE edge_table SET
   cost_time = CASE
      WHEN one_way='TF' THEN 10000.0
      ELSE cost_len/1000.0/speed_kmh::numeric*3600.0
      END,
   rcost_time = CASE
      WHEN one_way='FT' THEN 10000.0
      ELSE cost_len/1000.0/speed_kmh::numeric*3600.0
      END;

-- clean up the database because we have updated a lot of records

VACUUM ANALYZE VERBOSE edge_table;
```

Now your database should be ready to use any (most?) of the pgRouting algorithms.

Graph Analytics

Overview

It is common to find problems with graphs that have not been constructed fully noded or in graphs with z-levels at intersection that have been entered incorrectly. An other problem is one way streets that have been entered in the wrong direction. We can not detect errors with respect to “ground” truth, but we can look for inconsistencies and some anomalies in a graph and report them for additional inspections.

We do not current have any visualization tools for these problems, but I have used mapserver to render the graph and highlight potential problem areas. Someone familiar with graphviz might contribute tools for generating images with that.

Analyze a Graph

With **pgr_analyzeGraph** the graph can be checked for errors. For example for table “mytab” that has “mytab_vertices_pgr” as the vertices table:

```
SELECT pgr_analyzeGraph('mytab', 0.000002);
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 158
NOTICE:                           Dead ends: 20028
NOTICE:  Potential gaps found near dead ends: 527
NOTICE:               Intersections detected: 2560
NOTICE:                     Ring geometries: 0
pgr_analyzeGraph
----------
  OK
(1 row)
```

In the vertices table “mytab_vertices_pgr”:

- Deadends are identified by `cnt=1`
- Potencial gap problems are identified with `chk=1`.

```
SELECT count(*) as deadends  FROM mytab_vertices_pgr WHERE cnt = 1;
deadends
----------
   20028
(1 row)

SELECT count(*) as gaps  FROM mytab_vertices_pgr WHERE chk = 1;
 gaps
 -----
   527
(1 row)
```

For isolated road segments, for example, a segment where both ends are deadends. you can find these with the following query:

```
SELECT *
   FROM mytab a, mytab_vertices_pgr b, mytab_vertices_pgr c
   WHERE a.source=b.id AND b.cnt=1 AND a.target=c.id AND c.cnt=1;
```

If you want to visualize these on a graphic image, then you can use something like mapserver to render the edges and the vertices and style based on `cnt` or if they are isolated, etc. You can also do this with a tool like graphviz, or geoserver or other similar tools.

Analyze One Way Streets

pgr_analyzeOneWay analyzes one way streets in a graph and identifies any flipped segments. Basically if you count the edges coming into a node and the edges exiting a node the number has to be greater than one.

This query will add two columns to the vertices_tmp table `ein int` and `eout int` and populate it with the appropriate counts. After running this on a graph you can identify nodes with potential problems with the following query.

The rules are defined as an array of text strings that if match the `col` value would be counted as true for the source or target in or out condition.

Example

Lets assume we have a table “st” of edges and a column “one_way” that might have values like:

- ‘FT’ - oneway from the source to the target node.
- ‘TF’ - oneway from the target to the source node.
- ‘B’ - two way street.
- ‘’ - empty field, assume twoway.
- <NULL> - NULL field, use two_way_if_null flag.

Then we could form the following query to analyze the oneway streets for errors.

```
SELECT pgr_analyzeOneway('mytab',
        ARRAY['', 'B', 'TF'],
        ARRAY['', 'B', 'FT'],
        ARRAY['', 'B', 'FT'],
        ARRAY['', 'B', 'TF'],
        );

-- now we can see the problem nodes
SELECT * FROM mytab_vertices_pgr WHERE ein=0 OR eout=0;

-- and the problem edges connected to those nodes
SELECT gid FROM mytab a, mytab_vertices_pgr b WHERE a.source=b.id AND ein=0 OR eout=0
UNION
SELECT gid FROM mytab a, mytab_vertices_pgr b WHERE a.target=b.id AND ein=0 OR eout=0;
```

Typically these problems are generated by a break in the network, the one way direction set wrong, maybe an error related to z-levels or a network that is not properly noded.

The above tools do not detect all network issues, but they will identify some common problems. There are other problems that are hard to detect because they are more global in nature like multiple disconnected networks. Think of an island with a road network that is not connected to the mainland network because the bridge or ferry routes are missing.

Performance Tips

- **For the Routing functions**
- **For the topology functions:**

For the Routing functions

To get faster results bound your queries to the area of interest of routing to have, for example, no more than one million rows.

Use an inner query SQL that does not include some edges in the routing function

```
SELECT id, source, target from edge_table WHERE
      id < 17 and
      the_geom  && (select st_buffer(the_geom,1) as myarea FROM  edge_table where id = 5)
```

Integrating the inner query to the pgRouting function:

```
SELECT * FROM pgr_dijkstra(
      'SELECT id, source, target from edge_table WHERE
         id < 17 and
         the_geom  && (select st_buffer(the_geom,1) as myarea FROM  edge_table where id = 5)',
   1, 2)
```

For the topology functions:

When “you know” that you are going to remove a set of edges from the edges table, and without those edges you are going to use a routing function you can do the following:

Analize the new topology based on the actual topology:

```
pgr_analyzegraph('edge_table',rows_where:='id < 17');
```

Or create a new topology if the change is permanent:

```
pgr_createTopology('edge_table',rows_where:='id < 17');
pgr_analyzegraph('edge_table',rows_where:='id < 17');
```

How to contribute

Wiki

- Edit an existing **pgRouting Wiki** page.
- Or create a new Wiki page
  - Create a page on the **pgRouting Wiki**
  - Give the title an appropriate name
- **Example**

Adding Functionaity to pgRouting

Consult the **developer’s documentation**

Indices and tables

- **Index**
- **Search Page**

Reference

- **pgr_version** - Get pgRouting's version information.
- **pgr_full_version** - Get pgRouting's details of version.

pgr_version

`pgr_version` — Query for pgRouting version information.

Availability

- Version 3.0.0
  - Breaking change on result columns
  - Support for old signature ends
- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6 2.5 2.4 2.3 2.2 2.1 2.0**

Description

Returns pgRouting version information.

Signature

```
TEXT pgr_version();
```

Example:
pgRouting Version for this documentatoin

```
SELECT pgr_version();
 pgr_version
-------------
 3.0.0
(1 row)
```

Result Columns

Type	Description
TEXT	pgRouting version

See Also

- **pgr_full_version**

Indices and tables

- **Index**
- **Search Page**

pgr_full_version

`pgr_full_version` — Get the details of pgRouting version information.

Availability

- Version 3.0.0
  - New **official** function

Support

- **Supported versions:** current(**3.0**)

Description

Get the details of pgRouting version information

Signatures

```
    pgr_full_version()
RETURNS RECORD OF (version, build_type, compile_date, library, system, PostgreSQL, compiler, boost, hash)
```

Example:
Information when this documentation was build

```
SELECT * FROM pgr_full_version();
 version | build_type | compile_date |    library    |         system          |                  postgresql                  | compiler | boost |   hash
---------+------------+--------------+---------------+-------------------------+----------------------------------------------+----------+-------+----------
 3.0.0   | Debug      | 2020/05/18   | pgrouting-3.0 | Linux-4.15.0-99-generic | PostgreSQL 12.2 (Ubuntu 12.2-2.pgdg18.04+1) | GNU-8.4.0 | 1.65.1 | d185af6e2
(1 row)
```

Result Columns

Column	Type	Description
version	TEXT	pgRouting version
build_type	TEXT	The Build type
compile_date	TEXT	Compilation date
library	TEXT	Library name and version
system	TEXT	Operative system
postgreSQL	TEXT	pgsql used
compiler	TEXT	Compiler and version
boost	TEXT	Boost version
hash	TEXT	Git hash of pgRouting build

See Also

- **pgr_version**

Indices and tables

- **Index**
- **Search Page**

Function Families

Function Families

All Pairs - Family of Functions

- **pgr_floydWarshall** - Floyd-Warshall's algorithm.
- **pgr_johnson** - Johnson's algorithm

aStar - Family of functions

- **pgr_aStar** - A* algorithm for the shortest path.
- **pgr_aStarCost** - Get the aggregate cost of the shortest paths.
- **pgr_aStarCostMatrix** - Get the cost matrix of the shortest paths.

Bidirectional A* - Family of functions

- **pgr_bdAstar** - Bidirectional A* algorithm for obtaining paths.
- **pgr_bdAstarCost** - Bidirectional A* algorithm to calculate the cost of the paths.
- **pgr_bdAstarCostMatrix** - Bidirectional A* algorithm to calculate a cost matrix of paths.

Bidirectional Dijkstra - Family of functions

- **pgr_bdDijkstra** - Bidirectional Dijkstra algorithm for the shortest paths.
- **pgr_bdDijkstraCost** - Bidirectional Dijkstra to calculate the cost of the shortest paths
- **pgr_bdDijkstraCostMatrix** - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

Components - Family of functions

- **pgr_connectedComponents** - Connected components of an undirected graph.
- **pgr_strongComponents** - Strongly connected components of a directed graph.
- **pgr_biconnectedComponents** - Biconnected components of an undirected graph.
- **pgr_articulationPoints** - Articulation points of an undirected graph.
- **pgr_bridges** - Bridges of an undirected graph.

Contraction - Family of functions

- **pgr_contraction**

Dijkstra - Family of functions

- **pgr_dijkstra** - Dijkstra's algorithm for the shortest paths.
- **pgr_dijkstraCost** - Get the aggregate cost of the shortest paths.
- **pgr_dijkstraCostMatrix** - Use pgr_dijkstra to create a costs matrix.
- **pgr_drivingDistance** - Use pgr_dijkstra to calculate catchament information.
- **pgr_KSP** - Use Yen algorithm with pgr_dijkstra to get the K shortest paths.

Flow - Family of functions

- **pgr_maxFlow** - Only the Max flow calculation using Push and Relabel algorithm.
- **pgr_boykovKolmogorov** - Boykov and Kolmogorov with details of flow on edges.
- **pgr_edmondsKarp** - Edmonds and Karp algorithm with details of flow on edges.
- **pgr_pushRelabel** - Push and relabel algorithm with details of flow on edges.
- Applications
  - **pgr_edgeDisjointPaths** - Calculates edge disjoint paths between two groups of vertices.
  - **pgr_maxCardinalityMatch** - Calculates a maximum cardinality matching in a graph.

Kruskal - Family of functions

- **pgr_kruskal**
- **pgr_kruskalBFS**
- **pgr_kruskalDD**
- **pgr_kruskalDFS**

Prim - Family of functions

- **pgr_prim**
- **pgr_primBFS**
- **pgr_primDD**
- **pgr_primDFS**

Topology - Family of Functions

- **pgr_createTopology** - to create a topology based on the geometry.
- **pgr_createVerticesTable** - to reconstruct the vertices table based on the source and target information.
- **pgr_analyzeGraph** - to analyze the edges and vertices of the edge table.
- **pgr_analyzeOneWay** - to analyze directionality of the edges.
- **pgr_nodeNetwork** -to create nodes to a not noded edge table.

Traveling Sales Person - Family of functions

- **pgr_TSP** - When input is given as matrix cell information.
- **pgr_TSPeuclidean** - When input are coordinates.

pgr_trsp - Turn Restriction Shortest Path (TRSP) - Turn Restriction Shortest Path (TRSP)

Functions by categories

Cost - Category

- **pgr_aStarCost**
- **pgr_dijkstraCost**

Cost Matrix - Category

- **pgr_aStarCostMatrix**
- **pgr_dijkstraCostMatrix**

Driving Distance - Category

- **pgr_drivingDistance** - Driving Distance based on Dijkstra's algorithm
- **pgr_primDD** - Driving Distance based on Prim's algorithm
- **pgr_kruskalDD** - Driving Distance based on Kruskal's algorithm
- Post pocessing
  - **pgr_alphaShape** - Alpha shape computation

K shortest paths - Category

- **pgr_KSP** - Yen's algorithm based on pgr_dijkstra

Spanning Tree - Category

- **Kruskal - Family of functions**
- **Prim - Family of functions**

All Pairs - Family of Functions

The following functions work on all vertices pair combinations

- **pgr_floydWarshall** - Floyd-Warshall's algorithm.
- **pgr_johnson** - Johnson's algorithm

pgr_floydWarshall

`pgr_floydWarshall` - Returns the sum of the costs of the shortest path for each pair of nodes in the graph using Floyd-Warshall algorithm.

Boost Graph Inside

Availability

- Version 2.2.0
  - Signature change
  - Old signature no longer supported
- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The Floyd-Warshall algorithm, also known as Floyd's algorithm, is a good choice to calculate the sum of the costs of the shortest path for each pair of nodes in the graph, for *dense graphs*. We use Boost's implementation which runs in $\Theta(V^3)$ time,

The main characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for each pair of nodes in the graph.
- Process is done only on edges with positive costs.
- Boost returns a $V \times V$ matrix, where the infinity values. Represent the distance between vertices for which there is no path.
  - We return only the non infinity values in form of a set of *(start_vid, end_vid, agg_cost)*.
- Let be the case the values returned are stored in a table, so the unique index would be the pair *(start_vid, end_vid)*.
- For the undirected graph, the results are symmetric.
  - The *agg_cost* of *(u, v)* is the same as for *(v, u)*.
- When *start_vid* = *end_vid*, the *agg_cost* = 0.
- **Recommended, use a bounding box of no more than 3500 edges.**

Signatures

Summary

```
pgr floydWarshall(edges_sql [, directed])
RETURNS SET OF (start_vid, end_vid,  agg_cost)
OR EMPTY SET
```

Using defaults

```
pgr_floydWarshall(edges_sql)
RETURNS SET OF (start_vid, end_vid,  agg_cost)
OR EMPTY SET
```

Example 1:
For vertices {1, 2, 3, 4} on a **directed** graph

```
SELECT * FROM pgr_floydWarshall(
    'SELECT id, source, target, cost FROM edge_table where id < 5'
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       5 |        2
         2 |       5 |        1
(3 rows)
```

Complete Signature

```
pgr_floydWarshall(edges_sql [, directed])
RETURNS SET OF (start_vid, end_vid,  agg_cost)
OR EMPTY SET
```

Example 2:
For vertices {1, 2, 3, 4} on an **undirected** graph

```
SELECT * FROM pgr_floydWarshall(
    'SELECT id, source, target, cost FROM edge_table where id < 5',
    false
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       5 |        2
         2 |       1 |        1
         2 |       5 |        1
         5 |       1 |        2
         5 |       2 |        1
(6 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	SQL query as described above.
directed	BOOLEAN	(optional) Default is true (is directed). When set to false the graph is considered as Undirected

Inner query

Description of the edges_sql query (id is not necessary)

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(start_vid, end_vid, agg_cost)`

Column	Type	Description
start_vid	`BIGINT`	Identifier of the starting vertex.
end_vid	`BIGINT`	Identifier of the ending vertex.
agg_cost	`FLOAT`	Total cost from `start_vid` to `end_vid`.

See Also

- **pgr_johnson**
- **Boost floyd-Warshall** algorithm
- Queries uses the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_johnson

`pgr_johnson` - Returns the sum of the costs of the shortest path for each pair of nodes in the graph using Floyd-Warshall algorithm.

Boost Graph Inside

Availability

- Version 2.2.0
  - Signature change
  - Old signature no longer supported
- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The Johnson algorithm, is a good choice to calculate the sum of the costs of the shortest path for each pair of nodes in the graph, for *sparse graphs*. It usees the Boost's implementation which runs in $O(VE\log V)$ time,

The main characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for each pair of nodes in the graph.
- Process is done only on edges with positive costs.
- Boost returns a $V \times V$ matrix, where the infinity values. Represent the distance between vertices for which there is no path.
  - We return only the non infinity values in form of a set of *(start_vid, end_vid, agg_cost)*.
- Let be the case the values returned are stored in a table, so the unique index would be the pair *(start_vid, end_vid)*.
- For the undirected graph, the results are symmetric.
  - The *agg_cost* of *(u, v)* is the same as for *(v, u)*.
- When *start_vid* = *end_vid*, the *agg_cost* = 0.

Signatures

Summary

```
pgr_johnson(edges_sql)
pgr johnson(edges_sql [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Using default

```
pgr_johnson(edges_sql)
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example 1:
For vertices {1, 2, 3, 4} on a **directed** graph

```
SELECT * FROM pgr_johnson(
    'SELECT source, target, cost FROM edge_table WHERE id < 5
        ORDER BY id'
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       5 |        2
         2 |       5 |        1
(3 rows)
```

Complete Signature

```
pgr_johnson(edges_sql[, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example 2:
For vertices {1, 2, 3, 4} on an **undirected** graph

```
SELECT * FROM pgr_johnson(
    'SELECT source, target, cost FROM edge_table WHERE id < 5
        ORDER BY id',
    false
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       5 |        2
         2 |       1 |        1
         2 |       5 |        1
         5 |       1 |        2
         5 |       2 |        1
(6 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	SQL query as described above.
directed	BOOLEAN	(optional) Default is true (is directed). When set to false the graph is considered as Undirected

Inner query

Description of the edges_sql query (id is not necessary)

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(start_vid, end_vid, agg_cost)`

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex.
end_vid	BIGINT	Identifier of the ending vertex.
agg_cost	FLOAT	Total cost from `start_vid` to `end_vid`.

See Also

- **pgr_floydWarshall**
- **Boost Johnson** algorithm implementation.
- Queries uses the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2**

Performance

The following tests:

- non server computer
- with AMD 64 CPU
- 4G memory
- trusty
- posgreSQL version 9.3

Data

The following data was used

```
BBOX="-122.8,45.4,-122.5,45.6"
wget --progress=dot:mega -O "sampledata.osm" "https://www.overpass-api.de/api/xapi?*[bbox=][@meta]"
```

Data processing was done with osm2pgrouting-alpha

```
createdb portland
psql -c "create extension postgis" portland
psql -c "create extension pgrouting" portland
osm2pgrouting -f sampledata.osm -d portland -s 0
```

Results

Test:
One

This test is not with a bounding box The density of the passed graph is extremely low. For each <SIZE> 30 tests were executed to get the average The tested query is:

```
SELECT count(*) FROM pgr_floydWarshall(
   'SELECT gid as id, source, target, cost, reverse_cost FROM ways where id <=  <SIZE>');

SELECT count(*) FROM pgr_johnson(
   'SELECT gid as id, source, target, cost, reverse_cost FROM ways where id <=  <SIZE>');
```

The results of this tests are presented as:

SIZE:
is the number of edges given as input.
EDGES:

is the total number of records in the query.
DENSITY:
is the density of the data $\frac{E}{V \times (V-1)}$.
OUT ROWS:
is the number of records returned by the queries.
Floyd-Warshall:
is the average execution time in seconds of pgr_floydWarshall.
Johnson:
is the average execution time in seconds of pgr_johnson.

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
500	500	0.18E-7	1346	0.14	0.13
1000	1000	0.36E-7	2655	0.23	0.18
1500	1500	0.55E-7	4110	0.37	0.34
2000	2000	0.73E-7	5676	0.56	0.37
2500	2500	0.89E-7	7177	0.84	0.51
3000	3000	1.07E-7	8778	1.28	0.68
3500	3500	1.24E-7	10526	2.08	0.95
4000	4000	1.41E-7	12484	3.16	1.24
4500	4500	1.58E-7	14354	4.49	1.47
5000	5000	1.76E-7	16503	6.05	1.78
5500	5500	1.93E-7	18623	7.53	2.03
6000	6000	2.11E-7	20710	8.47	2.37
6500	6500	2.28E-7	22752	9.99	2.68
7000	7000	2.46E-7	24687	11.82	3.12
7500	7500	2.64E-7	26861	13.94	3.60
8000	8000	2.83E-7	29050	15.61	4.09
8500	8500	3.01E-7	31693	17.43	4.63
9000	9000	3.17E-7	33879	19.19	5.34
9500	9500	3.35E-7	36287	20.77	6.24
10000	10000	3.52E-7	38491	23.26	6.51

Test:
Two

This test is with a bounding box The density of the passed graph higher than of the Test One. For each <SIZE> 30 tests were executed to get the average The tested edge query is:

```
WITH
   buffer AS (SELECT ST_Buffer(ST_Centroid(ST_Extent(the_geom)), SIZE) AS geom FROM ways),
   bbox AS (SELECT ST_Envelope(ST_Extent(geom)) as box from buffer)
SELECT gid as id, source, target, cost, reverse_cost FROM ways where the_geom && (SELECT box from bbox);
```

The tested queries

```
SELECT count(*) FROM pgr_floydWarshall(<edge query>)
SELECT count(*) FROM pgr_johnson(<edge query>)
```

The results of this tests are presented as:

SIZE:
is the size of the bounding box.
EDGES:
is the total number of records in the query.
DENSITY:
is the density of the data $\frac{E}{V \times (V-1)}$.
OUT ROWS:
is the number of records returned by the queries.
Floyd-Warshall:
is the average execution time in seconds of pgr_floydWarshall.
Johnson:
is the average execution time in seconds of pgr_johnson.

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
0.001	44	0.0608	1197	0.10	0.10

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
0.002	99	0.0251	4330	0.10	0.10
0.003	223	0.0122	18849	0.12	0.12
0.004	358	0.0085	71834	0.16	0.16
0.005	470	0.0070	116290	0.22	0.19
0.006	639	0.0055	207030	0.37	0.27
0.007	843	0.0043	346930	0.64	0.38
0.008	996	0.0037	469936	0.90	0.49
0.009	1146	0.0032	613135	1.26	0.62
0.010	1360	0.0027	849304	1.87	0.82
0.011	1573	0.0024	1147101	2.65	1.04
0.012	1789	0.0021	1483629	3.72	1.35
0.013	1975	0.0019	1846897	4.86	1.68
0.014	2281	0.0017	2438298	7.08	2.28
0.015	2588	0.0015	3156007	10.28	2.80
0.016	2958	0.0013	4090618	14.67	3.76
0.017	3247	0.0012	4868919	18.12	4.48

See Also

- **pgr_johnson**
- **pgr_floydWarshall**
- **Boost floyd-Warshall** algorithm

Indices and tables

- **Index**
- **Search Page**

aStar - Family of functions

The A* (pronounced “A Star”) algorithm is based on Dijkstra’s algorithm with a heuristic that allow it to solve most shortest path problems by evaluation only a sub-set of the overall graph.

- **pgr_aStar** - A* algorithm for the shortest path.
- **pgr_aStarCost** - Get the aggregate cost of the shortest paths.
- **pgr_aStarCostMatrix** - Get the cost matrix of the shortest paths.

pgr_aStar

`pgr_aStar` — Shortest path using A* algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **Proposed** functions:
    - pgr_aStar(One to Many)
    - pgr_aStar(Many to One)
    - pgr_aStar(Many to Many)
- Version 2.3.0
  - Signature change on pgr_astar(One to One)
    - Old signature no longer supported
- Version 2.0.0
  - **Official** pgr_aStar(One to One)

Support

- **Supported versions:** current(**3.0**) **2.6**

- **Unsupported versions: 2.5 2.4 2.3 2.2 2.1 2.0**

Description

The main characteristics are:

- Default kind of graph is **directed** when
  - `directed` flag is missing.
  - `directed` flag is set to true
- Unless specified otherwise, ordering is:
  - first by `start_vid` (if exists)
  - then by `end_vid`
- Values are returned when there is a path
- Let v and u be nodes on the graph:
  - If there is no path from v to u:
    - no corresponding row is returned
    - `agg_cost` from v to u is ∞
  - There is no path when $v = u$ therefore
    - no corresponding row is returned
    - `agg_cost` from v to u is 0
- Edges with negative costs are not included in the graph.
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.
- Running time: $O((E + V) * \log V)$
- The results are equivalent to the union of the results of the *pgr_aStar(* **One to One** *)* on the:
  - *pgr_aStar(* **One to Many** *)*
  - *pgr_aStar(* **Many to One** *)*
  - *pgr_aStar(* **Many to Many** *)*
- `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

Signatures

Summary

```
pgr_aStar(edges_sql, from_vid,  to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_aStar(edges_sql, from_vid,  to_vids [, directed] [, heuristic] [, factor] [, epsilon])
pgr_aStar(edges_sql, from_vids, to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_aStar(edges_sql, from_vids, to_vids [, directed] [, heuristic] [, factor] [, epsilon])

RETURNS SET OF (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)
OR EMPTY SET
```

Optional parameters are *named parameters* and have a default value.

Using defaults

```
pgr_aStar(edges_sql, from_vid, to_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertex 12 on a **directed** graph

```
SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, 12);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |   11 |    1 |        2
   4 |        4 |   11 |   13 |    1 |        3
   5 |        5 |   12 |   -1 |    0 |        4
(5 rows)
```

One to One

```
pgr_aStar(edges_sql, from_vid,  to_vid  [, directed] [, heuristic] [, factor] [, epsilon])

RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 12 on an **undirected** graph using heuristic 2

```
SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, 12,
    directed := false, heuristic := 2);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |    3 |    1 |        1
   3 |        3 |    4 |   16 |    1 |        2
   4 |        4 |    9 |   15 |    1 |        3
   5 |        5 |   12 |   -1 |    0 |        4
(5 rows)
```

One to many

```
pgr_aStar(edges_sql, from_vid,  to_vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices {3, 12} on a **directed** graph using heuristic 2

```
SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, ARRAY[3, 12], heuristic := 2);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |      12 |    2 |    4 |    1 |        0
   8 |        2 |      12 |    5 |   10 |    1 |        1
   9 |        3 |      12 |   10 |   12 |    1 |        2
  10 |        4 |      12 |   11 |   13 |    1 |        3
  11 |        5 |      12 |   12 |   -1 |    0 |        4
(11 rows)
```

Many to One

```
pgr_aStar(edges_sql, from_vids, to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices {7, 2} to vertex 12 on a **directed** graph using heuristic 0

```
SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    ARRAY[7, 2], 12, heuristic := 0);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+----------+------+------+------+----------
   1 |        1 |        2 |    2 |    4 |    1 |        0
   2 |        2 |        2 |    5 |   10 |    1 |        1
   3 |        3 |        2 |   10 |   12 |    1 |        2
   4 |        4 |        2 |   11 |   13 |    1 |        3
   5 |        5 |        2 |   12 |   -1 |    0 |        4
   6 |        1 |        7 |    7 |    6 |    1 |        0
   7 |        2 |        7 |    8 |    7 |    1 |        1
   8 |        3 |        7 |    5 |   10 |    1 |        2
   9 |        4 |        7 |   10 |   12 |    1 |        3
  10 |        5 |        7 |   11 |   13 |    1 |        4
  11 |        6 |        7 |   12 |   -1 |    0 |        5
(11 rows)
```

Many to Many

```
pgr_aStar(edges_sql, from_vids, to_vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices {7, 2} to vertices {3, 12} on a **directed** graph using heuristic 2

```
SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    ARRAY[7, 2], ARRAY[3, 12], heuristic := 2);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |         2 |      12 |    2 |    4 |    1 |        0
   8 |        2 |         2 |      12 |    5 |   10 |    1 |        1
   9 |        3 |         2 |      12 |   10 |   12 |    1 |        2
  10 |        4 |         2 |      12 |   11 |   13 |    1 |        3
  11 |        5 |         2 |      12 |   12 |   -1 |    0 |        4
  12 |        1 |         7 |       3 |    7 |    6 |    1 |        0
  13 |        2 |         7 |       3 |    8 |    7 |    1 |        1
  14 |        3 |         7 |       3 |    5 |    8 |    1 |        2
  15 |        4 |         7 |       3 |    6 |    9 |    1 |        3
  16 |        5 |         7 |       3 |    9 |   16 |    1 |        4
  17 |        6 |         7 |       3 |    4 |    3 |    1 |        5
  18 |        7 |         7 |       3 |    3 |   -1 |    0 |        6
  19 |        1 |         7 |      12 |    7 |    6 |    1 |        0
  20 |        2 |         7 |      12 |    8 |    7 |    1 |        1
  21 |        3 |         7 |      12 |    5 |   10 |    1 |        2
  22 |        4 |         7 |      12 |   10 |   12 |    1 |        3
  23 |        5 |         7 |      12 |   11 |   13 |    1 |        4
  24 |        6 |         7 |      12 |   12 |   -1 |    0 |        5
(24 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	**edges_sql** inner query.
from_vid	ANY-INTEGER	Starting vertex identifier. Parameter in: • **One to One** • **One to Many**
from_vids	ARRAY[ANY-INTEGER]	Array of starting vertices identifiers. Parameter in: • **Many to One** • **Many to Many**
to_vid	ANY-INTEGER	Ending vertex identifier. Parameter in: • **One to One** • **Many to One**
to_vids	ARRAY[ANY-INTEGER]	Array of ending vertices identifiers. Parameter in: • **One to Many** • **Many to Many**

Optional Parameters

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true the graph is considered as Directed. • When false the graph is considered as Undirected.
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. Default5 • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	FLOAT	1	For units manipulation. *factor* > 0. See **Factor**
epsilon	FLOAT	1	For less restricted results. *epsilon* >= 1.

Inner query
edges_sql

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.

Column	Type	Default	Description
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)`

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

See Also

- **aStar - Family of functions**
- **Sample Data**
- **https://www.boost.org/libs/graph/doc/astar_search.html**
- **https://en.wikipedia.org/wiki/A*_search_algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_aStarCost

`pgr_aStarCost` — Returns the aggregate cost shortest path using **pgr_aStar** algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5 2.4**

Description

The main characteristics are:

- Default kind of graph is **directed** when
  - `directed` flag is missing.
  - `directed` flag is set to true
- Unless specified otherwise, ordering is:
  - first by `start_vid` (if exists)
  - then by `end_vid`
- Values are returned when there is a path
- Let v and u be nodes on the graph:
  - If there is no path from v to u:
    - no corresponding row is returned
    - `agg_cost` from v to u is ∞
  - There is no path when $v = u$ therefore
    - no corresponding row is returned
    - `agg_cost` from v to u is 0
- Edges with negative costs are not included in the graph.
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.
- Running time: $O((E + V) * \log V)$
- The results are equivalent to the union of the results of the *pgr_aStarCost(* **One to One** *)* on the:
  - *pgr_aStarCost(* **One to Many** *)*
  - *pgr_aStarCost(* **Many to One** *)*
  - *pgr_aStarCost(* **Many to Many** *)*

Signatures

Summary

```
pgr_aStarCost(edges_sql, from_vid,  to_vid   [, directed] [, heuristic] [, factor] [, epsilon])
pgr_aStarCost(edges_sql, from_vid,  to_vids  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_aStarCost(edges_sql, from_vids, to_vid   [, directed] [, heuristic] [, factor] [, epsilon])
pgr_aStarCost(edges_sql, from_vids, to_vids  [, directed] [, heuristic] [, factor] [, epsilon])

RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Optional parameters are *named parameters* and have a default value.

Using defaults

```
pgr_aStarCost(edges_sql, start_vid, end_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertex 12 on a **directed** graph

```
SELECT * FROM pgr_aStarCost(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, 12);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |      12 |        4
(1 row)
```

One to One

```
pgr_aStarCost(edges_sql, from_vid,  to_vid   [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 12 on an **undirected** graph using heuristic 2

```
SELECT * FROM pgr_aStarCost(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, 12,
    directed := false, heuristic := 2);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |      12 |        4
(1 row)
```

One to many

```
pgr_aStarCost(edges_sql, from_vid,  to_vids  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices {3, 12} on a **directed** graph using heuristic 2

```
SELECT * FROM pgr_aStarCost(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, ARRAY[3, 12], heuristic := 2);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      12 |        4
(2 rows)
```

Many to One

```
pgr_aStarCost(edges_sql, from_vids, to_vid   [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertices {7, 2} to vertex 12 on a **directed** graph using heuristic 0

```
SELECT * FROM pgr_aStarCost(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    ARRAY[7, 2], 12, heuristic := 0);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |      12 |        4
         7 |      12 |        5
(2 rows)
```

Many to Many

```
pgr_aStarCost(edges_sql, from_vids, to_vids  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertices {7, 2} to vertices {3, 12} on a **directed** graph using heuristic 2

```
SELECT * FROM pgr_aStarCost(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    ARRAY[7, 2], ARRAY[3, 12], heuristic := 2);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      12 |        4
         7 |       3 |        6
         7 |      12 |        5
(4 rows)
```

Parameters

Parameter	Type	Description

Parameter	Type	Description
edges_sql	TEXT	**edges_sql** inner query.
from_vid	ANY-INTEGER	Starting vertex identifier. Parameter in: • **One to One** • **One to Many**
from_vids	ARRAY[ANY-INTEGER]	Array of starting vertices identifiers. Parameter in: • **Many to One** • **Many to Many**
to_vid	ANY-INTEGER	Ending vertex identifier. Parameter in: • **One to One** • **Many to One**
to_vids	ARRAY[ANY-INTEGER]	Array of ending vertices identifiers. Parameter in: • **One to Many** • **Many to Many**

Optional Parameters

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true the graph is considered as Directed. • When false the graph is considered as Undirected.
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. Default5 • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	FLOAT	1	For units manipulation. *factor* > 0. See **Factor**
epsilon	FLOAT	1	For less restricted results. *epsilon* >= 1.

Inner query
edges_sql

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

See Also

- **aStar - Family of functions**
- **Cost - Category**
- **Cost Matrix - Category**
- Examples use **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_aStarCostMatrix

pgr_aStarCostMatrix - Calculates the a cost matrix using **pgr_aStar**.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4**

Description

The main characteristics are:

- Using internaly the **pgr_aStar** algorithm
- Returns a cost matrix.
- No ordering is performed
- let *v* and *u* are nodes on the graph:
  - when there is no path from *v* to *u*:
    - no corresponding row is returned
    - cost from *v* to *u* is inf
  - when v = u then
    - no corresponding row is returned
    - cost from *v* to *u* is 0
- When the graph is **undirected** the cost matrix is symmetric

Signatures

Summary

```
pgr_aStarCostMatrix(edges_sql, vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Using defaults

```
pgr_aStarCostMatrix(edges_sql, vids)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

Cost matrix for vertices \{1, 2, 3, 4\} on a **directed** graph

```
SELECT * FROM pgr_aStarCostMatrix(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5)
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        6
         1 |       4 |        5
         2 |       1 |        1
         2 |       3 |        5
         2 |       4 |        4
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        3
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Complete Signature

```
pgr_aStarCostMatrix(edges_sql, vids, [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

Symmetric cost matrix for vertices \{1, 2, 3, 4\} on an **undirected** graph using heuristic 2

```
SELECT * FROM pgr_aStarCostMatrix(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
    directed := false, heuristic := 2
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        2
         1 |       4 |        3
         2 |       1 |        1
         2 |       3 |        1
         2 |       4 |        2
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        1
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	**edges_sql** inner query.
vids	ARRAY[ANY-INTEGER]	Array of vertices identifiers.

Optional Parameters

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true the graph is considered as Directed. • When false the graph is considered as Undirected.
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	FLOAT	1	For units manipulation. factor > 0. See **Factor**
epsilon	FLOAT	1	For less restricted results. epsilon >= 1.

Inner query
edges_sql

edges_sql:

an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF `(start_vid, end_vid, agg_cost)`

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `end_vid`.

Additional Examples

Example:
Use with **pgr_TSP**

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_aStarCostMatrix(
        'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
        (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
        directed:= false, heuristic := 2
    )
    $$,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    1 |    1 |        0
   2 |    2 |    1 |        1
   3 |    3 |    1 |        2
   4 |    4 |    3 |        3
   5 |    1 |    0 |        6
(5 rows)
```

See Also

- **aStar - Family of functions**
- **Cost - Category**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4**

General Information

The main Characteristics are:

- Default kind of graph is **directed** when
  - `directed` flag is missing.
  - `directed` flag is set to true
- Unless specified otherwise, ordering is:
  - first by `start_vid` (if exists)
  - then by `end_vid`
- Values are returned when there is a path
- Let v and u be nodes on the graph:
  - If there is no path from v to u:
    - no corresponding row is returned
    - `agg_cost` from v to u is ∞
  - There is no path when $v = u$ therefore
    - no corresponding row is returned
    - `agg_cost` from *v* to *u* is 0
- Edges with negative costs are not included in the graph.
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.
- Running time: $O((E + V) * \log V)$

Advanced documentation

The A* (pronounced "A Star") algorithm is based on Dijkstra's algorithm with a heuristic, that is an estimation of the remaining cost from the vertex to the goal, that allows to solve most shortest path problems by evaluation only a sub-set of the overall graph. Running time: $O((E + V) * \log V)$

Heuristic

Currently the heuristic functions available are:

- 0: $h(v) = 0$ (Use this value to compare with pgr_dijkstra)
- 1: $h(v) = abs(max(\Delta x, \Delta y))$
- 2: $h(v) = abs(min(\Delta x, \Delta y))$
- 3: $h(v) = \Delta x * \Delta x + \Delta y * \Delta y$
- 4: $h(v) = sqrt(\Delta x * \Delta x + \Delta y * \Delta y)$
- 5: $h(v) = abs(\Delta x) + abs(\Delta y)$

where $\Delta x = x_1 - x_0$ and $\Delta y = y_1 - y_0$

Factor

Analysis 1

Working with cost/reverse_cost as length in degrees, x/y in lat/lon: Factor = 1 (no need to change units)

Analysis 2

Working with cost/reverse_cost as length in meters, x/y in lat/lon: Factor = would depend on the location of the points:

Latitude	Conversion	Factor
45	1 longitude degree is 78846.81 m	78846
0	1 longitude degree is 111319.46 m	111319

Analysis 3

Working with cost/reverse_cost as time in seconds, x/y in lat/lon: Factor: would depend on the location of the points and on the average speed say 25m/s is the speed.

Latitude	Conversion	Factor
45	1 longitude degree is (78846.81m)/(25m/s)	3153 s
0	1 longitude degree is (111319.46 m)/(25m/s)	4452 s

See Also

See Also

- **pgr_aStar**
- **pgr_aStarCost**
- **pgr_aStarCostMatrix**
- **https://www.boost.org/libs/graph/doc/astar_search.html**
- **https://en.wikipedia.org/wiki/A*_search_algorithm**

Indices and tables

- **Index**
- **Search Page**

Bidirectional A* - Family of functions

- **pgr_bdAstar** - Bidirectional A* algorithm for obtaining paths.
- **pgr_bdAstarCost** - Bidirectional A* algorithm to calculate the cost of the paths.
- **pgr_bdAstarCostMatrix** - Bidirectional A* algorithm to calculate a cost matrix of paths.

pgr_bdAstar

`pgr_bdAstar` — Returns the shortest path using Bidirectional A* algorithm.

Boost Graph Inside

Availability:

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Signature change on pgr_bdAstar(One to One)
    - Old signature no longer supported
  - New **Proposed** functions:
    - pgr_bdAstar(One to Many)
    - pgr_bdAstar(Many to One)
    - pgr_bdAstar(Many to Many)
- Version 2.0.0
  - **Official** pgr_bdAstar(One to One)

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The main characteristics are:

- Default kind of graph is **directed** when
  - `directed` flag is missing.
  - `directed` flag is set to true
- Unless specified otherwise, ordering is:
  - first by `start_vid` (if exists)
  - then by `end_vid`
- Values are returned when there is a path
- Let v and u be nodes on the graph:
  - If there is no path from v to u:
    - no corresponding row is returned
    - `agg_cost` from v to u is ∞
  - There is no path when $v = u$ therefore
    - no corresponding row is returned
    - `agg_cost` from v to u is 0
- Edges with negative costs are not included in the graph.
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.

- Running time: O((E + V) * \log V)
- The results are equivalent to the union of the results of the *pgr_bdAStar(* **One to One** *)* on the:
  - *pgr_bdAstar(* **One to Many** *)*
  - *pgr_bdAstar(* **Many to One** *)*
  - *pgr_bdAstar(* **Many to Many** *)*
- start_vid and end_vid in the result is used to distinguish to which path it belongs.

Signature

Summary

```
pgr_bdAstar(edges_sql, from_vid,  to_vid,  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_bdAstar(edges_sql, from_vid,  to_vids  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_bdAstar(edges_sql, from_vids, to_vid   [, directed] [, heuristic] [, factor] [, epsilon])
pgr_bdAstar(edges_sql, from_vids, to_vids  [, directed] [, heuristic] [, factor] [, epsilon])

RETURNS SET OF (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)
OR EMPTY SET
```

Optional parameters are *named parameters* and have a default value.

Using defaults

```
pgr_bdAstar(edges_sql, start_vid, end_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Example:
From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_bdAstar(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)
```

One to One

```
pgr_bdAstar(edges_sql, from_vid,  to_vid,  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Example:
From vertex 2 to vertex 3 on a **directed** graph using heuristic 2

```
SELECT * FROM pgr_bdAstar(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    2, 3,
    true, heuristic := 2
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)
```

One to many

```
pgr_bdAstar(edges_sql, from_vid,  to_vids  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertices \{3, 11\} on a **directed** graph using heuristic 3 and factor 3.5

```
SELECT * FROM pgr_bdAstar(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    2, ARRAY[3, 11],
    heuristic := 3, factor := 3.5
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |      11 |    2 |    4 |    1 |        0
   8 |        2 |      11 |    5 |    8 |    1 |        1
   9 |        3 |      11 |    6 |   11 |    1 |        2
  10 |        4 |      11 |   11 |   -1 |    0 |        3
(10 rows)
```

Many to One

```
pgr_bdAstar(edges_sql, from_vids, to_vid   [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertex 3 on an **undirected** graph using heuristic 4

```
SELECT * FROM pgr_bdAstar(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    ARRAY[2, 7], 3,
    false, heuristic := 4
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    2 |    1 |        0
   2 |        2 |         2 |    3 |   -1 |    0 |        1
   3 |        1 |         7 |    7 |    6 |    1 |        0
   4 |        2 |         7 |    8 |    7 |    1 |        1
   5 |        3 |         7 |    5 |    8 |    1 |        2
   6 |        4 |         7 |    6 |    5 |    1 |        3
   7 |        5 |         7 |    3 |   -1 |    0 |        4
(7 rows)
```

Many to Many

```
pgr_bdAstar(edges_sql, from_vids, to_vids  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertices \{3, 11\} on a **directed** graph using factor 0.5

```
SELECT * FROM pgr_bdAstar(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    ARRAY[2, 7], ARRAY[3, 11],
    factor := 0.5
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |         2 |      11 |    2 |    4 |    1 |        0
   8 |        2 |         2 |      11 |    5 |    8 |    1 |        1
   9 |        3 |         2 |      11 |    6 |   11 |    1 |        2
  10 |        4 |         2 |      11 |   11 |   -1 |    0 |        3
  11 |        1 |         7 |       3 |    7 |    6 |    1 |        0
  12 |        2 |         7 |       3 |    8 |    7 |    1 |        1
  13 |        3 |         7 |       3 |    5 |    8 |    1 |        2
  14 |        4 |         7 |       3 |    6 |    9 |    1 |        3
  15 |        5 |         7 |       3 |    9 |   16 |    1 |        4
  16 |        6 |         7 |       3 |    4 |    3 |    1 |        5
  17 |        7 |         7 |       3 |    3 |   -1 |    0 |        6
  18 |        1 |         7 |      11 |    7 |    6 |    1 |        0
  19 |        2 |         7 |      11 |    8 |    7 |    1 |        1
  20 |        3 |         7 |      11 |    5 |    8 |    1 |        2
  21 |        4 |         7 |      11 |    6 |   11 |    1 |        3
  22 |        5 |         7 |      11 |   11 |   -1 |    0 |        4
(22 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	**edges_sql** inner query.
from_vid	ANY-INTEGER	Starting vertex identifier. Parameter in: • **One to One** • **One to Many**
from_vids	ARRAY[ANY-INTEGER]	Array of starting vertices identifiers. Parameter in: • **Many to One** • **Many to Many**
to_vid	ANY-INTEGER	Ending vertex identifier. Parameter in: • **One to One** • **Many to One**
to_vids	ARRAY[ANY-INTEGER]	Array of ending vertices identifiers. Parameter in: • **One to Many** • **Many to Many**

Optional Parameters

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true the graph is considered as Directed. • When false the graph is considered as Undirected.
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	FLOAT	1	For units manipulation. factor > 0. See **Factor**
epsilon	FLOAT	1	For less restricted results. epsilon >= 1.

Inner query
edges_sql

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)`

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same `start_vid` to `end_vid` combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

See Also

- **aStar - Family of functions**
- **Bidirectional A* - Family of functions**
- **Sample Data** network.
- **https://www.boost.org/libs/graph/doc/astar_search.html**
- **https://en.wikipedia.org/wiki/A*_search_algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_bdAstarCost

`pgr_bdAstarCost` — Returns the aggregate cost shortest path using **pgr_aStar** algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **Proposed** function
- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5**

Description

- Default kind of graph is **directed** when
  - `directed` flag is missing.
  - `directed` flag is set to true
- Unless specified otherwise, ordering is:
  - first by `start_vid` (if exists)
  - then by `end_vid`
- Values are returned when there is a path
- Let v and u be nodes on the graph:
  - If there is no path from v to u:
    - no corresponding row is returned
    - `agg_cost` from v to u is \infty
  - There is no path when v = u therefore
    - no corresponding row is returned
    - `agg_cost` from *v* to *u* is 0
- Edges with negative costs are not included in the graph.
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.
- Running time: O((E + V) * \log V)
- The results are equivalent to the union of the results of the *pgr_bdAstarCost(* **One to One** *)* on the:
  - *pgr_bdAstarCost(* **One to Many** *)*
  - *pgr_bdAstarCost(* **Many to One** *)*
  - *pgr_bdAstarCost(* **Many to Many** *)*

Signatures

Summary

```
pgr_bdAstarCost(edges_sql, from_vid,  to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_bdAstarCost(edges_sql, from_vid,  to_vids [, directed] [, heuristic] [, factor] [, epsilon])
pgr_bdAstarCost(edges_sql, from_vids, to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
pgr_bdAstarCost(edges_sql, from_vids, to_vids [, directed] [, heuristic] [, factor] [, epsilon])

RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Optional parameters are *named parameters* and have a default value.

Using defaults

```
pgr_bdAstarCost(edges_sql, from_vid,  to_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_bdAstarCost(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    2, 3
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
(1 row)
```

One to One

```
pgr_bdAstarCost(edges_sql, from_vid,  to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **directed** graph using heuristic 2

```
SELECT * FROM pgr_bdAstarCost(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    2, 3,
    true, heuristic := 2
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
(1 row)
```

One to many

```
pgr_bdAstarCost(edges_sql, from_vid,  to_vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex *2* to vertices \{3, 11\} on a **directed** graph using heuristic *3* and factor 3.5

```
SELECT * FROM pgr_bdAstarCost(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    2, ARRAY[3, 11],
    heuristic := 3, factor := 3.5
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      11 |        3
(2 rows)
```

Many to One

```
pgr_bdAstarCost(edges_sql, from_vids, to_vid  [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{7, 2\} to vertex 3 on a **undirected** graph using heuristic 4

```
SELECT * FROM pgr_bdAstarCost(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    ARRAY[2, 7], 3,
    false, heuristic := 4
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        1
         7 |       3 |        4
(2 rows)
```

Many to Many

```
pgr_bdAstarCost(edges_sql, from_vids, to_vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{7, 2\} to vertices \{3, 11\} on a **directed** using heuristic 5 and factor 0.5

```
SELECT * FROM pgr_bdAstarCost(
    'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
    FROM edge_table',
    ARRAY[2, 7], ARRAY[3, 11],
    factor := 0.5
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      11 |        3
         7 |       3 |        6
         7 |      11 |        4
(4 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	**edges_sql** inner query.
from_vid	ANY-INTEGER	Starting vertex identifier. Parameter in: • **One to One** • **One to Many**
from_vids	ARRAY[ANY-INTEGER]	Array of starting vertices identifiers. Parameter in: • **Many to One** • **Many to Many**
to_vid	ANY-INTEGER	Ending vertex identifier. Parameter in: • **One to One** • **Many to One**
to_vids	ARRAY[ANY-INTEGER]	Array of ending vertices identifiers. Parameter in: • **One to Many** • **Many to Many**

Optional Parameters

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true the graph is considered as Directed. • When false the graph is considered as Undirected.
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	FLOAT	1	For units manipulation. factor > 0. See **Factor**
epsilon	FLOAT	1	For less restricted results. epsilon >= 1.

Inner query
edges_sql

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.

Column	Type	Default	Description
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

See Also

- **Bidirectional A* - Family of functions**
- **Cost - Category**
- **Cost Matrix - Category**
- Examples use **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_bdAstarCostMatrix

pgr_bdAstarCostMatrix - Calculates the a cost matrix using **pgr_aStar**.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **Proposed** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5**

Description

The main characteristics are:

- Using internaly the **pgr_bdAstar** algorithm
- Returns a cost matrix.
- No ordering is performed
- let *v* and *u* are nodes on the graph:
  - when there is no path from *v* to *u*:
    - no corresponding row is returned
    - cost from *v* to *u* is \inf
  - when v = u then
    - no corresponding row is returned
    - cost from *v* to *u* is 0
- When the graph is **undirected** the cost matrix is symmetric

Signatures

Summary

```
pgr_bdAstarCostMatrix(edges_sql, vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Using defaults

```
pgr_bdAstarCostMatrix(edges_sql, vids)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

Cost matrix for vertices \{1, 2, 3, 4\} on a **directed** graph

```
SELECT * FROM pgr_bdAstarCostMatrix(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5)
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        6
         1 |       4 |        5
         2 |       1 |        1
         2 |       3 |        5
         2 |       4 |        4
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        3
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Complete Signature

```
pgr_bdAstarCostMatrix(edges_sql, vids [, directed] [, heuristic] [, factor] [, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

Symmetric cost matrix for vertices \{1, 2, 3, 4\} on an **undirected** graph using heuristic 2

```
SELECT * FROM pgr_bdAstarCostMatrix(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
    false
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        2
         1 |       4 |        3
         2 |       1 |        1
         2 |       3 |        1
         2 |       4 |        2
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        1
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	**edges_sql** inner query.
vids	ARRAY[ANY-INTEGER]	Array of vertices identifiers.

Optional Parameters

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true the graph is considered as Directed. • When false the graph is considered as Undirected.

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	FLOAT	1	For units manipulation. factor > 0. See **Factor**
epsilon	FLOAT	1	For less restricted results. epsilon >= 1.

Inner query
edges_sql

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with **pgr_TSP**

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_bdAstarCostMatrix(
        'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
        (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
        false
    )
    $$,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    1 |    1 |        0
   2 |    2 |    1 |        1
   3 |    3 |    1 |        2
   4 |    4 |    3 |        3
   5 |    1 |    0 |        6
(5 rows)
```

See Also

- **aStar - Family of functions**
- **Bidirectional A* - Family of functions**
- **Cost - Category**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5**

Description

Based on A* algorithm, the bidirectional search finds a shortest path from a starting vertex (`start_vid`) to an ending vertex (`end_vid`). It runs two simultaneous searches: one forward from the `start_vid`, and one backward from the `end_vid`, stopping when the two meet in the middle. This implementation can be used with a directed graph and an undirected graph.

The main Characteristics are:

- Process is done only on edges with positive costs.
- Values are returned when there is a path.
- When the starting vertex and ending vertex are the same, there is no path.
  - The *agg_cost* the non included values *(v, v)* is *0*
- When the starting vertex and ending vertex are the different and there is no path:
  - The *agg_cost* the non included values *(u, v)* is \infty
- Running time (worse case scenario): O((E + V) * \log V)
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_astar

Signatures

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
x1	ANY-NUMERICAL		X coordinate of *source* vertex.

Column	Type	Default	Description
y1	ANY-NUMERICAL		Y coordinate of *source* vertex.
x2	ANY-NUMERICAL		X coordinate of *target* vertex.
y2	ANY-NUMERICAL		Y coordinate of *target* vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
start_vid	ANY-INTEGER	Starting vertex identifier.
start_vids	ARRAY[ANY-INTEGER]	Starting vertices identifierers.
end_vid	ANY-INTEGER	Ending vertex identifier.
end_vids	ARRAY[ANY-INTEGER]	Ending vertices identifiers.

directed	BOOLEAN	• Optional.
		  • When `false` the graph is considered as Undirected.
		  • Default is `true` which considers the graph as Directed.

heuristic	INTEGER	(optional). Heuristic number. Current valid values 0~5. Default`5`
		  • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra)
		  • 1: h(v) abs(max(dx, dy))
		  • 2: h(v) abs(min(dx, dy))
		  • 3: h(v) = dx * dx + dy * dy
		  • 4: h(v) = sqrt(dx * dx + dy * dy)
		  • 5: h(v) = abs(dx) + abs(dy)

factor	FLOAT	(optional). For units manipulation. factor > 0. Default `1`. see **Factor**
epsilon	FLOAT	(optional). For less restricted results. epsilon >= 1. Default `1`.

See Also

Indices and tables

- **Index**
- **Search Page**

Bidirectional Dijkstra - Family of functions

- **pgr_bdDijkstra** - Bidirectional Dijkstra algorithm for the shortest paths.
- **pgr_bdDijkstraCost** - Bidirectional Dijkstra to calculate the cost of the shortest paths
- **pgr_bdDijkstraCostMatrix** - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

pgr_bdDijkstra

`pgr_bdDijkstra` — Returns the shortest path(s) using Bidirectional Dijkstra algorithm.

Boost Graph Inside

Availability:

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **Proposed** functions:
    - pgr_bdDijkstra(One to Many)
    - pgr_bdDijkstra(Many to One)

    - pgr_bdDijkstra(Many to Many)
- Version 2.4.0
  - Signature change on pgr_bdDijsktra(One to One)
    - Old signature no longer supported
- Version 2.0.0
  - **Official** pgr_bdDijkstra(One to One)

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The main characteristics are:

- Process is done only on edges with positive costs.
- Values are returned when there is a path.
- When the starting vertex and ending vertex are the same, there is no path.
  - The *agg_cost* the non included values *(v, v)* is *0*
- When the starting vertex and ending vertex are the different and there is no path:
  - The *agg_cost* the non included values *(u, v)* is \infty
- Running time (worse case scenario): O((V \log V + E))
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_dijkstra

Signatures

Summary

```
pgr_bdDijkstra(edges_sql, start_vid,  end_vid  [, directed])
pgr_bdDijkstra(edges_sql, start_vid,  end_vids [, directed])
pgr_bdDijkstra(edges_sql, start_vids, end_vid  [, directed])
pgr_bdDijkstra(edges_sql, start_vids, end_vids [, directed])

RETURNS SET OF (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)
OR EMPTY SET
```

Using defaults

```
pgr_bdDijkstra(edges_sql, start_vid, end_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertex 3

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)
```

One to One

```
pgr_bdDijkstra(edges_sql, start_vid, end_vid [, directed])
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertex 3 on an **undirected** graph

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)
```

One to many

```
pgr_bdDijkstra(edges_sql, start_vid, end_vids [, directed])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{3, 11\} on a **directed** graph

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3, 11]);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |      11 |    2 |    4 |    1 |        0
   8 |        2 |      11 |    5 |    8 |    1 |        1
   9 |        3 |      11 |    6 |   11 |    1 |        2
  10 |        4 |      11 |   11 |   -1 |    0 |        3
(10 rows)
```

Many to One

```
pgr_bdDijkstra(edges_sql, start_vids, end_vid [, directed])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 7], 3);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+----------+------+------+------+----------
   1 |        1 |        2 |    2 |    4 |    1 |        0
   2 |        2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |        2 |    6 |    9 |    1 |        2
   4 |        4 |        2 |    9 |   16 |    1 |        3
   5 |        5 |        2 |    4 |    3 |    1 |        4
   6 |        6 |        2 |    3 |   -1 |    0 |        5
   7 |        1 |        7 |    7 |    6 |    1 |        0
   8 |        2 |        7 |    8 |    7 |    1 |        1
   9 |        3 |        7 |    5 |    8 |    1 |        2
  10 |        4 |        7 |    6 |    9 |    1 |        3
  11 |        5 |        7 |    9 |   16 |    1 |        4
  12 |        6 |        7 |    4 |    3 |    1 |        5
  13 |        7 |        7 |    3 |   -1 |    0 |        6
(13 rows)
```

Many to Many

```
pgr_bdDijkstra(edges_sql, start_vids, end_vids [, directed])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertices \{3, 11\} on a **directed** graph

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 7], ARRAY[3, 11]);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |         2 |      11 |    2 |    4 |    1 |        0
   8 |        2 |         2 |      11 |    5 |    8 |    1 |        1
   9 |        3 |         2 |      11 |    6 |   11 |    1 |        2
  10 |        4 |         2 |      11 |   11 |   -1 |    0 |        3
  11 |        1 |         7 |       3 |    7 |    6 |    1 |        0
  12 |        2 |         7 |       3 |    8 |    7 |    1 |        1
  13 |        3 |         7 |       3 |    5 |    8 |    1 |        2
  14 |        4 |         7 |       3 |    6 |    9 |    1 |        3
  15 |        5 |         7 |       3 |    9 |   16 |    1 |        4
  16 |        6 |         7 |       3 |    4 |    3 |    1 |        5
  17 |        7 |         7 |       3 |    3 |   -1 |    0 |        6
  18 |        1 |         7 |      11 |    7 |    6 |    1 |        0
  19 |        2 |         7 |      11 |    8 |    7 |    1 |        1
  20 |        3 |         7 |      11 |    5 |   10 |    1 |        2
  21 |        4 |         7 |      11 |   10 |   12 |    1 |        3
  22 |        5 |         7 |      11 |   11 |   -1 |    0 |        4
(22 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same start_vid to end_vid combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**

Column	Type	Description
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

See Also

- The queries use the **Sample Data** network.
- **Bidirectional Dijkstra - Family of functions**
- **https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf**
- **https://en.wikipedia.org/wiki/Bidirectional_search**

Indices and tables

- **Index**
- **Search Page**

pgr_bdDijkstraCost

`pgr_bdDijkstraCost` — Returns the shortest path(s)'s cost using Bidirectional Dijkstra algorithm.

Boost Graph Inside

Availability:

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5**

Description

The main characteristics are:

- Process is done only on edges with positive costs.
- Values are returned when there is a path.
- When the starting vertex and ending vertex are the same, there is no path.
  - The *agg_cost* the non included values *(v, v)* is *0*
- When the starting vertex and ending vertex are the different and there is no path:
  - The *agg_cost* the non included values *(u, v)* is \infty
- Running time (worse case scenario): O((V \log V + E))
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_dijkstra

Signatures

Summary

```
pgr_bdDijkstraCost(edges_sql, from_vid,  to_vid  [, directed])
pgr_bdDijkstraCost(edges_sql, from_vid,  to_vids [, directed])
pgr_bdDijkstraCost(edges_sql, from_vids, to_vid  [, directed])
pgr_bdDijkstraCost(edges_sql, from_vids, to_vids [, directed])

RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Using default

```
pgr_bdDijkstraCost(edges_sql, from_vid,  to_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
(1 row)
```

One to One

```
pgr_bdDijkstraCost(edges_sql, from_vid,  to_vid  [, directed])
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **undirected** graph

```
SELECT * FROM pgr_bdDijkstraCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    false
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        1
(1 row)
```

One to Many

```
pgr_bdDijkstraCost(edges_sql, from_vid,  to_vids [, directed])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{3, 11\} on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3, 11]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      11 |        3
(2 rows)
```

Many to One

```
pgr_bdDijkstraCost(edges_sql, from_vids, to_vids [, directed])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 7], 3);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         7 |       3 |        6
(2 rows)
```

Many to Many

```
pgr_bdDijkstraCost(edges_sql, start_vids, end_vids [, directed])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertices \{3, 11\} on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 7], ARRAY[3, 11]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      11 |        3
         7 |       3 |        6
         7 |      11 |        4
(4 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

See Also

- The queries use the **Sample Data** network.
- **pgr_bdDijkstra**
- **https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf**
- **https://en.wikipedia.org/wiki/Bidirectional_search**

Indices and tables

- **Index**
- **Search Page**

`pgr_bdDijkstraCostMatrix` - Calculates the a cost matrix using **pgr_bdDijkstra**.

Boost Graph Inside

Availability:

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **proposed** function
- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5**

Description

The main characteristics are:

- Process is done only on edges with positive costs.
- Values are returned when there is a path.
- When the starting vertex and ending vertex are the same, there is no path.
  - The *agg_cost* the non included values *(v, v)* is *0*
- When the starting vertex and ending vertex are the different and there is no path:
  - The *agg_cost* the non included values *(u, v)* is \infty
- Running time (worse case scenario): O((V \log V + E))
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_dijkstra
- Returns a cost matrix.

Signatures

Summary

```
pgr_bdDijkstraCostMatrix(edges_sql, start_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Using default

```
pgr_bdDijkstraCostMatrix(edges_sql, start_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:
Cost matrix for vertices \{1, 2, 3, 4\} on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5)
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        6
         1 |       4 |        5
         2 |       1 |        1
         2 |       3 |        5
         2 |       4 |        4
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        3
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Complete Signature

```
pgr_bdDijkstraCostMatrix(edges_sql, start_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:
Symmetric cost matrix for vertices \{1, 2, 3, 4\} on an **undirected** graph

```
SELECT * FROM pgr_bdDijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
    false
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        2
         1 |       4 |        3
         2 |       1 |        1
         2 |       3 |        1
         2 |       4 |        2
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        1
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of the vertices.
directed	BOOLEAN	(optional). When false the graph is considered as Undirected. Default is true which considers the graph as Directed.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with tsp

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_bdDijkstraCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
        false
    )
    $$,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    1 |    1 |        0
   2 |    2 |    1 |        1
   3 |    3 |    1 |        2
   4 |    4 |    3 |        3
   5 |    1 |    0 |        6
(5 rows)
```

See Also

- **pgr_bdDijkstra**
- **Cost Matrix - Category**
- **pgr_TSP**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5**

Synopsis

Based on Dijkstra's algorithm, the bidirectional search finds a shortest path a starting vertex `start_vid`) to an ending vertex (`end_vid`). It runs two simultaneous searches: one forward from the source, and one backward from the target, stopping when the two meet in the middle. This implementation can be used with a directed graph and an undirected graph.

Characteristics

The main Characteristics are:

- Process is done only on edges with positive costs.
- Values are returned when there is a path.
- When the starting vertex and ending vertex are the same, there is no path.
  - The *agg_cost* the non included values *(v, v)* is *0*
- When the starting vertex and ending vertex are the different and there is no path:
  - The *agg_cost* the non included values *(u, v)* is \infty
- Running time (worse case scenario): O((V \log V + E))
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_dijkstra

See Also

Indices and tables

- **Index**
- **Search Page**

Components - Family of functions

- **pgr_connectedComponents** - Connected components of an undirected graph.
- **pgr_strongComponents** - Strongly connected components of a directed graph.
- **pgr_biconnectedComponents** - Biconnected components of an undirected graph.
- **pgr_articulationPoints** - Articulation points of an undirected graph.
- **pgr_bridges** - Bridges of an undirected graph.

pgr_connectedComponents

`pgr_connectedComponents` — Connected components of an undirected graph using a DFS-based approach.

Boost Graph Inside

Availability

- Version 3.0.0
  - Return columns change:
    - `n_seq` is removed
    - `seq` changed type to `BIGINT`
  - **Official** function
- Version 2.5.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions: 2.6 2.5**

Description

A connected component of an undirected graph is a set of vertices that are all reachable from each other.

The main characteristics are:

- The signature is for an **undirected** graph.
- Components are described by vertices
- The returned values are ordered:
  - *component* ascending
  - *node* ascending
- Running time: O(V + E)

Signatures

```
pgr_connectedComponents(edges_sql)

RETURNS SET OF (seq, component, node)
OR EMPTY SET
```

Example:
The connected components of the graph

```
SELECT * FROM pgr_connectedComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | component | node
-----+-----------+------
   1 |         1 |    1
   2 |         1 |    2
   3 |         1 |    3
   4 |         1 |    4
   5 |         1 |    5
   6 |         1 |    6
   7 |         1 |    7
   8 |         1 |    8
   9 |         1 |    9
  10 |         1 |   10
  11 |         1 |   11
  12 |         1 |   12
  13 |         1 |   13
  14 |        14 |   14
  15 |        14 |   15
  16 |        16 |   16
  17 |        16 |   17
(17 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	`TEXT`		Inner query as described below.

Inner query

edges SQL:

an SQL query of an **undirected** graph, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, component, node)

Column	Type	Description
seq	BIGINT	Sequential value starting from **1**.
component	BIGINT	Component identifier. It is equal to the minimum node identifier in the component.
node	BIGINT	Identifier of the vertex that belongs to **component**.

See Also

- **Components - Family of functions**
- The queries use the **Sample Data** network.
- Boost: **Connected components**
- wikipedia: **Connected component**

Indices and tables

- **Index**
- **Search Page**

pgr_strongComponents

pgr_strongComponents — Strongly connected components of a directed graph using Tarjan's algorithm based on DFS.

Boost Graph Inside

Availability

- Version 3.0.0
  - Return columns change:
    - n_seq is removed
    - seq changed type to BIGINT
  - **Official** function
- Version 2.5.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5**

Description

A strongly connected component of a directed graph is a set of vertices that are all reachable from each other.

The main characteristics are:

- The signature is for a **directed** graph.
- Components are described by vertices
- The returned values are ordered:
  - *component* ascending
  - *node* ascending
- Running time: O(V + E)

Signatures

```
pgr_strongComponents(Edges SQL)

RETURNS SET OF (seq, component, node)
OR EMPTY SET
```

Example:
The strong components of the graph

```
SELECT * FROM pgr_strongComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | component | node
-----+-----------+------
   1 |         1 |    1
   2 |         1 |    2
   3 |         1 |    3
   4 |         1 |    4
   5 |         1 |    5
   6 |         1 |    6
   7 |         1 |    7
   8 |         1 |    8
   9 |         1 |    9
  10 |         1 |   10
  11 |         1 |   11
  12 |         1 |   12
  13 |         1 |   13
  14 |        14 |   14
  15 |        14 |   15
  16 |        16 |   16
  17 |        16 |   17
(17 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		Inner query as described below.

Inner query

edges SQL:
an SQL query of a **directed** graph, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, component, node)

Column	Type	Description
seq	BIGINT	Sequential value starting from **1**.
component	BIGINT	Component identifier. It is equal to the minimum node identifier in the component.
node	BIGINT	Identifier of the vertex that belongs to**component**.

See Also

- **Components - Family of functions**
- The queries use the **Sample Data** network.
- Boost: **Strong components**
- wikipedia: **Strongly connected component**

Indices and tables

- **Index**
- **Search Page**

pgr_biconnectedComponents

pgr_biconnectedComponents — Return the biconnected components of an undirected graph. In particular, the algorithm implemented by Boost.Graph.

Boost Graph Inside

Availability

- Version 3.0.0
  - Return columns change:
    - n_seq is removed
    - seq changed type to BIGINT
  - **Official** function
- Version 2.5.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions: 2.6 2.5**

Description

The biconnected components of an undirected graph are the maximal subsets of vertices such that the removal of a vertex from particular component will not disconnect the component. Unlike connected components, vertices may belong to multiple biconnected components. Vertices can be present in multiple biconnected components, but each edge can only be contained in a single biconnected component.

The main characteristics are:

- The signature is for an**undirected** graph.
- Components are described by edges.
- The returned values are ordered:
  - *component* ascending.
  - *edge* ascending.
- Running time: O(V + E)

Signatures

```
pgr_biconnectedComponents(Edges SQL)

RETURNS SET OF (seq, component, edge)
OR EMPTY SET
```

Example:
The biconnected components of the graph

```
SELECT * FROM pgr_biconnectedComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | component | edge
-----+-----------+------
   1 |         1 |    1
   2 |         2 |    2
   3 |         2 |    3
   4 |         2 |    4
   5 |         2 |    5
   6 |         2 |    8
   7 |         2 |    9
   8 |         2 |   10
   9 |         2 |   11
  10 |         2 |   12
  11 |         2 |   13
  12 |         2 |   15
  13 |         2 |   16
  14 |         6 |    6
  15 |         7 |    7
  16 |        14 |   14
  17 |        17 |   17
  18 |        18 |   18
(18 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		Inner query as described below.

Inner query

edges SQL:
an SQL query of an **undirected** graph, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, component, edge)

Column	Type	Description
seq	BIGINT	Sequential value starting from **1**.
component	BIGINT	Component identifier. It is equal to the minimum edge identifier in the component.
edge	BIGINT	Identifier of the edge.

See Also

- **Components - Family of functions**
- The queries use the **Sample Data** network.
- Boost: **Biconnected components**
- wikipedia: **Biconnected component**

Indices and tables

- **Index**
- **Search Page**

`pgr_articulationPoints` - Return the articulation points of an undirected graph.

Boost Graph Inside

Availability

- Version 3.0.0
  - Return columns change: `seq` is removed
  - **Official** function
- Version 2.5.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5**

Description

Those vertices that belong to more than one biconnected component are called articulation points or, equivalently, cut vertices. Articulation points are vertices whose removal would increase the number of connected components in the graph. This implementation can only be used with an undirected graph.

The main characteristics are:

- The signature is for an **undirected** graph.
- The returned values are ordered:
  - *node* ascending
- Running time: O(V + E)

Signatures

```
pgr_articulationPoints(Edges SQL)

RETURNS SET OF (node)
OR EMPTY SET
```

Example:
The articulation points of the graph

```
SELECT * FROM pgr_articulationPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 node
------
    2
    5
    8
   10
(4 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	`TEXT`		Inner query as described below.

Inner query

edges SQL:
an SQL query of an **undirected** graph, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	`ANY-INTEGER`		Identifier of the edge.
source	`ANY-INTEGER`		Identifier of the first end point vertex of the edge.
target	`ANY-INTEGER`		Identifier of the second end point vertex of the edge.

Column	Type	Default	Description
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (node)

Column	Type	Description
node	BIGINT	Identifier of the vertex.

See Also

- **Components - Family of functions**
- The queries use the **Sample Data** network.
- Boost: **Biconnected components & articulation points**
- wikipedia: **Biconnected component**

Indices and tables

- **Index**
- **Search Page**

pgr_bridges

pgr_bridges - Return the bridges of an undirected graph.

Boost Graph Inside

Availability

- Version 3.0.0
  - Return columns change: seq is removed
  - **Official** function
- Version 2.5.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5**

Description

A bridge is an edge of an undirected graph whose deletion increases its number of connected components. This implementation can only be used with an undirected graph.

The main characteristics are:

- The signature is for an **undirected** graph.
- The returned values are ordered:
  - *edge* ascending
- Running time: O(E * (V + E))

Signatures

```
pgr_bridges(Edges SQL)

RETURNS SET OF (edge)
OR EMPTY SET
```

Example:
The bridges of the graph

```
SELECT * FROM pgr_bridges(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 edge
------
    1
    6
    7
   14
   17
   18
(6 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		Inner query as described below.

Inner query

edges SQL:
an SQL query of an **undirected** graph, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (edge)

Column	Type	Description
edge	BIGINT	Identifier of the edge that is a bridge.

See Also

- **https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5**

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		Inner query as described below.

Inner query

Edges SQL:
an SQL query which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

pgr_connectedComponents & pgr_strongComponents

Returns set of (seq, component, node)

Column	Type	Description
seq	BIGINT	Sequential value starting from **1**.
component	BIGINT	Component identifier. It is equal to the minimum node identifier in the component.
node	BIGINT	Identifier of the vertex that belongs to**component**.

pgr_biconnectedComponents

Returns set of (seq, component, edge)

Column	Type	Description
seq	BIGINT	Sequential value starting from **1**.
component	BIGINT	Component identifier. It is equal to the minimum edge identifier in the component.
edge	BIGINT	Identifier of the edge.

pgr_articulationPoints

Returns set of (node)

Column	Type	Description
node	BIGINT	Identifier of the vertex.

pgr_bridges

Returns set of (edge)

Column	Type	Description
edge	BIGINT	Identifier of the edge that is a bridge.

See Also

Indices and tables

- **Index**
- **Search Page**

Contraction - Family of functions

- **pgr_contraction**

pgr_contraction

`pgr_contraction` — Performs graph contraction and returns the contracted vertices and edges.

Boost Graph Inside

Availability

- Version 3.0.0
  - Return columns change: `seq` is removed
  - Name change from `pgr_contractGraph`
  - Bug fixes
  - **Official** function
- Version 2.3.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3**

Description

Contraction reduces the size of the graph by removing some of the vertices and edges and, for example, might add edges that represent a sequence of original edges decreasing the total time and space used in graph algorithms.

The main Characteristics are:

- Process is done only on edges with positive costs.
- Does not return the full contracted graph
  - Only changes on the graph are returned
- Currnetly there are two types of contraction methods
  - Dead End Contraction
  - Linear Contraction
- The returned values include
  - the added edges by linear contraction.
  - the modified vertices by dead end contraction.
- The returned values are ordered as follows:
  - column *id* ascending when type = *v*
  - column *id* descending when type = *e*

Signatures

Summary

The pgr_contraction function has the following signature:

```
pgr_contraction(Edges SQL, Contraction order [, max_cycles] [, forbidden_vertices] [, directed])
RETURNS SETOF (type, id, contracted_vertices, source, target, cost)
```

Example:
Making a dead end contraction and a linear contraction with vertex 2 forbidden from being contracted

```
SELECT * FROM pgr_contraction(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
ARRAY[1, 2], forbidden_vertices:=ARRAY[2]);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 v    |  2 | {1}                 |     -1 |     -1 |   -1
 v    |  5 | {7,8}               |     -1 |     -1 |   -1
 v    | 10 | {13}                |     -1 |     -1 |   -1
 v    | 15 | {14}                |     -1 |     -1 |   -1
 v    | 17 | {16}                |     -1 |     -1 |   -1
(5 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	SQL query as described in **Inner query**
Ccontraction Order	ARRAY[ANY-INTEGER]	Ordered contraction operations. • 1 = Dead end contraction • 2 = Linear contraction

Optional Parameters

Column	Type	Default	Description
forbidden_vertices	ARRAY[ANY-INTEGER]	Empty	Identifiers of vertices forbidden from contraction.
max_cycles	INTEGER	1	Number of times the contraction operations on*contraction_order* will be performed.
directed	BOOLEAN	true	• When true the graph is considered as*Directed*. • When false the graph is considered as*Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

RETURNS SETOF (type, id, contracted_vertices, source, target, cost)

The function returns a single row. The columns of the row are:

Column	Type	Description
type	TEXT	Type of the*id*. • **'v'** when the row is a vertex. • **'e'** when the row is an edge.

id	BIGINT	All numbers on this column are DISTINCT
		• When type = **'v'**.
		  • Identifier of the modified vertex.
		• When type = **'e'**.
		  • Decreasing sequence starting from **-1**.
		  • Representing a pseudo *id* as is not incorporated in the set of original edges.

contracted_vertices	ARRAY[BIGINT]	Array of contracted vertex identifiers.

Column	Type	Description
source	BIGINT	• When type = **'v'**: -1 • When type = **'e'**: Identifier of the source vertex of the current edge (source, target).
target	BIGINT	• When type = **'v'**: -1 • When type = **'e'**: Identifier of the target vertex of the current edge (source, target).
cost	FLOAT	• When type = **'v'**: -1 • When type = **'e'**: Weight of the current edge (source, target).

Additional Examples

Example:
Only dead end contraction

```
SELECT * FROM pgr_contraction(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table',
ARRAY[1]);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 v    |  2 | {1}                 |     -1 |     -1 |   -1
 v    |  5 | {7,8}               |     -1 |     -1 |   -1
 v    | 10 | {13}                |     -1 |     -1 |   -1
 v    | 15 | {14}                |     -1 |     -1 |   -1
 v    | 17 | {16}                |     -1 |     -1 |   -1
(5 rows)
```

Example:
Only linear contraction

```
SELECT * FROM pgr_contraction(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table',
ARRAY[2]);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 e    | -1 | {8}                 |      5 |      7 |    2
 e    | -2 | {8}                 |      7 |      5 |    2
(2 rows)
```

See Also

- **Contraction - Family of functions**

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5 2.4 2.3 2.2**

Introduction

In large graphs, like the road graphs, or electric networks, graph contraction can be used to speed up some graph algorithms. Contraction reduces the size of the graph by removing some of the vertices and edges and, for example, might add edges that represent a sequence of original edges decreasing the total time and space used in graph algorithms.

This implementation gives a flexible framework for adding contraction algorithms in the future, currently, it supports two algorithms:

1. Dead end contraction
2. Linear contraction

Allowing the user to:

- Forbid contraction on a set of nodes.
- Decide the order of the contraction algorithms and set the maximum number of times they are to be executed.

Dead end contraction

In the algorithm, dead end contraction is represented by 1.

Dead end

In case of an undirected graph, a node is considered a *dead end* node when

- **The number of adjacent vertices is 1.**

In case of a directed graph, a node is considered a *dead end* node when

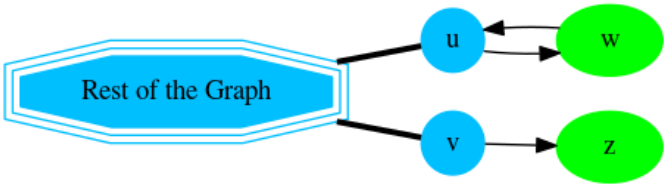
- **The number of adjacent vertices is 1.**
  - **There are no outgoing edges and has at least one incoming edge.**
  - **There are no incoming edges and has at least one outgoing edge.**

When the conditions are true then the **Operation: Dead End Contraction** can be done.

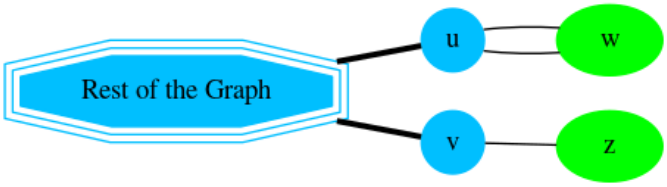
The number of adjacent vertices is 1.

- The green nodes are **dead end** nodes
- The blue nodes have an unlimited number of incoming and outgoing edges.

Directed graph



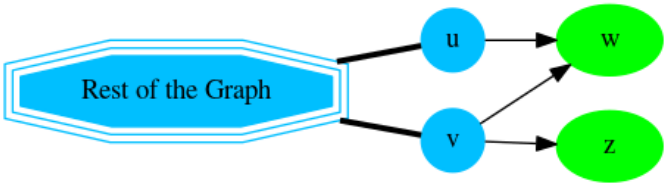
Undirected graph



There are no outgoing edges and has at least one incoming edge.

- The green nodes are **dead end** nodes
- The blue nodes have an unlimited number of incoming and outgoing edges.

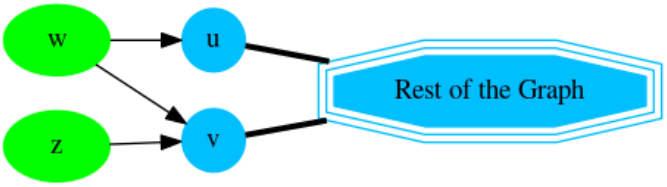
Directed graph



There are no incoming edges and has at least one outgoing edge.

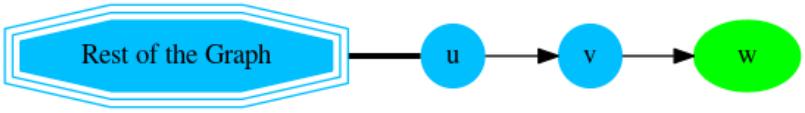
- The green nodes are **dead end** nodes
- The blue nodes have an unlimited number of incoming and outgoing edges.
- Considering that the nodes are *dead starts* nodes

Directed graph

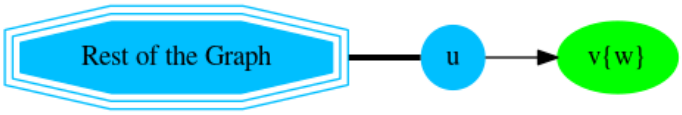


Operation: Dead End Contraction

The dead end contraction will stop until there are no more dead end nodes. For example from the following graph where `w` is the **dead end** node:



After contracting `w`, node `v` is now a **dead end** node and is contracted:



After contracting `v`, stop. Node `u` has the information of nodes that were contrcted.

Node `u` has the information of nodes that were contracted.

Linear contraction

In the algorithm, linear contraction is represented by 2.

Linear

In case of an undirected graph, a node is considered a*linear* node when

- **The number of adjacent vertices is 2.**

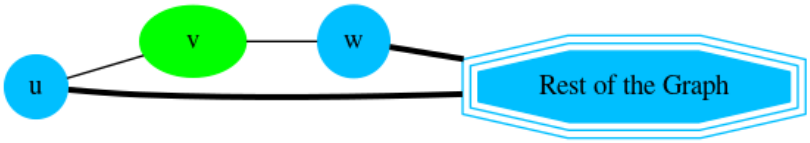
In case of a directed graph, a node is considered a*linear* node when

- **The number of adjacent vertices is 2.**
  - **Linearity is symmetrical**

The number of adjacent vertices is 2.
- The green nodes are **linear** nodes
- The blue nodes have an unlimited number of incoming and outgoing edges.

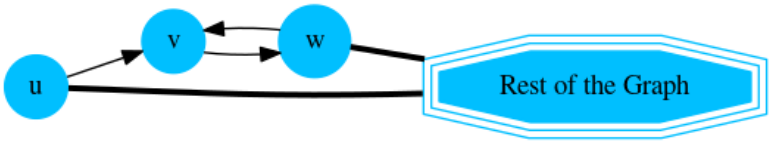
Directed

Undirected



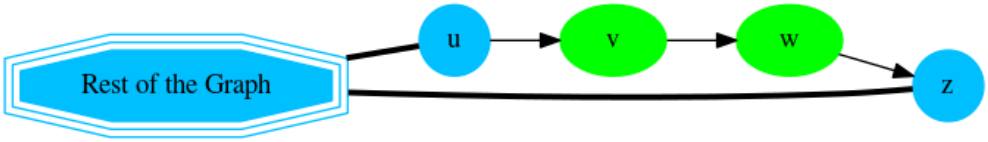
Linearity is symmetrical

Using a contra example, vertex `v` is not linear because it's not possible to go from`w` to `u` via `v`.



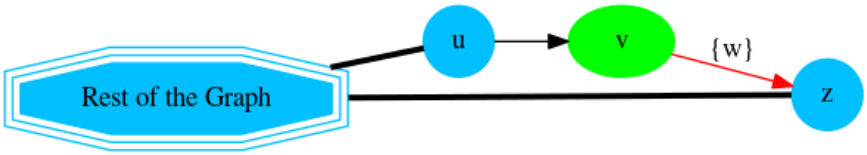
Operation: Linear Contraction

The linear contraction will stop until there are no more linear nodes. For example from the following graph where`v` and `w` are **linear** nodes:



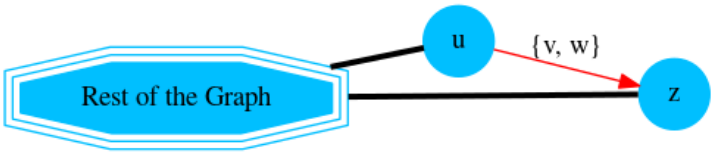
After contracting `w`,

- The vertex `w` is removed from the graph
  - The edges v \rightarrow w and w \rightarrow z are removed from the graph.
- A new edge v \rightarrow z is inserted represented with red color.



Contracting `v`:

- The vertex `v` is removed from the graph
  - The edges u \rightarrow v and v \rightarrow z are removed from the graph.
- A new edge u \rightarrow z is inserted represented with red color.



Edge u \rightarrow z has the information of nodes that were contracted.

The cycle

Contracting a graph, can be done with more than one operation. The order of the operations affect the resulting contracted graph, after applying one operation, the set of vertices that can be contracted by another operation changes.

This implementation, cycles `max_cycles` times through `operations_order` .

```
<input>
do max_cycles times {
   for (operation in operations_order)
    { do operation }
}
<output>
```

Contracting Sample Data

In this section, building and using a contracted graph will be shown by example.

- The **Sample Data** for an undirected graph is used
- a dead end operation first followed by a linear operation.

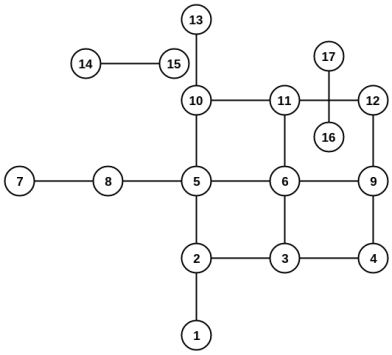
Construction of the graph in the database

Original Data

The following query shows the original data involved in the contraction operation.

```
SELECT id, source, target, cost, reverse_cost FROM edge_table;
 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
  1 |      1 |      2 |    1 |            1
  2 |      2 |      3 |   -1 |            1
  3 |      3 |      4 |   -1 |            1
  4 |      2 |      5 |    1 |            1
  5 |      3 |      6 |    1 |           -1
  6 |      7 |      8 |    1 |            1
  7 |      8 |      5 |    1 |            1
  8 |      5 |      6 |    1 |            1
  9 |      6 |      9 |    1 |            1
 10 |      5 |     10 |    1 |            1
 11 |      6 |     11 |    1 |           -1
 12 |     10 |     11 |    1 |           -1
 13 |     11 |     12 |    1 |           -1
 14 |     10 |     13 |    1 |            1
 15 |      9 |     12 |    1 |            1
 16 |      4 |      9 |    1 |            1
 17 |     14 |     15 |    1 |            1
 18 |     16 |     17 |    1 |            1
(18 rows)
```

The original graph:



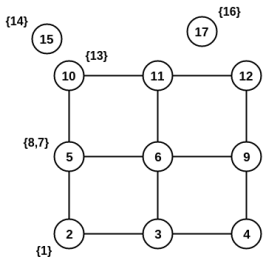
Contraction Results

The results do not represent the contracted graph. They represent the changes done to the graph after applying the contraction algorithm.

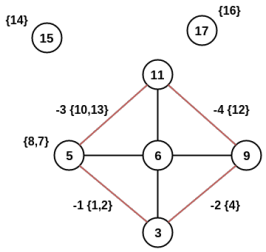
Observe that vertices, for example, 6 do not appear in the results because it was not affected by the contraction algorithm.

```
SELECT * FROM pgr_contraction(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    array[1,2], directed:=false);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 v    |  5 | {7,8}               |     -1 |     -1 |   -1
 v    | 15 | {14}                |     -1 |     -1 |   -1
 v    | 17 | {16}                |     -1 |     -1 |   -1
 e    | -1 | {1,2}               |      3 |      5 |    2
 e    | -2 | {4}                 |      3 |      9 |    2
 e    | -3 | {10,13}             |      5 |     11 |    2
 e    | -4 | {12}                |      9 |     11 |    2
(7 rows)
```

After doing the dead end contraction operation:



After doing the linear contraction operation to the graph above:



The process to create the contraction graph on the database:

- **Add additional columns**
- **Store contraction information**
- **Update the vertices and edge tables**

Add additional columns

Adding extra columns to the `edge_table` and `edge_table_vertices_pgr` tables, where:

Column	Description
contracted_vertices	The vertices set belonging to the vertex/edge
is_contracted	On the *vertex* table - when `true` the vertex is contracted, its not part of the contracted graph. - when `false` the vertex is not contracted, its part of the contracted graph.

Column	Description
is_new	On the *edge* table: • when `true` the edge was generated by the contraction algorithm. its part of the contracted graph. • when `false` the edge is an original edge, might be or not part of the contracted graph.

```
ALTER TABLE edge_table_vertices_pgr ADD is_contracted BOOLEAN DEFAULT false;
ALTER TABLE
ALTER TABLE edge_table_vertices_pgr ADD contracted_vertices BIGINT[];
ALTER TABLE
ALTER TABLE edge_table ADD is_new BOOLEAN DEFAULT false;
ALTER TABLE
ALTER TABLE edge_table ADD contracted_vertices BIGINT[];
ALTER TABLE
```

Store contraction information

Store the **contraction results** in a table

```
SELECT * INTO contraction_results
FROM pgr_contraction(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table',
   array[1,2], directed:=false);
SELECT 7
```

Update the vertices and edge tables

Update the *vertex* table using the contraction information

Use `edge_table_vertices_pgr.is_contracted` to indicate the vertices that are contracted.

```
UPDATE edge_table_vertices_pgr
SET is_contracted = true
WHERE id IN (SELECT  unnest(contracted_vertices) FROM  contraction_results);
UPDATE 10
```

Add to `edge_table_vertices_pgr.contracted_vertices` the contracted vertices belonging to the vertices.

```
UPDATE edge_table_vertices_pgr
SET contracted_vertices = contraction_results.contracted_vertices
FROM contraction_results
WHERE type = 'v' AND edge_table_vertices_pgr.id = contraction_results.id;
UPDATE 3
```

The modified `edge_table_vertices_pgr`.

```
SELECT id, contracted_vertices, is_contracted
FROM edge_table_vertices_pgr
ORDER BY id;
 id | contracted_vertices | is_contracted
----+---------------------+---------------
  1 |                     | t
  2 |                     | t
  3 |                     | f
  4 |                     | t
  5 | {7,8}               | f
  6 |                     | f
  7 |                     | t
  8 |                     | t
  9 |                     | f
 10 |                     | t
 11 |                     | f
 12 |                     | t
 13 |                     | t
 14 |                     | t
 15 | {14}                | f
 16 |                     | t
 17 | {16}                | f
(17 rows)
```

Update the *edge* table using the contraction information

Insert the new edges generated by pgr_contraction.

```
INSERT INTO edge_table(source, target, cost, reverse_cost, contracted_vertices, is_new)
SELECT source, target, cost, -1, contracted_vertices, true
FROM contraction_results
WHERE type = 'e';
INSERT 0 4
```

The modified `edge_table`.

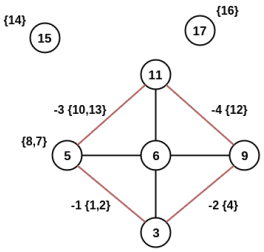
```
SELECT id, source, target, cost, reverse_cost, contracted_vertices, is_new
FROM edge_table
ORDER BY id;
 id | source | target | cost | reverse_cost | contracted_vertices | is_new
----+--------+--------+------+--------------+---------------------+--------
  1 |      1 |      2 |    1 |            1 |                     | f
  2 |      2 |      3 |   -1 |            1 |                     | f
  3 |      3 |      4 |   -1 |            1 |                     | f
  4 |      2 |      5 |    1 |            1 |                     | f
  5 |      3 |      6 |    1 |           -1 |                     | f
  6 |      7 |      8 |    1 |            1 |                     | f
  7 |      8 |      5 |    1 |            1 |                     | f
  8 |      5 |      6 |    1 |            1 |                     | f
  9 |      6 |      9 |    1 |            1 |                     | f
 10 |      5 |     10 |    1 |            1 |                     | f
 11 |      6 |     11 |    1 |           -1 |                     | f
 12 |     10 |     11 |    1 |           -1 |                     | f
 13 |     11 |     12 |    1 |           -1 |                     | f
 14 |     10 |     13 |    1 |            1 |                     | f
 15 |      9 |     12 |    1 |            1 |                     | f
 16 |      4 |      9 |    1 |            1 |                     | f
 17 |     14 |     15 |    1 |            1 |                     | f
 18 |     16 |     17 |    1 |            1 |                     | f
 19 |      3 |      5 |    2 |           -1 | {1,2}               | t
 20 |      3 |      9 |    2 |           -1 | {4}                 | t
 21 |      5 |     11 |    2 |           -1 | {10,13}             | t
 22 |      9 |     11 |    2 |           -1 | {12}                | t
(22 rows)
```

The contracted graph
Vertices that belong to the contracted graph.

```
SELECT id
FROM edge_table_vertices_pgr
WHERE is_contracted = false
ORDER BY id;
 id
----
  3
  5
  6
  9
 11
 15
 17
(7 rows)
```

Edges that belong to the contracted graph.

```
WITH
vertices_in_graph AS (
    SELECT id
    FROM edge_table_vertices_pgr
    WHERE is_contracted = false
)
SELECT id, source, target, cost, reverse_cost, contracted_vertices
FROM edge_table
WHERE source IN (SELECT * FROM vertices_in_graph)
AND target IN (SELECT * FROM vertices_in_graph)
ORDER BY id;
 id | source | target | cost | reverse_cost | contracted_vertices
----+--------+--------+------+--------------+---------------------
  5 |      3 |      6 |    1 |           -1 |
  8 |      5 |      6 |    1 |            1 |
  9 |      6 |      9 |    1 |            1 |
 11 |      6 |     11 |    1 |           -1 |
 19 |      3 |      5 |    2 |           -1 | {1,2}
 20 |      3 |      9 |    2 |           -1 | {4}
 21 |      5 |     11 |    2 |           -1 | {10,13}
 22 |      9 |     11 |    2 |           -1 | {12}
(8 rows)
```



Using the contracted graph

Using the contracted graph with `pgr_dijkstra`

There are three cases when calculating the shortest path between a given source and target in a contracted graph:

- Case 1: Both source and target belong to the contracted graph.
- Case 2: Source and/or target belong to an edge subgraph.
- Case 3: Source and/or target belong to a vertex.

Case 1: Both source and target belong to the contracted graph.

Using the **Edges that belong to the contracted graph.** on lines 10 to 19.

```
1  CREATE OR REPLACE FUNCTION my_dijkstra(
2     departure BIGINT, destination BIGINT,
3     OUT seq INTEGER, OUT path_seq INTEGER,
4     OUT node BIGINT, OUT edge BIGINT,
5     OUT cost FLOAT, OUT agg_cost FLOAT)
6  RETURNS SETOF RECORD AS
7  $BODY$
8  SELECT * FROM pgr_dijkstra(
9     $$
10    WITH
11    vertices_in_graph AS (
12       SELECT id
13       FROM edge_table_vertices_pgr
14       WHERE is_contracted = false
15    )
16    SELECT id, source, target, cost, reverse_cost
17    FROM edge_table
18    WHERE source IN (SELECT * FROM vertices_in_graph)
19    AND target IN (SELECT * FROM vertices_in_graph)
20    $$,
21    departure, destination, false);
22 $BODY$
23 LANGUAGE SQL VOLATILE;
24 CREATE FUNCTION
```

Case 1

When both source and target belong to the contracted graph, a path is found.

```
SELECT * FROM my_dijkstra(3, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    5 |    1 |        0
   2 |        2 |    6 |   11 |    1 |        1
   3 |        3 |   11 |   -1 |    0 |        2
(3 rows)
```

Case 2

When source and/or target belong to an edge subgraph then a path is not found.

In this case, the contracted graph do not have an edge connecting with node `4`.

```
SELECT * FROM my_dijkstra(4, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

Case 3

When source and/or target belong to a vertex then a path is not found.

In this case, the contracted graph do not have an edge connecting with node `7` and of node `4` of the second case.

```
SELECT * FROM my_dijkstra(4, 7);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

Case 2: Source and/or target belong to an edge subgraph.

Refining the above function to include nodes that belong to an edge.

- The vertices that need to be expanded are calculated on lines 10 to 16.
- Adding to the contracted graph that additional section on lines 25 to 27.

```
1  CREATE OR REPLACE FUNCTION my_dijkstra(
2     departure BIGINT, destination BIGINT,
3     OUT seq INTEGER, OUT path_seq INTEGER,
4     OUT node BIGINT, OUT edge BIGINT,
5     OUT cost FLOAT, OUT agg_cost FLOAT)
6  RETURNS SETOF RECORD AS
7  $BODY$
8  SELECT * FROM pgr_dijkstra(
9     $$
10    WITH
11    edges_to_expand AS (
12       SELECT id
13       FROM edge_table
14       WHERE ARRAY[$$ || departure || $$]::BIGINT[] <@ contracted_vertices
15          OR ARRAY[$$ || destination || $$]::BIGINT[] <@ contracted_vertices
16    ),
17
18    vertices_in_graph AS (
19       SELECT id
20       FROM edge_table_vertices_pgr
21       WHERE is_contracted = false
22
23       UNION
24
25       SELECT unnest(contracted_vertices)
26       FROM edge_table
27       WHERE id IN (SELECT id FROM edges_to_expand)
28    )
29
30    SELECT id, source, target, cost, reverse_cost
31    FROM edge_table
32    WHERE source IN (SELECT * FROM vertices_in_graph)
33    AND target IN (SELECT * FROM vertices_in_graph)
34    $$,
35    departure, destination, false);
36 $BODY$
37 LANGUAGE SQL VOLATILE;
38 CREATE FUNCTION
```

Case 1

When both source and target belong to the contracted graph, a path is found.

```
SELECT * FROM my_dijkstra(3, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    5 |    1 |        0
   2 |        2 |    6 |   11 |    1 |        1
   3 |        3 |   11 |   -1 |    0 |        2
(3 rows)
```

Case 2

When source and/or target belong to an edge subgraph, now, a path is found.

The routing graph now has an edge connecting with node4.

```
SELECT * FROM my_dijkstra(4, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    4 |   16 |    1 |        0
   2 |        2 |    9 |   22 |    2 |        1
   3 |        3 |   11 |   -1 |    0 |        3
(3 rows)
```

Case 3

When source and/or target belong to a vertex then a path is not found.

In this case, the contracted graph do not have an edge connecting with node7.

```
SELECT * FROM my_dijkstra(4, 7);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

Case 3: Source and/or target belong to a vertex.

Refining the above function to include nodes that belong to an edge.

- The vertices that need to be expanded are calculated on lines 18 to 23.
- Adding to the contracted graph that additional section on lines 38 to 40.

```
1  CREATE OR REPLACE FUNCTION my_dijkstra(
2     departure BIGINT, destination BIGINT,
3     OUT seq INTEGER, OUT path_seq INTEGER,
4     OUT node BIGINT, OUT edge BIGINT,
5     OUT cost FLOAT, OUT agg_cost FLOAT)
6  RETURNS SETOF RECORD AS
7  $BODY$
8  SELECT * FROM pgr_dijkstra(
9     $$
10    WITH
11    edges_to_expand AS (
12       SELECT id
13       FROM edge_table
14       WHERE ARRAY[$$ || departure || $$]::BIGINT[] <@ contracted_vertices
15          OR ARRAY[$$ || destination || $$]::BIGINT[] <@ contracted_vertices
16    ),
17
18    vertices_to_expand AS (
19       SELECT id
20       FROM edge_table_vertices_pgr
21       WHERE ARRAY[$$ || departure || $$]::BIGINT[] <@ contracted_vertices
22          OR ARRAY[$$ || destination || $$]::BIGINT[] <@ contracted_vertices
23    ),
24
25    vertices_in_graph AS (
26       SELECT id
27       FROM edge_table_vertices_pgr
28       WHERE is_contracted = false
29
30       UNION
31
32       SELECT unnest(contracted_vertices)
33       FROM edge_table
34       WHERE id IN (SELECT id FROM edges_to_expand)
35
36       UNION
37
38       SELECT unnest(contracted_vertices)
39       FROM edge_table_vertices_pgr
40       WHERE id IN (SELECT id FROM vertices_to_expand)
41    )
42
43    SELECT id, source, target, cost, reverse_cost
44    FROM edge_table
45    WHERE source IN (SELECT * FROM vertices_in_graph)
46    AND target IN (SELECT * FROM vertices_in_graph)
47    $$,
48    departure, destination, false);
49 $BODY$
50 LANGUAGE SQL VOLATILE;
51 CREATE FUNCTION
```

Case 1

When both source and target belong to the contracted graph, a path is found.

```
SELECT * FROM my_dijkstra(3, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    5 |    1 |        0
   2 |        2 |    6 |   11 |    1 |        1
   3 |        3 |   11 |   -1 |    0 |        2
(3 rows)
```

Case 2

The code change do not affect this case so when source and/or target belong to an edge subgraph, a path is still found.

```
SELECT * FROM my_dijkstra(4, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    4 |   16 |    1 |        0
   2 |        2 |    9 |   22 |    2 |        1
   3 |        3 |   11 |   -1 |    0 |        3
(3 rows)
```

Case 3

When source and/or target belong to a vertex, now, a path is found.

Now, the routing graph has an edge connecting with node7.

```
SELECT * FROM my_dijkstra(4, 7);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    4 |    3 |    1 |        0
   2 |        2 |    3 |   19 |    2 |        1
   3 |        3 |    5 |    7 |    1 |        3
   4 |        4 |    8 |    6 |    1 |        4
   5 |        5 |    7 |   -1 |    0 |        5
(5 rows)
```

See Also

- **https://www.cs.cmu.edu/afs/cs/academic/class/15210-f12/www/lectures/lecture16.pdf**
- **https://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf**
- The queries use **pgr_contraction** function and the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Dijkstra - Family of functions

- **pgr_dijkstra** - Dijkstra's algorithm for the shortest paths.
- **pgr_dijkstraCost** - Get the aggregate cost of the shortest paths.
- **pgr_dijkstraCostMatrix** - Use pgr_dijkstra to create a costs matrix.
- **pgr_drivingDistance** - Use pgr_dijkstra to calculate catchament information.
- **pgr_KSP** - Use Yen algorithm with pgr_dijkstra to get the K shortest paths.

proposed

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

- **pgr_dijkstraVia - Proposed** - Get a route of a seuence of vertices.

pgr_dijkstra

`pgr_dijkstra` — Returns the shortest path(s) using Dijkstra algorithm. In particular, the Dijkstra algorithm implemented by Boost.Graph.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** functions
- Version 2.2.0
  - New **proposed** functions:
    - pgr_dijkstra(One to Many)
    - pgr_dijkstra(Many to One)
    - pgr_dijkstra(Many to Many)
- Version 2.1.0
  - Signature change on pgr_dijkstra(One to One)
- Version 2.0.0
  - **Official** pgr_dijkstra(One to One)

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956. It is a graph search algorithm that solves the shortest path problem for a graph with non-negative edge path costs, producing a shortest path from a starting vertex (`start_vid`) to an ending vertex `end_vid`). This implementation can be used with a directed graph and an undirected graph.

The main characteristics are:

- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* the non included values *(v, v)* is *0*
  - When the starting vertex and ending vertex are the different and there is no path:
    - The *agg_cost* the non included values *(u, v)* is \infty
- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: O(| start_vids | * (V \log V + E))

Signatures

Summary

```
pgr_dijkstra(edges_sql, start_vid,  end_vid  [, directed])
pgr_dijkstra(edges_sql, start_vid,  end_vids [, directed])
pgr_dijkstra(edges_sql, start_vids, end_vid  [, directed])
pgr_dijkstra(edges_sql, start_vids, end_vids [, directed])
RETURNS SET OF (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)
OR EMPTY SET
```

Using defaults

```
pgr_dijkstra(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Example:
From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)
```

One to One

```
pgr_dijkstra(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid,
BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **undirected** graph

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)
```

One to many

```
pgr_dijkstra(TEXT edges_sql, BIGINT start_vid, ARRAY[ANY_INTEGER] end_vids,
BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{3, 5\} on an **undirected** graph

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    5 |    1 |        2
   4 |        4 |       3 |    3 |   -1 |    0 |        3
   5 |        1 |       5 |    2 |    4 |    1 |        0
   6 |        2 |       5 |    5 |   -1 |    0 |        1
(6 rows)
```

Many to One

```
pgr_dijkstra(TEXT edges_sql, ARRAY[ANY_INTEGER] start_vids, BIGINT end_vid,
    BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertex 5 on a **directed** graph

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   13 |    1 |        0
   4 |        2 |        11 |   12 |   15 |    1 |        1
   5 |        3 |        11 |    9 |    9 |    1 |        2
   6 |        4 |        11 |    6 |    8 |    1 |        3
   7 |        5 |        11 |    5 |   -1 |    0 |        4
(7 rows)
```

Many to Many

```
pgr_dijkstra(TEXT edges_sql, ARRAY[ANY_INTEGER] start_vids, ARRAY[ANY_INTEGER] end_vids,
    BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertices \{3, 5\} on an **undirected** graph

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    2 |    1 |        0
   2 |        2 |         2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   4 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   5 |        1 |        11 |       3 |   11 |   11 |    1 |        0
   6 |        2 |        11 |       3 |    6 |    5 |    1 |        1
   7 |        3 |        11 |       3 |    3 |   -1 |    0 |        2
   8 |        1 |        11 |       5 |   11 |   11 |    1 |        0
   9 |        2 |        11 |       5 |    6 |    8 |    1 |        1
  10 |        3 |        11 |       5 |    5 |   -1 |    0 |        2
(10 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same start_vid to end_vid combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from start_v to node.

Additional Examples

The examples of this section are based on the **Sample Data** network.

The examples include combinations from starting vertices 2 and 11 to ending vertices 3 and 5 in a directed and undirected graph with and with out reverse_cost.

Examples:
For queries marked as directed with cost and reverse_cost columns

The examples in this section use the following **Network for queries marked as directed and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3,5]
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |       5 |    2 |    4 |    1 |        0
   8 |        2 |       5 |    5 |   -1 |    0 |        1
(8 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
```

```
   1 |    1 |  11 |  13 |   1 |    0
   2 |    2 |  12 |  15 |   1 |    1
   3 |    3 |   9 |  16 |   1 |    2
   4 |    4 |   4 |   3 |   1 |    3
   5 |    5 |   3 |  -1 |   0 |    4
(5 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |    1 |  11 |  13 |   1 |    0
   2 |    2 |  12 |  15 |   1 |    1
   3 |    3 |   9 |   9 |   1 |    2
   4 |    4 |   6 |   8 |   1 |    3
   5 |    5 |   5 |  -1 |   0 |    4
(5 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |    1 |       2 |   2 |   4 |    1 |    0
   2 |    2 |       2 |   5 |  -1 |    0 |    1
   3 |    1 |      11 |  11 |  13 |    1 |    0
   4 |    2 |      11 |  12 |  15 |    1 |    1
   5 |    3 |      11 |   9 |   9 |    1 |    2
   6 |    4 |      11 |   6 |   8 |    1 |    3
   7 |    5 |      11 |   5 |  -1 |    0 |    4
(7 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |    1 |       2 |       3 |   2 |   4 |    1 |    0
   2 |    2 |       2 |       3 |   5 |   8 |    1 |    1
   3 |    3 |       2 |       3 |   6 |   9 |    1 |    2
   4 |    4 |       2 |       3 |   9 |  16 |    1 |    3
   5 |    5 |       2 |       3 |   4 |   3 |    1 |    4
   6 |    6 |       2 |       3 |   3 |  -1 |    0 |    5
   7 |    1 |       2 |       5 |   2 |   4 |    1 |    0
   8 |    2 |       2 |       5 |   5 |  -1 |    0 |    1
   9 |    1 |      11 |       3 |  11 |  13 |    1 |    0
  10 |    2 |      11 |       3 |  12 |  15 |    1 |    1
  11 |    3 |      11 |       3 |   9 |  16 |    1 |    2
  12 |    4 |      11 |       3 |   4 |   3 |    1 |    3
  13 |    5 |      11 |       3 |   3 |  -1 |    0 |    4
  14 |    1 |      11 |       5 |  11 |  13 |    1 |    0
  15 |    2 |      11 |       5 |  12 |  15 |    1 |    1
  16 |    3 |      11 |       5 |   9 |   9 |    1 |    2
  17 |    4 |      11 |       5 |   6 |   8 |    1 |    3
  18 |    5 |      11 |       5 |   5 |  -1 |    0 |    4
(18 rows)
```

Examples:

For queries marked as `undirected` with `cost` and `reverse_cost` columns

The examples in this section use the following **Network for queries marked as undirected and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 5,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    5 |    1 |        1
   3 |        3 |    3 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 5,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    8 |    1 |        1
   3 |        3 |    5 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5,
    FALSE
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   12 |    1 |        0
   4 |        2 |        11 |   10 |   10 |    1 |        1
   5 |        3 |        11 |    5 |   -1 |    0 |        2
(5 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    2 |    1 |        0
   2 |        2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |       5 |    2 |    4 |    1 |        0
   4 |        2 |       5 |    5 |   -1 |    0 |        1
(4 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    2 |    1 |        0
   2 |        2 |         2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   4 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   5 |        1 |        11 |       3 |   11 |   11 |    1 |        0
   6 |        2 |        11 |       3 |    6 |    5 |    1 |        1
   7 |        3 |        11 |       3 |    3 |   -1 |    0 |        2
   8 |        1 |        11 |       5 |   11 |   11 |    1 |        0
   9 |        2 |        11 |       5 |    6 |    8 |    1 |        1
  10 |        3 |        11 |       5 |    5 |   -1 |    0 |        2
(10 rows)
```

Examples:

For queries marked as `directed` with `cost` column

The examples in this section use the following **Network for queries marked as directed and only cost column is used**

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    11, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    11, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, ARRAY[3,5]
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       5 |    2 |    4 |    1 |        0
   2 |        2 |       5 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
(2 rows)
```

Examples:

For queries marked as `undirected` with `cost` column

The examples in this section use the following **Network for queries marked as undirected and only cost column is used**

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    5 |    1 |        2
   4 |        4 |    3 |   -1 |    0 |        3
(4 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, 5,
```

```
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    11, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    5 |    1 |        1
   3 |        3 |    3 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    11, 5,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    8 |    1 |        1
   3 |        3 |    5 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2,11], 5,
    FALSE
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   12 |    1 |        0
   4 |        2 |        11 |   10 |   10 |    1 |        1
   5 |        3 |        11 |    5 |   -1 |    0 |        2
(5 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    5 |    1 |        2
   4 |        4 |       3 |    3 |   -1 |    0 |        3
   5 |        1 |       5 |    2 |    4 |    1 |        0
   6 |        2 |       5 |    5 |   -1 |    0 |        1
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    5 |    1 |        2
   4 |        4 |         2 |       3 |    3 |   -1 |    0 |        3
   5 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   6 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   7 |        1 |        11 |       3 |   11 |   11 |    1 |        0
   8 |        2 |        11 |       3 |    6 |    5 |    1 |        1
   9 |        3 |        11 |       3 |    3 |   -1 |    0 |        2
  10 |        1 |        11 |       5 |   11 |   11 |    1 |        0
  11 |        2 |        11 |       5 |    6 |    8 |    1 |        1
  12 |        3 |        11 |       5 |    5 |   -1 |    0 |        2
(12 rows)
```

Equvalences between signatures

Examples:

For queries marked as `directed` with `cost` and `reverse_cost` columns

The examples in this section use the following:

- **Network for queries marked as directed and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    TRUE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2,3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3],
    TRUE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3]
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2], ARRAY[3],
    TRUE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
(6 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2], ARRAY[3]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
(6 rows)
```

Examples:

For queries marked as `undirected` with `cost` and `reverse_cost` columns

The examples in this section use the following:

- **Network for queries marked as undirected and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    2 |    1 |        0
   2 |        2 |       3 |    3 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2], 3,
    FALSE
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    2 |    1 |        0
   2 |        2 |         2 |    3 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2], ARRAY[3],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    2 |    1 |        0
   2 |        2 |         2 |       3 |    3 |   -1 |    0 |        1
(2 rows)
```

See Also

- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_dijkstraCost

`pgr_dijkstraCost`

Using Dijkstra algorithm implemented by Boost.Graph, and extract only the aggregate cost of the shortest path(s) found, for the combination of vertices given.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **Official** function
- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2**

Description

The `pgr_dijkstraCost` algorithm, is a good choice to calculate the sum of the costs of the shortest path for a subset of pairs of nodes of the graph. We make use of the Boost's implementation of dijkstra which runs in $O(V \log V + E)$ time.

The main characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of *(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* int the non included values *(v, v)* is *0*
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values *(u, v)* is \infty
- Let be the case the values returned are stored in a table, so the unique index would be the pair *(start_vid, end_vid)*.
- For undirected graphs, the results are symmetric.
  - The *agg_cost* of *(u, v)* is the same as for *(v, u)*.
- Any duplicated value in the *start_vids* or *end_vids* is ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: O(| start_vids | * (V \log V + E))

Signatures

Summary

```
pgr_dijkstraCost(edges_sql, from_vid,  to_vid  [, directed])
pgr_dijkstraCost(edges_sql, from_vid,  to_vids [, directed])
pgr_dijkstraCost(edges_sql, from_vids, to_vid  [, directed])
pgr_dijkstraCost(edges_sql, from_vids, to_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Using defaults

```
pgr_dijkstraCost(edges_sql, from_vid,  to_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
    'select id, source, target, cost, reverse_cost from edge_table',
    2, 3);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
(1 row)
```

One to One

```
pgr_dijkstraCost(edges_sql, from_vid,  to_vid  [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **undirected** graph

```
SELECT * FROM pgr_dijkstraCost(
    'select id, source, target, cost, reverse_cost from edge_table',
    2, 3, false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        1
(1 row)
```

One to Many

```
pgr_dijkstraCost(edges_sql, from_vid,  to_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{3, 11\} on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
    'select id, source, target, cost, reverse_cost from edge_table',
    2, ARRAY[3, 11]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      11 |        3
(2 rows)
```

Many to One

```
 pgr_dijkstraCost(edges_sql, from_vids, to_vid  [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
    'select id, source, target, cost, reverse_cost from edge_table',
    ARRAY[2, 7], 3);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         7 |       3 |        6
(2 rows)
```

Many to Many

```
pgr_dijkstraCost(edges_sql, from_vids, to_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 7\} to vertices \{3, 11\} on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
    'select id, source, target, cost, reverse_cost from edge_table',
    ARRAY[2, 7], ARRAY[3, 11]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         2 |       3 |        5
         2 |      11 |        3
         7 |       3 |        6
         7 |      11 |        4
(4 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.

Column	Type	Default	Description
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

Additional Examples

Example 1:
Demonstration of repeated values are ignored, and result is sorted.

```
SELECT * FROM pgr_dijkstraCost(
        'select id, source, target, cost, reverse_cost from edge_table',
            ARRAY[5, 3, 4, 3, 3, 4], ARRAY[3, 5, 3, 4]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         3 |       4 |        3
         3 |       5 |        2
         4 |       3 |        1
         4 |       5 |        3
         5 |       3 |        4
         5 |       4 |        3
(6 rows)
```

Example 2:
Making *start_vids* the same as *end_vids*

```
SELECT * FROM pgr_dijkstraCost(
        'select id, source, target, cost, reverse_cost from edge_table',
            ARRAY[5, 3, 4], ARRAY[5, 3, 4]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         3 |       4 |        3
         3 |       5 |        2
         4 |       3 |        1
         4 |       5 |        3
         5 |       3 |        4
         5 |       4 |        3
(6 rows)
```

See Also

- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**
- **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_dijkstraCostMatrix

pgr_dijkstraCostMatrix - Calculates the a cost matrix using pgr_dijktras.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.3.0
  - New **proposed** function
- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3**

Description

Using Dijkstra algorithm, calculate and return a cost matrix.

Signatures

Summary

```
pgr_dijkstraCostMatrix(edges_sql, start_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Using defaults

```
pgr_dijkstraCostMatrix(edges_sql, start_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:
Cost matrix for vertices \{1, 2, 3, 4\} on a **directed** graph

```
SELECT * FROM pgr_dijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5)
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        6
         1 |       4 |        5
         2 |       1 |        1
         2 |       3 |        5
         2 |       4 |        4
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        3
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Complete Signature

```
pgr_dijkstraCostMatrix(edges_sql, start_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:
Symmetric cost matrix for vertices \{1, 2, 3, 4\} on an **undirected** graph

```
SELECT * FROM pgr_dijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
    false
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |       2 |        1
         1 |       3 |        2
         1 |       4 |        3
         2 |       1 |        1
         2 |       3 |        1
         2 |       4 |        2
         3 |       1 |        2
         3 |       2 |        1
         3 |       4 |        1
         4 |       1 |        3
         4 |       2 |        2
         4 |       3 |        1
(12 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of the vertices.
directed	BOOLEAN	(optional). When false the graph is considered as Undirected. Default is true which considers the graph as Directed.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns SET OF (start_vid, end_vid, agg_cost)

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with tsp

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_dijkstraCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 5),
        false
    )
    $$,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    1 |    1 |        0
   2 |    2 |    1 |        1
   3 |    3 |    1 |        2
   4 |    4 |    3 |        3
   5 |    1 |    0 |        6
(5 rows)
```

See Also

- **Dijkstra - Family of functions**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_drivingDistance

`pgr_drivingDistance` - Returns the driving distance from a start node.

Boost Graph Inside

Availability

- Version 2.1.0:
  - Signature change pgr_drivingDistance(single vertex)
  - New **Official** pgr_drivingDistance(multiple vertices)
- Version 2.0.0:
  - **Official** pgr_drivingDistance(single vertex)

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

Using the Dijkstra algorithm, extracts all the nodes that have costs less than or equal to the value `distance`. The edges extracted will conform to the corresponding spanning tree.

Signatures

Summary

```
pgr_drivingDistance(edges_sql, start_vid,  distance [, directed])
pgr_drivingDistance(edges_sql, start_vids, distance [, directed] [, equicost])
RETURNS SET OF (seq, [start_vid,] node, edge, cost, agg_cost)
```

Using defaults

```
pgr_drivingDistance(edges_sql, start_vid, distance)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example:
TBD

Single Vertex

```
pgr_drivingDistance(edges_sql, start_vid, distance [, directed])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example:
TBD

Multiple Vertices

```
pgr_drivingDistance(edges_sql, start_vids, distance, [, directed] [, equicost])
RETURNS SET OF (seq, start_vid, node, edge, cost, agg_cost)
```

Example:
TBD

Parameters

Column	Type	Description
edges_sql	TEXT	SQL query as described above.
start_vid	BIGINT	Identifier of the starting vertex.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of the starting vertices.
distance	FLOAT	Upper limit for the inclusion of the node in the result.
directed	BOOLEAN	(optional). When `false` the graph is considered as Undirected. Default is `true` which considers the graph as Directed.
equicost	BOOLEAN	(optional). When `true` the node will only appear in the closest `start_vid` list. Default is `false` which resembles several calls using the single starting point signatures. Tie brakes are arbitrary.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(seq [, start_v], node, edge, cost, agg_cost)`

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
start_vid	INTEGER	Identifier of the starting vertex.
node	BIGINT	Identifier of the node in the path within the limits from `start_vid`.
edge	BIGINT	Identifier of the edge used to arrive to `node`. `0` when the `node` is the `start_vid`.
cost	FLOAT	Cost to traverse `edge`.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `node`.

Additional Examples

Example:
For queries marked as `directed` with `cost` and `reverse_cost` columns

The examples in this section use the following **Network for queries marked as directed and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        2, 3
      );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |    2 |   -1 |    0 |        0
   2 |    1 |    1 |    1 |        1
   3 |    5 |    4 |    1 |        1
   4 |    6 |    8 |    1 |        2
   5 |    8 |    7 |    1 |        2
   6 |   10 |   10 |    1 |        2
   7 |    7 |    6 |    1 |        3
   8 |    9 |    9 |    1 |        3
   9 |   11 |   12 |    1 |        3
  10 |   13 |   14 |    1 |        3
(10 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        13, 3
      );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   13 |   -1 |    0 |        0
   2 |   10 |   14 |    1 |        1
   3 |    5 |   10 |    1 |        2
   4 |   11 |   12 |    1 |        2
   5 |    2 |    4 |    1 |        3
   6 |    6 |    8 |    1 |        3
   7 |    8 |    7 |    1 |        3
   8 |   12 |   13 |    1 |        3
(8 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        array[2,13], 3
      );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    1 |    1 |    1 |        1
   3 |      2 |    5 |    4 |    1 |        1
   4 |      2 |    6 |    8 |    1 |        2
   5 |      2 |    8 |    7 |    1 |        2
   6 |      2 |   10 |   10 |    1 |        2
   7 |      2 |    7 |    6 |    1 |        3
   8 |      2 |    9 |    9 |    1 |        3
   9 |      2 |   11 |   12 |    1 |        3
  10 |      2 |   13 |   14 |    1 |        3
  11 |     13 |   13 |   -1 |    0 |        0
  12 |     13 |   10 |   14 |    1 |        1
  13 |     13 |    5 |   10 |    1 |        2
  14 |     13 |   11 |   12 |    1 |        2
  15 |     13 |    2 |    4 |    1 |        3
  16 |     13 |    6 |    8 |    1 |        3
  17 |     13 |    8 |    7 |    1 |        3
  18 |     13 |   12 |   13 |    1 |        3
(18 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        array[2,13], 3, equicost:=true
      );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    1 |    1 |    1 |        1
   3 |      2 |    5 |    4 |    1 |        1
   4 |      2 |    6 |    8 |    1 |        2
   5 |      2 |    8 |    7 |    1 |        2
   6 |      2 |    7 |    6 |    1 |        3
   7 |      2 |    9 |    9 |    1 |        3
   8 |     13 |   13 |   -1 |    0 |        0
   9 |     13 |   10 |   14 |    1 |        1
  10 |     13 |   11 |   12 |    1 |        2
  11 |     13 |   12 |   13 |    1 |        3
(11 rows)
```

Example:
For queries marked as `undirected` with `cost` and `reverse_cost` columns

The examples in this section use the following **Network for queries marked as undirected and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        2, 3, false
      );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |    2 |   -1 |    0 |        0
   2 |    1 |    1 |    1 |        1
   3 |    3 |    2 |    1 |        1
   4 |    5 |    4 |    1 |        1
   5 |    4 |    3 |    1 |        2
   6 |    6 |    8 |    1 |        2
   7 |    8 |    7 |    1 |        2
   8 |   10 |   10 |    1 |        2
   9 |    7 |    6 |    1 |        3
  10 |    9 |   16 |    1 |        3
  11 |   11 |   12 |    1 |        3
  12 |   13 |   14 |    1 |        3
(12 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        13, 3, false
      );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   13 |   -1 |    0 |        0
   2 |   10 |   14 |    1 |        1
   3 |    5 |   10 |    1 |        2
   4 |   11 |   12 |    1 |        2
   5 |    2 |    4 |    1 |        3
   6 |    6 |    8 |    1 |        3
   7 |    8 |    7 |    1 |        3
   8 |   12 |   13 |    1 |        3
(8 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        array[2,13], 3, false
      );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    1 |    1 |    1 |        1
   3 |      2 |    3 |    2 |    1 |        1
   4 |      2 |    5 |    4 |    1 |        1
   5 |      2 |    4 |    3 |    1 |        2
   6 |      2 |    6 |    8 |    1 |        2
   7 |      2 |    8 |    7 |    1 |        2
   8 |      2 |   10 |   10 |    1 |        2
   9 |      2 |    7 |    6 |    1 |        3
  10 |      2 |    9 |   16 |    1 |        3
  11 |      2 |   11 |   12 |    1 |        3
  12 |      2 |   13 |   14 |    1 |        3
  13 |     13 |   13 |   -1 |    0 |        0
  14 |     13 |   10 |   14 |    1 |        1
  15 |     13 |    5 |   10 |    1 |        2
  16 |     13 |   11 |   12 |    1 |        2
  17 |     13 |    2 |    4 |    1 |        3
  18 |     13 |    6 |    8 |    1 |        3
  19 |     13 |    8 |    7 |    1 |        3
  20 |     13 |   12 |   13 |    1 |        3
(20 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        array[2,13], 3, false, equicost:=true
      );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    1 |    1 |    1 |        1
   3 |      2 |    3 |    2 |    1 |        1
   4 |      2 |    5 |    4 |    1 |        1
   5 |      2 |    4 |    3 |    1 |        2
   6 |      2 |    6 |    8 |    1 |        2
   7 |      2 |    8 |    7 |    1 |        2
   8 |      2 |    7 |    6 |    1 |        3
   9 |      2 |    9 |   16 |    1 |        3
  10 |     13 |   13 |   -1 |    0 |        0
  11 |     13 |   10 |   14 |    1 |        1
  12 |     13 |   11 |   12 |    1 |        2
  13 |     13 |   12 |   13 |    1 |        3
(13 rows)
```

Example:

For queries marked as `directed` with `cost` column

The examples in this section use the following **Network for queries marked as directed and only cost column is used**

```
SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        2, 3
       );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |    2 |   -1 |    0 |        0
   2 |    5 |    4 |    1 |        1
   3 |    6 |    8 |    1 |        2
   4 |   10 |   10 |    1 |        2
   5 |    9 |    9 |    1 |        3
   6 |   11 |   11 |    1 |        3
   7 |   13 |   14 |    1 |        3
(7 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        13, 3
       );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   13 |   -1 |    0 |        0
(1 row)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        array[2,13], 3
       );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    5 |    4 |    1 |        1
   3 |      2 |    6 |    8 |    1 |        2
   4 |      2 |   10 |   10 |    1 |        2
   5 |      2 |    9 |    9 |    1 |        3
   6 |      2 |   11 |   11 |    1 |        3
   7 |      2 |   13 |   14 |    1 |        3
   8 |     13 |   13 |   -1 |    0 |        0
(8 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        array[2,13], 3, equicost:=true
       );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    5 |    4 |    1 |        1
   3 |      2 |    6 |    8 |    1 |        2
   4 |      2 |   10 |   10 |    1 |        2
   5 |      2 |    9 |    9 |    1 |        3
   6 |      2 |   11 |   11 |    1 |        3
   7 |     13 |   13 |   -1 |    0 |        0
(7 rows)
```

Example:

For queries marked as `undirected` with `cost` column

The examples in this section use the following **Network for queries marked as undirected and only cost column is used**

```
SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        2, 3, false
       );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |    2 |   -1 |    0 |        0
   2 |    1 |    1 |    1 |        1
   3 |    5 |    4 |    1 |        1
   4 |    6 |    8 |    1 |        2
   5 |    8 |    7 |    1 |        2
   6 |   10 |   10 |    1 |        2
   7 |    3 |    5 |    1 |        3
   8 |    7 |    6 |    1 |        3
   9 |    9 |    9 |    1 |        3
  10 |   11 |   12 |    1 |        3
  11 |   13 |   14 |    1 |        3
(11 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        13, 3, false
       );
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   13 |   -1 |    0 |        0
   2 |   10 |   14 |    1 |        1
   3 |    5 |   10 |    1 |        2
   4 |   11 |   12 |    1 |        2
   5 |    2 |    4 |    1 |        3
   6 |    6 |    8 |    1 |        3
   7 |    8 |    7 |    1 |        3
   8 |   12 |   13 |    1 |        3
(8 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        array[2,13], 3, false
       );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    1 |    1 |    1 |        1
   3 |      2 |    5 |    4 |    1 |        1
   4 |      2 |    6 |    8 |    1 |        2
   5 |      2 |    8 |    7 |    1 |        2
   6 |      2 |   10 |   10 |    1 |        2
   7 |      2 |    3 |    5 |    1 |        3
   8 |      2 |    7 |    6 |    1 |        3
   9 |      2 |    9 |    9 |    1 |        3
  10 |      2 |   11 |   12 |    1 |        3
  11 |      2 |   13 |   14 |    1 |        3
  12 |     13 |   13 |   -1 |    0 |        0
  13 |     13 |   10 |   14 |    1 |        1
  14 |     13 |    5 |   10 |    1 |        2
  15 |     13 |   11 |   12 |    1 |        2
  16 |     13 |    2 |    4 |    1 |        3
  17 |     13 |    6 |    8 |    1 |        3
  18 |     13 |    8 |    7 |    1 |        3
  19 |     13 |   12 |   13 |    1 |        3
(19 rows)

SELECT * FROM pgr_drivingDistance(
        'SELECT id, source, target, cost FROM edge_table',
        array[2,13], 3, false, equicost:=true
       );
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |      2 |    2 |   -1 |    0 |        0
   2 |      2 |    1 |    1 |    1 |        1
   3 |      2 |    5 |    4 |    1 |        1
   4 |      2 |    6 |    8 |    1 |        2
   5 |      2 |    8 |    7 |    1 |        2
   6 |      2 |    3 |    5 |    1 |        3
   7 |      2 |    7 |    6 |    1 |        3
   8 |      2 |    9 |    9 |    1 |        3
   9 |     13 |   13 |   -1 |    0 |        0
  10 |     13 |   10 |   14 |    1 |        1
  11 |     13 |   11 |   12 |    1 |        2
  12 |     13 |   12 |   13 |    1 |        3
(12 rows)
```

See Also

- **pgr_alphaShape** - Alpha shape computation
- **Sample Data** network.

Indices and tables

- **Index**

- **Search Page**

pgr_KSP

`pgr_KSP` — Returns the “K” shortest paths.

Boost Graph Inside

Availability

- Version 2.1.0
  - Signature change
    - Old signature no longer supported
- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The K shortest path routing algorithm based on Yen’s algorithm. “K” is the number of shortest paths desired.

Signatures

Summary

```
pgr_KSP(edges_sql, start_vid, end_vid, K [, directed] [, heap_paths])
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Using defaults

```
pgr_ksp(edges_sql, start_vid, end_vid, K);
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Example:
TBD

Complete Signature

```
pgr_KSP(edges_sql, start_vid, end_vid, K [, directed] [, heap_paths])
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Example:
TBD

Parameters

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
start_vid	`BIGINT`	Identifier of the starting vertex.
end_vid	`BIGINT`	Identifier of the ending vertex.
k	`INTEGER`	The desiered number of paths.
directed	`BOOLEAN`	(optional). When `false` the graph is considered as Undirected. Default is `true` which considers the graph as Directed.
heap_paths	`BOOLEAN`	(optional). When `true` returns all the paths stored in the process heap. Default is `false` which only returns `k` paths.

Roughly, if the shortest path has `N` edges, the heap will contain about than `N * k` paths for small value of `k` and `k > 1`.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, path_seq, path_id, node, edge, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	Relative position in the path of node and edge. Has value **1** for the beginning of a path.
path_id	BIGINT	Path identifier. The ordering of the paths For two paths i, j if i < j then agg_cost(i) <= agg_cost(j).
node	BIGINT	Identifier of the node in the path.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the route.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

Additional Examples

Example:
To handle the one flag to choose signatures

The examples in this section use the following **Network for queries marked as directed and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2,
    directed:=true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
(10 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
(10 rows)
```

Example:

For queries marked as `directed` with `cost` and `reverse_cost` columns

The examples in this section use the following **Network for queries marked as directed and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
(10 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2, heap_paths:=true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
  11 |       3 |        1 |    2 |    4 |    1 |        0
  12 |       3 |        2 |    5 |   10 |    1 |        1
  13 |       3 |        3 |   10 |   12 |    1 |        2
  14 |       3 |        4 |   11 |   13 |    1 |        3
  15 |       3 |        5 |   12 |   -1 |    0 |        4
(15 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2, true, true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
  11 |       3 |        1 |    2 |    4 |    1 |        0
  12 |       3 |        2 |    5 |   10 |    1 |        1
  13 |       3 |        3 |   10 |   12 |    1 |        2
  14 |       3 |        4 |   11 |   13 |    1 |        3
  15 |       3 |        5 |   12 |   -1 |    0 |        4
(15 rows)
```

Examples:

For queries marked as `undirected` with `cost` and `reverse_cost` columns

The examples in this section use the following **Network for queries marked as undirected and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2, directed:=false
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    2 |    1 |        0
   2 |       1 |        2 |    3 |    3 |    1 |        1
   3 |       1 |        3 |    4 |   16 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |   10 |    1 |        1
   8 |       2 |        3 |   10 |   12 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
(10 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 12, 2, false, true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    2 |    1 |        0
   2 |       1 |        2 |    3 |    3 |    1 |        1
   3 |       1 |        3 |    4 |   16 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
  11 |       3 |        1 |    2 |    4 |    1 |        0
  12 |       3 |        2 |    5 |   10 |    1 |        1
  13 |       3 |        3 |   10 |   12 |    1 |        2
  14 |       3 |        4 |   11 |   13 |    1 |        3
  15 |       3 |        5 |   12 |   -1 |    0 |        4
  16 |       4 |        1 |    2 |    4 |    1 |        0
  17 |       4 |        2 |    5 |   10 |    1 |        1
  18 |       4 |        3 |   10 |   12 |    1 |        2
  19 |       4 |        4 |   11 |   11 |    1 |        3
  20 |       4 |        5 |    6 |    9 |    1 |        4
  21 |       4 |        6 |    9 |   15 |    1 |        5
  22 |       4 |        7 |   12 |   -1 |    0 |        6
(22 rows)
```

Example:

For queries marked as `directed` with `cost` column

The examples in this section use the following **Network for queries marked as directed and only cost column is used**

```
SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost FROM edge_table',
    2, 3, 2
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost FROM edge_table',
    2, 12, 2
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
(10 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost FROM edge_table',
    2, 12, 2, heap_paths:=true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
  11 |       3 |        1 |    2 |    4 |    1 |        0
  12 |       3 |        2 |    5 |   10 |    1 |        1
  13 |       3 |        3 |   10 |   12 |    1 |        2
  14 |       3 |        4 |   11 |   13 |    1 |        3
  15 |       3 |        5 |   12 |   -1 |    0 |        4
(15 rows)

SELECT * FROM pgr_KSP(
    'SELECT id, source, target, cost FROM edge_table',
    2, 12, 2, true, true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
  11 |       3 |        1 |    2 |    4 |    1 |        0
  12 |       3 |        2 |    5 |   10 |    1 |        1
  13 |       3 |        3 |   10 |   12 |    1 |        2
  14 |       3 |        4 |   11 |   13 |    1 |        3
  15 |       3 |        5 |   12 |   -1 |    0 |        4
(15 rows)
```

Example:

For queries marked as `undirected` with `cost` column

The examples in this section use the following **Network for queries marked as undirected and only cost column is used**

```
SELECT  * FROM pgr_KSP(
    'SELECT id, source, target, cost FROM edge_table',
    2, 12, 2, directed:=false
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
(10 rows)

SELECT  * FROM pgr_KSP(
    'SELECT id, source, target, cost FROM edge_table',
    2, 12, 2, directed:=false, heap_paths:=true
  );
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    2 |    4 |    1 |        0
   2 |       1 |        2 |    5 |    8 |    1 |        1
   3 |       1 |        3 |    6 |    9 |    1 |        2
   4 |       1 |        4 |    9 |   15 |    1 |        3
   5 |       1 |        5 |   12 |   -1 |    0 |        4
   6 |       2 |        1 |    2 |    4 |    1 |        0
   7 |       2 |        2 |    5 |    8 |    1 |        1
   8 |       2 |        3 |    6 |   11 |    1 |        2
   9 |       2 |        4 |   11 |   13 |    1 |        3
  10 |       2 |        5 |   12 |   -1 |    0 |        4
  11 |       3 |        1 |    2 |    4 |    1 |        0
  12 |       3 |        2 |    5 |   10 |    1 |        1
  13 |       3 |        3 |   10 |   12 |    1 |        2
  14 |       3 |        4 |   11 |   13 |    1 |        3
  15 |       3 |        5 |   12 |   -1 |    0 |        4
(15 rows)
```

See Also

- **https://en.wikipedia.org/wiki/K_shortest_path_routing**
- **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_dijkstraVia - Proposed

`pgr_dijkstraVia` — Using dijkstra algorithm, it finds the route that goes through a list of vertices.

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6 2.5 2.4 2.3 2.2**

Description

Given a list of vertices and a graph, this function is equivalent to finding the shortest path between vertex_i and vertex_{i+1} for all i < size_of(vertex_via).

The paths represents the sections of the route.

Signatures

Summary

```
pgr_dijkstraVia(edges_sql, via_vertices [, directed] [, strict] [, U_turn_on_edge])
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
    node, edge, cost, agg_cost, route_agg_cost)
OR EMPTY SET
```

Using default

```
pgr_dijkstraVia(edges_sql, via_vertices)
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
    node, edge, cost, agg_cost, route_agg_cost)
OR EMPTY SET
```

Example:
Find the route that visits the vertices\{ 1, 3, 9\} in that order

```
SELECT * FROM pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 3, 9]
);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost | route_agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------+----------------
   1 |       1 |        1 |         1 |       3 |    1 |    1 |    1 |        0 |              0
   2 |       1 |        2 |         1 |       3 |    2 |    4 |    1 |        1 |              1
   3 |       1 |        3 |         1 |       3 |    5 |    8 |    1 |        2 |              2
   4 |       1 |        4 |         1 |       3 |    6 |    9 |    1 |        3 |              3
   5 |       1 |        5 |         1 |       3 |    9 |   16 |    1 |        4 |              4
   6 |       1 |        6 |         1 |       3 |    4 |    3 |    1 |        5 |              5
   7 |       1 |        7 |         1 |       3 |    3 |   -1 |    0 |        6 |              6
   8 |       2 |        1 |         3 |       9 |    3 |    5 |    1 |        0 |              6
   9 |       2 |        2 |         3 |       9 |    6 |    9 |    1 |        1 |              7
  10 |       2 |        3 |         3 |       9 |    9 |   -2 |    0 |        2 |              8
(10 rows)
```

Complete Signature

```
pgr_dijkstraVia(edges_sql, via_vertices [, directed] [, strict] [, U_turn_on_edge])
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
    node, edge, cost, agg_cost, route_agg_cost)
OR EMPTY SET
```

Example:
Find the route that visits the vertices\{ 1, 3, 9\} in that order on an **undirected** graph, avoiding U-turns when possible

```
SELECT * FROM pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 3, 9], false, strict:=true, U_turn_on_edge:=false
);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost | route_agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------+----------------
   1 |       1 |        1 |         1 |       3 |    1 |    1 |    1 |        0 |              0
   2 |       1 |        2 |         1 |       3 |    2 |    2 |    1 |        1 |              1
   3 |       1 |        3 |         1 |       3 |    3 |   -1 |    0 |        2 |              2
   4 |       2 |        1 |         3 |       9 |    3 |    5 |    1 |        0 |              2
   5 |       2 |        2 |         3 |       9 |    6 |    9 |    1 |        1 |              3
   6 |       2 |        3 |         3 |       9 |    9 |   -2 |    0 |        2 |              4
(6 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		SQL query as described above.

Parameter	Type	Default	Description
via_vertices	ARRAY[ANY-INTEGER]		Array of ordered vertices identifiers that are going to be visited.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as Undirected.
strict	BOOLEAN	false	• When false ignores missing paths returning all paths found • When true if a path is missing stops and returns *EMPTY SET*
U_turn_on_edge	BOOLEAN	true	• When true departing from a visited vertex will not try to avoid using the edge used to reach it. In other words, U turn using the edge with same *id* is allowed. • When false when a departing from a visited vertex tries to avoid using the edge used to reach it. In other words, U turn using the edge with same *id* is used when no other path is found.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (start_vid, end_vid, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
path_pid	BIGINT	Identifier of the path.
path_seq	BIGINT	Sequential value starting from 1 for the path.
start_vid	BIGINT	Identifier of the starting vertex of the path.
end_vid	BIGINT	Identifier of the ending vertex of the path.
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path. -2 for the last node of the route.
cost	FLOAT	Cost to traverse from node using edge to the next node in the route sequence.
agg_cost	FLOAT	Total cost from start_vid to end_vid of the path.
route_agg_cost	FLOAT	Total cost from start_vid of path_pid = 1 to end_vid of the current path_pid .

Additional Examples

Example 1:
Find the route that visits the vertices \{1, 5, 3, 9, 4\} in that order

```
SELECT * FROM pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost | route_agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------+----------------
   1 |       1 |        1 |         1 |       5 |    1 |    1 |    0 |        0 |              0
   2 |       1 |        2 |         1 |       5 |    2 |    4 |    1 |        1 |              1
   3 |       1 |        3 |         1 |       5 |    5 |   -1 |    0 |        2 |              2
   4 |       2 |        1 |         5 |       3 |    5 |    8 |    1 |        0 |              2
   5 |       2 |        2 |         5 |       3 |    6 |    9 |    1 |        1 |              3
   6 |       2 |        3 |         5 |       3 |    9 |   16 |    1 |        2 |              4
   7 |       2 |        4 |         5 |       3 |    4 |    3 |    1 |        3 |              5
   8 |       2 |        5 |         5 |       3 |    3 |   -1 |    0 |        4 |              6
   9 |       3 |        1 |         3 |       9 |    3 |    5 |    1 |        0 |              6
  10 |       3 |        2 |         3 |       9 |    6 |    9 |    1 |        1 |              7
  11 |       3 |        3 |         3 |       9 |    9 |   -1 |    0 |        2 |              8
  12 |       4 |        1 |         9 |       4 |    9 |   16 |    1 |        0 |              8
  13 |       4 |        2 |         9 |       4 |    4 |   -2 |    0 |        1 |              9
(13 rows)
```

Example 2:
What's the aggregate cost of the third path?

```
SELECT agg_cost FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE path_id = 3 AND edge <0;
 agg_cost
----------
        2
(1 row)
```

Example 3:
What's the route's aggregate cost of the route at the end of the third path?

```
SELECT route_agg_cost FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE path_id = 3 AND edge < 0;
 route_agg_cost
----------------
              8
(1 row)
```

Example 4:
How are the nodes visited in the route?

```
SELECT row_number() over () as node_seq, node
FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE edge <> -1 ORDER BY seq;
 node_seq | node
----------+------
        1 |    1
        2 |    2
        3 |    5
        4 |    6
        5 |    9
        6 |    4
        7 |    3
        8 |    6
        9 |    9
       10 |    4
(10 rows)
```

Example 5:
What are the aggregate costs of the route when the visited vertices are reached?

```
SELECT path_id, route_agg_cost FROM  pgr_dijkstraVia(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
   ARRAY[1, 5, 3, 9, 4]
)
WHERE edge < 0;
 path_id | route_agg_cost
---------+----------------
       1 |              2
       2 |              6
       3 |              8
       4 |              9
(4 rows)
```

Example 6:

Show the route's seq and aggregate cost and a status of "passes in front" or "visits" node9

```
SELECT seq, route_agg_cost, node, agg_cost ,
CASE WHEN edge = -1 THEN 'visits'
ELSE 'passes in front'
END as status
FROM  pgr_dijkstraVia(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
   ARRAY[1, 5, 3, 9, 4])
WHERE node = 9 and (agg_cost  <> 0 or seq = 1);
 seq | route_agg_cost | node | agg_cost |     status
-----+----------------+------+----------+-----------------
   6 |              4 |    9 |        2 | passes in front
  11 |              8 |    9 |        2 | visits
(2 rows)

ROLLBACK;
ROLLBACK
```

See Also

- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**
- **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5 2.4 2.3 2.2**

The problem definition (Advanced documentation)

Given the following query:

pgr_dijkstra(sql, start_{vid}, end_{vid}, directed)

where sql = \{(id_i, source_i, target_i, cost_i, reverse_cost_i)\}

and

- source = \bigcup source_i,
- target = \bigcup target_i,

The graphs are defined as follows:

Directed graph

The weighted directed graph, G_d(V,E), is definied by:

- the set of vertices V
  - V = source \cup target \cup {start_{vid}} \cup {end_{vid}}
- the set of edges E
  - E = \begin{cases} \text{ } \{(source_i, target_i, cost_i) \text{ when } cost >=0 \} & \quad \text{if } reverse_cost = \varnothing \\ \text{ } \text{ } & \quad \text{ } \\ \text{ } \{(source_i, target_i, cost_i) \text{ when } cost >=0 \} & \quad \text{ } \\ \cup \{(target_i, source_i, reverse_cost_i) \text{ when } reverse_cost_i>=0 \} & \quad \text{if } reverse_cost \neq \varnothing \\ \end{cases}

Undirected graph

The weighted undirected graph, G_u(V,E), is definied by:

- the set of vertices V
  - $V = source \cup target \cup \{start_v\{vid\}\} \cup \{end_\{vid\}\}$
- the set of edges E
  - $E = \begin{cases} \text{ } \{(source_i, target_i, cost_i) \text{ when } cost >=0 \} & \quad \text{ } \\ \cup \{(target_i, source_i, cost_i) \text{ when } cost >=0 \} & \quad \text{ if } reverse\_cost = \varnothing \\ \text{ } \text{ } & \text{ } \\ \text{ } \{(source_i, target_i, cost_i) \text{ when } cost >=0 \} & \text{ } \\ \cup \{(target_i, source_i, cost_i) \text{ when } cost >=0 \} & \text{ } \\ \cup \{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >=0)\} & \text{ } \\ \cup \{(source_i, target_i, reverse\_cost_i) \text{ when } reverse\_cost_i >=0)\} & \quad \text{ if } reverse\_cost \neq \varnothing \\ \end{cases}$

The problem

Given:

- $start_\{vid\} \in V$ a starting vertex
- $end_\{vid\} \in V$ an ending vertex
- $G(V,E) = \begin{cases} G_d(V,E) & \quad \text{ if6 } directed = true \\ G_u(V,E) & \quad \text{ if5 } directed = false \\ \end{cases}$

Then:

- $\boldsymbol\{\pi\} = \{(path\_seq_i, node_i, edge_i, cost_i, agg\_cost_i)\}$

where:

- $path\_seq_i = i$
- $path\_seq_\{| \pi |\} = | \pi |$
- $node_i \in V$
- $node_1 = start_\{vid\}$
- $node_\{| \pi |\} = end_\{vid\}$
- $\forall i \neq | \pi |, \quad (node_i, node_\{i+1\}, cost_i) \in E$
- $edge_i = \begin{cases} id_\{(node_i, node_\{i+1\},cost_i)\} &\quad \text{when } i \neq | \pi | \\ -1 &\quad \text{when } i = | \pi | \\ \end{cases}$
- $cost_i = cost_\{(node_i, node_\{i+1\})\}$
- $agg\_cost_i = \begin{cases} 0 &\quad \text{when } i = 1 \\ \displaystyle\sum_\{k=1\}^\{i\} cost_\{(node_\{k-1\}, node_k)\} &\quad \text{when } i \neq 1 \\ \end{cases}$

In other words: The algorithm returns a the shortest path between $start_\{vid\}$ and $end_\{vid\}$, if it exists, in terms of a sequence of nodes and of edges,

- $path\_seq$ indicates the relative position in the path of the $node$ or $edge$.
- $cost$ is the cost of the edge to be used to go to the next node.
- agg_cost is the cost from the $start_\{vid\}$ up to the node.

If there is no path, the resulting set is empty.

See Also

Indices and tables

- **Index**
- **Search Page**

Flow - Family of functions

- **pgr_maxFlow** - Only the Max flow calculation using Push and Relabel algorithm.
- **pgr_boykovKolmogorov** - Boykov and Kolmogorov with details of flow on edges.
- **pgr_edmondsKarp** - Edmonds and Karp algorithm with details of flow on edges.
- **pgr_pushRelabel** - Push and relabel algorithm with details of flow on edges.
- Applications
  - **pgr_edgeDisjointPaths** - Calculates edge disjoint paths between two groups of vertices.
  - **pgr_maxCardinalityMatch** - Calculates a maximum cardinality matching in a graph.

Experimental

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

- **pgr_maxFlowMinCost - Experimental** - Details of flow and cost on edges.
- **pgr_maxFlowMinCost_Cost - Experimental** - Only the Min Cost calculation.

pgr_maxFlow

`pgr_maxFlow` — Calculates the maximum flow in a directed graph from the source(s) to the targets(s) using the Push Relabel algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **Proposed** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4**

Description

The main characteristics are:

- The graph is **directed**.
- Calculates the maximum flow from the *source(s)* to the *target(s)*.
  - When the maximum flow is **0** then there is no flow and **0** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Uses the **pgr_pushRelabel** algorithm.
- Running time: O(V ^ 3)

Signatures

Summary

```
pgr_maxFlow(Edges SQL, source,  target)
pgr_maxFlow(Edges SQL, sources,  target)
pgr_maxFlow(Edges SQL, source,  targets)
pgr_maxFlow(Edges SQL, sources,  targets)
RETURNS BIGINT
```

One to One

```
pgr_maxFlow(Edges SQL, source,  target)
RETURNS BIGINT
```

Example:

From vertex 6 to vertex 11

```
SELECT * FROM pgr_maxFlow(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, 11
);
 pgr_maxflow
-------------
         230
(1 row)
```

One to Many

```
pgr_maxFlow(Edges SQL, source,  targets)
RETURNS BIGINT
```

Example:

From vertex 6 to vertices \{11, 1, 13\}

```
SELECT * FROM pgr_maxFlow(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, ARRAY[11, 1, 13]
);
 pgr_maxflow
-------------
         340
(1 row)
```

Many to One

```
pgr_maxFlow(Edges SQL, sources,  target)
RETURNS BIGINT
```

Example:

From vertices \{6, 8, 12\} to vertex 11

```
SELECT * FROM pgr_maxFlow(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , ARRAY[6, 8, 12], 11
);
 pgr_maxflow
-------------
         230
(1 row)
```

Many to Many

```
pgr_maxFlow(Edges SQL, sources,  targets)
RETURNS BIGINT
```

Example:

From vertices \{6, 8, 12\} to vertices \{1, 3, 11\}

```
SELECT * FROM pgr_maxFlow(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 pgr_maxflow
-------------
         360
(1 row)
```

Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Return Columns

Type	Description
BIGINT	Maximum flow possible from the source(s) to the target(s)

See Also

- **Flow - Family of functions**
- **https://www.boost.org/libs/graph/doc/push_relabel_max_flow.html**
- **https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_boykovKolmogorov

pgr_boykovKolmogorov — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Boykov Kolmogorov algorithm.

Boost Graph Inside

Availability:

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from `pgr_maxFlowBoykovKolmogorov`
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3**

Description

The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time: Polynomial

Signatures

Summary

```
pgr_boykovKolmogorov(Edges SQL, source,  target)
pgr_boykovKolmogorov(Edges SQL, sources, target)
pgr_boykovKolmogorov(Edges SQL, source,  targets)
pgr_boykovKolmogorov(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One

```
pgr_boykovKolmogorov(Edges SQL, source,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertex 6 to vertex 11

```
SELECT * FROM pgr_boykovKolmogorov(
    'SELECT id,
          source,
          target,
          capacity,
          reverse_capacity
    FROM edge_table'
    , 6, 11
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         5 |      10 |  100 |                30
   2 |    8 |         6 |       5 |  100 |                30
   3 |   11 |         6 |      11 |  130 |                 0
   4 |   12 |        10 |      11 |  100 |                 0
(4 rows)
```

One to Many

```
pgr_boykovKolmogorov(Edges SQL, source,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertex 6 to vertices \{1, 3, 11\}

```
SELECT * FROM pgr_boykovKolmogorov(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, ARRAY[1, 3, 11]
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         2 |       1 |   50 |                80
   2 |    3 |         4 |       3 |   80 |                50
   3 |    4 |         5 |       2 |   50 |                 0
   4 |   10 |         5 |      10 |   80 |                50
   5 |    8 |         6 |       5 |  130 |                 0
   6 |    9 |         6 |       9 |   80 |                50
   7 |   11 |         6 |      11 |  130 |                 0
   8 |   16 |         9 |       4 |   80 |                 0
   9 |   12 |        10 |      11 |   80 |                20
(9 rows)
```

Many to One

```
pgr_boykovKolmogorov(Edges SQL, sources,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertices \{6, 8, 12\} to vertex 11

```
SELECT * FROM pgr_boykovKolmogorov(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , ARRAY[6, 8, 12], 11
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         5 |      10 |  100 |                30
   2 |    8 |         6 |       5 |  100 |                30
   3 |   11 |         6 |      11 |  130 |                 0
   4 |   12 |        10 |      11 |  100 |                 0
(4 rows)
```

Many to Many

```
pgr_boykovKolmogorov(Edges SQL, sources,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertices \{6, 8, 12\} to vertices \{1, 3, 11\}

```
SELECT * FROM pgr_boykovKolmogorov(
   'SELECT id,
         source,
         target,
         capacity,
         reverse_capacity
    FROM edge_table'
    , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         2 |       1 |   50 |                80
   2 |    3 |         4 |       3 |   80 |                50
   3 |    4 |         5 |       2 |   50 |                 0
   4 |   10 |         5 |      10 |  100 |                30
   5 |    8 |         6 |       5 |  130 |                 0
   6 |    9 |         6 |       9 |   80 |                50
   7 |   11 |         6 |      11 |  130 |                 0
   8 |    7 |         8 |       5 |   20 |                30
   9 |   16 |         9 |       4 |   80 |                 0
  10 |   12 |        10 |      11 |  100 |                 0
(10 rows)
```

Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
start_vid	BIGINT	Identifier of the first end point vertex of the edge.
end_vid	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (start_vid, end_vid).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (start_vid, end_vid).

See Also

- **Flow - Family of functions**, **pgr_pushRelabel**, **pgr_edmondsKarp**
- **https://www.boost.org/libs/graph/doc/boykov_kolmogorov_max_flow.html**

Indices and tables

- **Index**
- **Search Page**

`pgr_edmondsKarp` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Push Relabel Algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from `pgr_maxFlowEdmondsKarp`
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3**

Description

The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time: O(V * E ^ 2)

Signatures

Summary

```
pgr_edmondsKarp(Edges SQL, source,  target)
pgr_edmondsKarp(Edges SQL, sources, target)
pgr_edmondsKarp(Edges SQL, source,  targets)
pgr_edmondsKarp(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One

```
pgr_edmondsKarp(Edges SQL, source,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertex 6 to vertex 11

```
SELECT * FROM pgr_edmondsKarp(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, 11
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         5 |      10 |  100 |                30
   2 |    8 |         6 |       5 |  100 |                30
   3 |   11 |         6 |      11 |  130 |                 0
   4 |   12 |        10 |      11 |  100 |                 0
(4 rows)
```

One to Many

```
pgr_edmondsKarp(Edges SQL, source,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertex 6 to vertices \{1, 3, 11\}

```
SELECT * FROM pgr_edmondsKarp(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, ARRAY[1, 3, 11]
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         2 |       1 |   50 |                80
   2 |    3 |         4 |       3 |   80 |                50
   3 |    4 |         5 |       2 |   50 |                 0
   4 |   10 |         5 |      10 |   80 |                50
   5 |    8 |         6 |       5 |  130 |                 0
   6 |    9 |         6 |       9 |   80 |                50
   7 |   11 |         6 |      11 |  130 |                 0
   8 |   16 |         9 |       4 |   80 |                 0
   9 |   12 |        10 |      11 |   80 |                20
(9 rows)
```

Many to One

```
pgr_edmondsKarp(Edges SQL, sources,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertices \{6, 8, 12\} to vertex 11

```
SELECT * FROM pgr_edmondsKarp(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , ARRAY[6, 8, 12], 11
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         5 |      10 |  100 |                30
   2 |    8 |         6 |       5 |  100 |                30
   3 |   11 |         6 |      11 |  130 |                 0
   4 |   12 |        10 |      11 |  100 |                 0
(4 rows)
```

Many to Many

```
pgr_edmondsKarp(Edges SQL, sources,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertices \{6, 8, 12\} to vertices \{1, 3, 11\}

```
SELECT * FROM pgr_edmondsKarp(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
     FROM edge_table'
   , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         2 |       1 |   50 |                80
   2 |    3 |         4 |       3 |   80 |                50
   3 |    4 |         5 |       2 |   50 |                 0
   4 |   10 |         5 |      10 |  100 |                30
   5 |    8 |         6 |       5 |  130 |                 0
   6 |    9 |         6 |       9 |   80 |                50
   7 |   11 |         6 |      11 |  130 |                 0
   8 |    7 |         8 |       5 |   20 |                30
   9 |   16 |         9 |       4 |   80 |                 0
  10 |   12 |        10 |      11 |  100 |                 0
(10 rows)
```

Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
start_vid	BIGINT	Identifier of the first end point vertex of the edge.
end_vid	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (start_vid, end_vid).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (start_vid, end_vid).

See Also

- **Flow - Family of functions**, **pgr_boykovKolmogorov**, **pgr_pushRelabel**
- **https://www.boost.org/libs/graph/doc/edmonds_karp_max_flow.html**
- **https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_pushRelabel

`pgr_pushRelabel` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Push Relabel Algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from `pgr_maxFlowPushRelabel`
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3**

Description

The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time: O(V ^ 3)

Signatures

Summary

```
pgr_pushRelabel(Edges SQL, source,  target)
pgr_pushRelabel(Edges SQL, sources, target)
pgr_pushRelabel(Edges SQL, source,  targets)
pgr_pushRelabel(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One

```
pgr_pushRelabel(Edges SQL, source,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertex 6 to vertex 11

```
SELECT * FROM pgr_pushRelabel(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, 11
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         5 |      10 |  100 |                30
   2 |    8 |         6 |       5 |  100 |                30
   3 |   11 |         6 |      11 |  130 |                 0
   4 |   12 |        10 |      11 |  100 |                 0
(4 rows)
```

One to Many

Calculates the flow on the graph edges that maximizes the flow from the*source* to all of the*targets*.

```
pgr_pushRelabel(Edges SQL, source,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertex 6 to vertices \{11, 1, 13\}

```
SELECT * FROM pgr_pushRelabel(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, ARRAY[11, 1, 13]
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         2 |       1 |  130 |                 0
   2 |    2 |         3 |       2 |   80 |                20
   3 |    3 |         4 |       3 |   80 |                50
   4 |    4 |         5 |       2 |   50 |                 0
   5 |    7 |         5 |       8 |   50 |                80
   6 |   10 |         5 |      10 |   80 |                50
   7 |    8 |         6 |       5 |  130 |                 0
   8 |    9 |         6 |       9 |   80 |                50
   9 |   11 |         6 |      11 |  130 |                 0
  10 |    6 |         7 |       8 |   50 |                 0
  11 |    6 |         8 |       7 |   50 |                50
  12 |    7 |         8 |       5 |   50 |                 0
  13 |   16 |         9 |       4 |   80 |                 0
  14 |   12 |        10 |      11 |   80 |                20
(14 rows)
```

Many to One

```
pgr_pushRelabel(Edges SQL, sources,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertices \{6, 8, 12\} to vertex 11

```
SELECT * FROM pgr_pushRelabel(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , ARRAY[6, 8, 12], 11
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         5 |      10 |  100 |                30
   2 |    8 |         6 |       5 |  100 |                30
   3 |   11 |         6 |      11 |  130 |                 0
   4 |   12 |        10 |      11 |  100 |                 0
(4 rows)
```

Many to Many

```
pgr_pushRelabel(Edges SQL, sources,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:

From vertices \{6, 8, 12\} to vertices \{1, 3, 11\}

```
SELECT * FROM pgr_pushRelabel(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
     FROM edge_table'
    , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         2 |       1 |   50 |                80
   2 |    3 |         4 |       3 |   80 |                50
   3 |    4 |         5 |       2 |   50 |                 0
   4 |   10 |         5 |      10 |  100 |                30
   5 |    8 |         6 |       5 |  130 |                 0
   6 |    9 |         6 |       9 |   30 |               100
   7 |   11 |         6 |      11 |  130 |                 0
   8 |    7 |         8 |       5 |   20 |                30
   9 |   16 |         9 |       4 |   80 |                 0
  10 |   12 |        10 |      11 |  100 |                 0
  11 |   15 |        12 |       9 |   50 |                 0
(11 rows)
```

Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

Edges SQL:

an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

Result Columns

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
start_vid	BIGINT	Identifier of the first end point vertex of the edge.
end_vid	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (start_vid, end_vid).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (start_vid, end_vid).

See Also

- **Flow - Family of functions**, **pgr_boykovKolmogorov**, **pgr_edmondsKarp**
- **https://www.boost.org/libs/graph/doc/push_relabel_max_flow.html**
- **https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_edgeDisjointPaths

`pgr_edgeDisjointPaths` — Calculates edge disjoint paths between two groups of vertices.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions: 2.6 2.5 2.4 2.3**

Description

Calculates the edge disjoint paths between two groups of vertices. Utilizes underlying maximum flow algorithms to calculate the paths.

The main characterics are:

- Calculates the edge disjoint paths between any two groups of vertices.
- Returns EMPTY SET when source and destination are the same, or cannot be reached.
- The graph can be directed or undirected.
- One to many, many to one, many to many versions are also supported.
- Uses **pgr_boykovKolmogorov** to calculate the paths.

Signatures

Summary

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid)
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid [, directed])
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vids [, directed])
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vid [, directed])
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vids [, directed])

RETURNS SET OF (seq, path_id, path_seq, [start_vid,] [end_vid,] node, edge, cost, agg_cost)
OR EMPTY SET
```

Using defaults

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid)
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 3 to vertex 5 on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    3, 5
);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    3 |    2 |    1 |        0
   2 |       1 |        2 |    2 |    4 |    1 |        1
   3 |       1 |        3 |    5 |   -1 |    0 |        2
   4 |       2 |        1 |    3 |    5 |    1 |        0
   5 |       2 |        2 |    6 |    8 |    1 |        1
   6 |       2 |        3 |    5 |   -1 |    0 |        2
(6 rows)
```

One to One

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid, directed)
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 3 to vertex 5 on an **undirected** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    3, 5,
    directed := false
);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    3 |    2 |    1 |        0
   2 |       1 |        2 |    2 |    4 |    1 |        1
   3 |       1 |        3 |    5 |   -1 |    0 |        2
   4 |       2 |        1 |    3 |    3 |   -1 |        0
   5 |       2 |        2 |    4 |   16 |    1 |       -1
   6 |       2 |        3 |    9 |    9 |    1 |        0
   7 |       2 |        4 |    6 |    8 |    1 |        1
   8 |       2 |        5 |    5 |   -1 |    0 |        2
   9 |       3 |        1 |    3 |    5 |    1 |        0
  10 |       3 |        2 |    6 |   11 |    1 |        1
  11 |       3 |        3 |   11 |   12 |   -1 |        2
  12 |       3 |        4 |   10 |   10 |    1 |        1
  13 |       3 |        5 |    5 |   -1 |    0 |        2
(13 rows)
```

One to Many

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vids, directed)
RETURNS SET OF (seq, path_id, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 3 to vertices \{4, 5, 10\} on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    3, ARRAY[4, 5, 10]
);
 seq | path_id | path_seq | end_vid | node | edge | cost | agg_cost
-----+---------+----------+---------+------+------+------+----------
   1 |       1 |        1 |       4 |    3 |    5 |    1 |        0
   2 |       1 |        2 |       4 |    6 |    9 |    1 |        1
   3 |       1 |        3 |       4 |    9 |   16 |    1 |        2
   4 |       1 |        4 |       4 |    4 |   -1 |    0 |        3
   5 |       2 |        1 |       5 |    3 |    2 |    1 |        0
   6 |       2 |        2 |       5 |    2 |    4 |    1 |        1
   7 |       2 |        3 |       5 |    5 |   -1 |    0 |        2
   8 |       3 |        1 |       5 |    3 |    5 |    1 |        0
   9 |       3 |        2 |       5 |    6 |    8 |    1 |        1
  10 |       3 |        3 |       5 |    5 |   -1 |    0 |        2
  11 |       4 |        1 |      10 |    3 |    2 |    1 |        0
  12 |       4 |        2 |      10 |    2 |    4 |    1 |        1
  13 |       4 |        3 |      10 |    5 |   10 |    1 |        2
  14 |       4 |        4 |      10 |   10 |   -1 |    0 |        3
(14 rows)
```

Many to One

```
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vid, directed)
RETURNS SET OF (seq, path_id, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{3, 6\} to vertex 5 on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[3, 6], 5
);
 seq | path_id | path_seq | start_vid | node | edge | cost | agg_cost
-----+---------+----------+-----------+------+------+------+----------
   1 |       1 |        1 |         0 |    3 |    2 |    1 |        0
   2 |       1 |        2 |         0 |    2 |    4 |    1 |        1
   3 |       1 |        3 |         0 |    5 |   -1 |    0 |        2
   4 |       2 |        1 |         1 |    3 |    5 |    1 |        0
   5 |       2 |        2 |         1 |    6 |    8 |    1 |        1
   6 |       2 |        3 |         1 |    5 |   -1 |    0 |        2
   7 |       3 |        1 |         2 |    6 |    8 |    1 |        0
   8 |       3 |        2 |         2 |    5 |   -1 |    0 |        1
   9 |       4 |        1 |         3 |    6 |    9 |    1 |        0
  10 |       4 |        2 |         3 |    9 |   16 |    1 |        1
  11 |       4 |        3 |         3 |    4 |    3 |    1 |        2
  12 |       4 |        4 |         3 |    3 |    2 |    1 |        3
  13 |       4 |        5 |         3 |    2 |    4 |    1 |        4
  14 |       4 |        6 |         3 |    5 |   -1 |    0 |        5
(14 rows)
```

Many to Many

```
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vids, directed)
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{3, 6\} to vertices \{4, 5, 10\} on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[3, 6], ARRAY[4, 5, 10]
);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------
   1 |       1 |        1 |         0 |       4 |    3 |    5 |    1 |        0
   2 |       1 |        2 |         0 |       4 |    6 |    9 |    1 |        1
   3 |       1 |        3 |         0 |       4 |    9 |   16 |    1 |        2
   4 |       1 |        4 |         0 |       4 |    4 |   -1 |    0 |        3
   5 |       2 |        1 |         1 |       5 |    3 |    2 |    1 |        0
   6 |       2 |        2 |         1 |       5 |    2 |    4 |    1 |        1
   7 |       2 |        3 |         1 |       5 |    5 |   -1 |    0 |        2
   8 |       3 |        1 |         2 |       5 |    3 |    5 |    1 |        0
   9 |       3 |        2 |         2 |       5 |    6 |    8 |    1 |        1
  10 |       3 |        3 |         2 |       5 |    5 |   -1 |    0 |        2
  11 |       4 |        1 |         3 |      10 |    3 |    2 |    1 |        0
  12 |       4 |        2 |         3 |      10 |    2 |    4 |    1 |        1
  13 |       4 |        3 |         3 |      10 |    5 |   10 |    1 |        2
  14 |       4 |        4 |         3 |      10 |   10 |   -1 |    0 |        3
  15 |       5 |        1 |         4 |       4 |    6 |    9 |    1 |        0
  16 |       5 |        2 |         4 |       4 |    9 |   16 |    1 |        1
  17 |       5 |        3 |         4 |       4 |    4 |   -1 |    0 |        2
  18 |       6 |        1 |         5 |       5 |    6 |    8 |    1 |        0
  19 |       6 |        2 |         5 |       5 |    5 |   -1 |    0 |        1
  20 |       7 |        1 |         6 |       5 |    6 |    9 |    1 |        0
  21 |       7 |        2 |         6 |       5 |    9 |   16 |    1 |        1
  22 |       7 |        3 |         6 |       5 |    4 |    3 |    1 |        2
  23 |       7 |        4 |         6 |       5 |    3 |    2 |    1 |        3
  24 |       7 |        5 |         6 |       5 |    2 |    4 |    1 |        4
  25 |       7 |        6 |         6 |       5 |    5 |   -1 |    0 |        5
  26 |       8 |        1 |         7 |      10 |    6 |    8 |    1 |        0
  27 |       8 |        2 |         7 |      10 |    5 |   10 |    1 |        1
  28 |       8 |        3 |         7 |      10 |   10 |   -1 |    0 |        2
(28 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same start_vid to end_vid combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from start_v to node.

See Also

- **Flow - Family of functions**

Indices and tables

- **Index**
- **Search Page**

pgr_maxCardinalityMatch

pgr_maxCardinalityMatch — Calculates a maximum cardinality matching in a graph.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from pgr_maximumCardinalityMatching
  - **Proposed** function

- Version 2.3.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6 2.5 2.4 2.3**

Description

The main characteristics are:

- A matching or independent edge set in a graph is a set of edges without common vertices.
- A maximum matching is a matching that contains the largest possible number of edges.
  - There may be many maximum matchings.
  - Calculates **one** possible maximum cardinality matching in a graph.
- The graph can be **directed** or **undirected**.
- Running time: O(E*V * \alpha(E,V))
  - \alpha(E,V) is the inverse of the **Ackermann function**.

Signatures

```
pgr_maxCardinalityMatch(Edges SQL [, directed])

RETURNS SET OF (seq, edge_id, source, target)
OR EMPTY SET
```

Example:
For an **undirected** graph

```
SELECT * FROM pgr_maxCardinalityMatch(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    directed := false
);
 seq | edge | source | target
-----+------+--------+--------
   1 |    1 |      1 |      2
   2 |    3 |      3 |      4
   3 |    9 |      6 |      9
   4 |    6 |      7 |      8
   5 |   14 |     10 |     13
   6 |   13 |     11 |     12
   7 |   17 |     14 |     15
   8 |   18 |     16 |     17
(8 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		SQL query as described above.
directed	BOOLEAN	true	Determines the type of the graph. - When `true` Graph is considered *Directed* - When `false` the graph is considered as *Undirected*.

Inner query

Edges SQL:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
id	ANY-INTEGER	Identifier of the edge.
source	ANY-INTEGER	Identifier of the first end point vertex of the edge.
target	ANY-INTEGER	Identifier of the second end point vertex of the edge.
going	ANY-NUMERIC	A positive value represents the existence of the edge (`source`, `target`).
coming	ANY-NUMERIC	A positive value represents the existence of the edge (`target`, `source`).

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERIC:
SMALLINT, INTEGER, BIGINT, REAL FLOAT

Result Columns

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query.
source	BIGINT	Identifier of the first end point of the edge.
target	BIGINT	Identifier of the second end point of the edge.

See Also

- **Flow - Family of functions**
- **https://www.boost.org/libs/graph/doc/maximum_matching.html**
- **https://en.wikipedia.org/wiki/Matching_%28graph_theory%29**
- **https://en.wikipedia.org/wiki/Ackermann_function**

Indices and tables

- **Index**
- **Search Page**

pgr_maxFlowMinCost - Experimental

pgr_maxFlowMinCost — Calculates the flow on the graph edges that maximizes the flow and minimizes the cost from the sources to the targets.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
- New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- **TODO** check which statement is true:
  - The cost value of all input edges must be nonnegative.
  - Process is done when the cost value of all input edges is nonnegative.
  - Process is done on edges with nonnegative cost.
- Running time: O(U * (E + V * logV))
  - where U is the value of the max flow.
  - U is upper bound on number of iterations. In many real world cases number of iterations is much smaller than U.

Signatures

Summary

```
pgr_maxFlowMinCost(Edges SQL, source,  target)
pgr_maxFlowMinCost(Edges SQL, sources, target)
pgr_maxFlowMinCost(Edges SQL, source,  targets)
pgr_maxFlowMinCost(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

One to One

```
pgr_maxFlowMinCost(Edges SQL, source, target)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 2 to vertex 3

```
SELECT * FROM pgr_MaxFlowMinCost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    4 |      2 |      5 |   80 |                20 |   80 |       80
   2 |    3 |      4 |      3 |   80 |                50 |   80 |      160
   3 |    8 |      5 |      6 |   80 |                20 |   80 |      240
   4 |    9 |      6 |      9 |   80 |                50 |   80 |      320
   5 |   16 |      9 |      4 |   80 |                 0 |   80 |      400
(5 rows)
```

One to Many

```
pgr_maxFlowMinCost(Edges SQL, source, targets)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

Example:
From vertex 13 to vertices \{7, 1, 4\}

```
SELECT * FROM pgr_MaxFlowMinCost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    13, ARRAY[7, 1, 4]
);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    1 |      2 |      1 |   50 |                80 |   50 |       50
   2 |    4 |      5 |      2 |   50 |                 0 |   50 |      100
   3 |   16 |      9 |      4 |   50 |                30 |   50 |      150
   4 |   10 |     10 |      5 |   50 |                 0 |   50 |      200
   5 |   12 |     10 |     11 |   50 |                50 |   50 |      250
   6 |   13 |     11 |     12 |   50 |                50 |   50 |      300
   7 |   15 |     12 |      9 |   50 |                 0 |   50 |      350
   8 |   14 |     13 |     10 |  100 |                30 |  100 |      450
(8 rows)
```

Many to One

```
pgr_maxFlowMinCost(Edges SQL, sources, target)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{1, 7, 14\} to vertex 12

```
SELECT * FROM pgr_MaxFlowMinCost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    ARRAY[1, 7, 14], 12
);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    1 |      1 |      2 |   80 |                 0 |   80 |       80
   2 |    4 |      2 |      5 |   80 |                20 |   80 |      160
   3 |    8 |      5 |      6 |  100 |                 0 |  100 |      260
   4 |   10 |      5 |     10 |   30 |               100 |   30 |      290
   5 |    9 |      6 |      9 |   50 |                80 |   50 |      340
   6 |   11 |      6 |     11 |   50 |                80 |   50 |      390
   7 |    6 |      7 |      8 |   50 |                 0 |   50 |      440
   8 |    7 |      8 |      5 |   50 |                 0 |   50 |      490
   9 |   15 |      9 |     12 |   50 |                30 |   50 |      540
  10 |   12 |     10 |     11 |   30 |                70 |   30 |      570
  11 |   13 |     11 |     12 |   80 |                20 |   80 |      650
(11 rows)
```

Many to Many

```
pgr_maxFlowMinCost(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{7, 13\} to vertices \{3, 9\}

```
SELECT * FROM pgr_MaxFlowMinCost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    ARRAY[7, 13], ARRAY[3, 9]
);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    8 |      5 |      6 |  100 |                 0 |  100 |      100
   2 |    9 |      6 |      9 |  100 |                30 |  100 |      200
   3 |    6 |      7 |      8 |   50 |                 0 |   50 |      250
   4 |    7 |      8 |      5 |   50 |                 0 |   50 |      300
   5 |   10 |     10 |      5 |   50 |                 0 |   50 |      350
   6 |   12 |     10 |     11 |   50 |                50 |   50 |      400
   7 |   13 |     11 |     12 |   50 |                50 |   50 |      450
   8 |   15 |     12 |      9 |   50 |                 0 |   50 |      500
   9 |   14 |     13 |     10 |  100 |                30 |  100 |      600
(9 rows)
```

Parameters

Column	Type	Default	Description

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Capacity of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Capacity of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* if it exists.
reverse_cost	ANY-NUMERICAL	0	Weight of the edge *(target, source)* if it exists.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
source	BIGINT	Identifier of the first end point vertex of the edge.
target	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (source, target).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (source, target).
cost	FLOAT	The cost of sending this flow through the edge in the direction (source, target).
agg_cost	FLOAT	The aggregate cost.

See Also

- **Flow - Family of functions**
- **https://www.boost.org/libs/graph/doc/successive_shortest_path_nonnegative_weights.html**

Indices and tables

- **Index**
- **Search Page**

pgr_maxFlowMinCost_Cost - Experimental

pgr_maxFlowMinCost_Cost — Calculates the minmum cost maximum flow in a directed graph from the source(s) to the targets(s).

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

The main characteristics are:

- The graph is **directed**.
- **The cost value of all input edges must be nonnegative.**
- When the maximum flow is 0 then there is no flow and **0** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Uses the **pgr_maxFlowMinCost** algorithm.
- Running time: O(U * (E + V * logV)), where U is the value of the max flow.U is upper bound on number of iteration. In many real world cases number of iterations is much smaller than U.

Signatures

Summary

```
pgr_maxFlowMinCost_Cost(Edges SQL, source, target)
pgr_maxFlowMinCost_Cost(Edges SQL, sources, target)
pgr_maxFlowMinCost_Cost(Edges SQL, source, targets)
pgr_maxFlowMinCost_Cost(Edges SQL, sources, targets)
RETURNS FLOAT
```

One to One

```
pgr_maxFlowMinCost_Cost(Edges SQL, source, target)
RETURNS FLOAT
```

Example:
From vertex 2 to vertex 3

```
SELECT * FROM pgr_MaxFlowMinCost_Cost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    2, 3
);
 pgr_maxflowmincost_cost
-------------------------
                     400
(1 row)
```

One to Many

```
pgr_maxFlowMinCost_Cost(Edges SQL, source, targets)
RETURNS FLOAT
```

Example:

From vertex 13 to vertices \{7, 1, 4\}

```
SELECT * FROM pgr_MaxFlowMinCost_Cost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    13, ARRAY[7, 1, 4]
);
 pgr_maxflowmincost_cost
-------------------------
                     450
(1 row)
```

Many to One

```
pgr_maxFlowMinCost_Cost(Edges SQL, sources, target)
RETURNS FLOAT
```

Example:

From vertices \{1, 7, 14\} to vertex 12

```
SELECT * FROM pgr_MaxFlowMinCost_Cost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    ARRAY[1, 7, 14], 12
);
 pgr_maxflowmincost_cost
-------------------------
                     650
(1 row)
```

Many to Many

```
pgr_maxFlowMinCost_Cost(Edges SQL, sources, targets)
RETURNS FLOAT
```

Example:

From vertices \{7, 13\} to vertices \{3, 9\}

```
SELECT * FROM pgr_MaxFlowMinCost_Cost(
    'SELECT id,
     source, target,
     capacity, reverse_capacity,
     cost, reverse_cost FROM edge_table',
    ARRAY[7, 13], ARRAY[3, 9]
);
 pgr_maxflowmincost_cost
-------------------------
                     600
(1 row)
```

Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.

Column	Type	Default	Description
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Capacity of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Capacity of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* if it exists.
reverse_cost	ANY-NUMERICAL	0	Weight of the edge *(target, source)* if it exists.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

Type	Description
FLOAT	Minimum Cost Maximum Flow possible from the source(s) to the target(s)

See Also

- **Flow - Family of functions**
- **https://www.boost.org/libs/graph/doc/successive_shortest_path_nonnegative_weights.html**

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5 2.4 2.3**

Flow Functions General Information

The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets

pgr_maxFlow is the maximum Flow and that maximum is guaranteed to be the same on the functions **pgr_pushRelabel**, **pgr_edmondsKarp**, **pgr_boykovKolmogorov**, but the actual flow through each edge may vary.

Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in **Inner Query**.
source	BIGINT		Identifier of the starting vertex of the flow.
sources	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
target	BIGINT		Identifier of the ending vertex of the flow.
targets	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

For pgr_pushRelabel, pgr_edmondsKarp, pgr_boykovKolmogorov :

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

For **pgr_maxFlowMinCost - Experimental** and **pgr_maxFlowMinCost_Cost - Experimental**:

Edges SQL:
an SQL query of a directed graph of capacities, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
capacity	ANY-INTEGER		Capacity of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_capacity	ANY-INTEGER	-1	Capacity of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* if it exists.
reverse_cost	ANY-NUMERICAL	0	Weight of the edge *(target, source)* if it exists.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

For pgr_pushRelabel, pgr_edmondsKarp, pgr_boykovKolmogorov :

Column	Type	Description

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
start_vid	BIGINT	Identifier of the first end point vertex of the edge.
end_vid	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (start_vid, end_vid).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (start_vid, end_vid).

For pgr_maxFlowMinCost - Experimental

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge in the original query(edges_sql).
source	BIGINT	Identifier of the first end point vertex of the edge.
target	BIGINT	Identifier of the second end point vertex of the edge.
flow	BIGINT	Flow through the edge in the direction (source, target).
residual_capacity	BIGINT	Residual capacity of the edge in the direction (source, target).
cost	FLOAT	The cost of sending this flow through the edge in the direction (source, target).
agg_cost	FLOAT	The aggregate cost.

Adcanced Documentation

A flow network is a directed graph where each edge has a capacity and a flow. The flow through an edge must not exceed the capacity of the edge. Additionally, the incoming and outgoing flow of a node must be equal except for source which only has outgoing flow, and the destination(sink) which only has incoming flow.

Maximum flow algorithms calculate the maximum flow through the graph and the flow of each edge.

The maximum flow through the graph is guaranteed to be the same with all implementations, but the actual flow through each edge may vary. Given the following query:

pgr_maxFlow (edges_sql, source_vertex, sink_vertex)

where edges_sql = \{(id_i, source_i, target_i, capacity_i, reverse_capacity_i)\}

Graph definition

The weighted directed graph, G(V,E), is defined as:

- the set of vertices V
  - source_vertex \cup sink_vertex \bigcup source_i \bigcup target_i
- the set of edges E
  - E = \begin{cases} \text{ } \{(source_i, target_i, capacity_i) \text{ when } capacity > 0 \} & \quad \text{ if } reverse_capacity = \varnothing \\ \text{ } & \quad \text{ } \\ \{(source_i, target_i, capacity_i) \text{ when } capacity > 0 \} & \text{ } \\ \cup \{(target_i, source_i, reverse_capacity_i) \text{ when } reverse_capacity_i > 0)\} & \quad \text{ if } reverse_capacity \neq \varnothing \\ \end{cases}

Maximum flow problem

Given:

- G(V,E)
- source_vertex \in V the source vertex
- sink_vertex \in V the sink vertex

Then:

- pgr_maxFlow(edges_sql, source, sink) = \boldsymbol{\Phi}
- \boldsymbol{\Phi} = {(id_i, edge_id_i, source_i, target_i, flow_i, residual_capacity_i)}

Where:

\boldsymbol{\Phi} is a subset of the original edges with their residual capacity and flow. The maximum flow through the graph can be obtained by aggregating on the source or sink and summing the flow from/to it. In particular:

- id_i = i
- edge_id = id_i in edges_sql
- residual_capacity_i = capacity_i - flow_i

See Also

- **https://en.wikipedia.org/wiki/Maximum_flow_problem**

Indices and tables

- **Index**
- **Search Page**

Kruskal - Family of functions

- **pgr_kruskal**
- **pgr_kruskalBFS**
- **pgr_kruskalDD**
- **pgr_kruskalDFS**

Boost Graph Inside

pgr_kruskal

`pgr_kruskal` — Returns the minimum spanning tree of graph using Kruskal algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

This algorithm finds the minimum spanning forest in a possibly disconnected graph using Kruskal's algorithm.

The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time: O(E * log E)
- EMPTY SET is returned when there are no edges in the graph.

Signatures

Summary

```
pgr_kruskal(edges_sql)

RETURNS SET OF (seq, edge, cost)
OR EMPTY SET
```

Example:
Minimum Spanning Forest

```
SELECT * FROM pgr_kruskal(
    'SELECT id, source, target, cost, reverse_cost
        FROM edge_table ORDER BY id'
) ORDER BY edge;
 edge | cost
------+------
    1 |    1
    2 |    1
    3 |    1
    6 |    1
    7 |    1
   10 |    1
   11 |    1
   12 |    1
   13 |    1
   14 |    1
   15 |    1
   16 |    1
   17 |    1
   18 |    1
(14 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (edge, cost)

Column	Type	Description
edge	BIGINT	Identifier of the edge.
cost	FLOAT	Cost to traverse the edge.

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Kruskal's algorithm documentation**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_kruskalBFS

pgr_kruskalBFS — Prim algorithm for Minimum Spanning Tree with Depth First Search ordering.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

Visits and extracts the nodes information in Breath First Search ordering of the Minimum Spanning Tree created using Prims's algorithm.

The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time: O(E * log E)
- Returned tree nodes from a root vertex are on Breath First Search order
- Breath First Search Running time: O(E + V)

Signatures

```
pgr_kruskalBFS(Edges SQL, Root vid [, max_depth])
pgr_kruskalBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single vertex

```
pgr_kruskalBFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree having as root vertex 2

```
SELECT * FROM pgr_kruskalBFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     3 |         2 |    9 |   16 |    1 |        3
   6 |     4 |         2 |   12 |   15 |    1 |        4
   7 |     5 |         2 |   11 |   13 |    1 |        5
   8 |     6 |         2 |    6 |   11 |    1 |        6
   9 |     6 |         2 |   10 |   12 |    1 |        6
  10 |     7 |         2 |    5 |   10 |    1 |        7
  11 |     7 |         2 |   13 |   14 |    1 |        7
  12 |     8 |         2 |    8 |    7 |    1 |        8
  13 |     9 |         2 |    7 |    6 |    1 |        9
(13 rows)
```

Multiple vertices

```
pgr_kruskalBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:

The Minimum Spanning Tree starting on vertices \{13, 2\} with depth <= 3

```
SELECT * FROM pgr_kruskalBFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[13,2], max_depth := 3
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     3 |         2 |    9 |   16 |    1 |        3
   6 |     0 |        13 |   13 |   -1 |    0 |        0
   7 |     1 |        13 |   10 |   14 |    1 |        1
   8 |     2 |        13 |    5 |   10 |    1 |        2
   9 |     2 |        13 |   11 |   12 |    1 |        2
  10 |     3 |        13 |    8 |    7 |    1 |        3
  11 |     3 |        13 |    6 |   11 |    1 |        3
  12 |     3 |        13 |   12 |   13 |    1 |        3
(12 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single vertex** • When value is 0 then gets the spanning forest starting in aleatory nodes for each tree in the forest.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple vertices** • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Optional Parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit for depth of node in the tree • When value is Negative then **throws error**

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
depth	BIGINT	Depth of the node. • 0 when node = start_vid.
start_vid	BIGINT	Identifier of the root vertex. • I n **Multiple Vertices** results are in ascending order.
node	BIGINT	Identifier of node reached using edge.
edge	BIGINT	Identifier of the edge used to arrive to node. • -1 when node = start_vid.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Kruskal's algorithm documentation**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_kruskalDD

pgr_kruskalDD — Catchament nodes using Kruskal's algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

Using Kruskal's algorithm, extracts the nodes that have aggregate costs less than or equal to the valuDistance from a **root** vertex (or vertices) within the calculated minimum spanning tree.

The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time: O(E * log E)
- Returned tree nodes from a root vertex are on Depth First Search order.
- Depth First Search running time: O(E + V)

Signatures

```
pgr_kruskalDD(edges_sql, root_vid, distance)
pgr_kruskalDD(edges_sql, root_vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single vertex

```
pgr_kruskalDD(edges_sql, root_vid, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertex 2 with agg_cost <= 3.5

```
SELECT * FROM pgr_kruskalDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2, 3.5
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     3 |         2 |    9 |   16 |    1 |        3
(5 rows)
```

Multiple vertices

```
pgr_kruskalDD(edges_sql, root_vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertices \{13, 2\} with agg_cost <= 3.5;

```
SELECT * FROM pgr_kruskalDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[13,2],
    3.5
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     3 |         2 |    9 |   16 |    1 |        3
   6 |     0 |        13 |   13 |   -1 |    0 |        0
   7 |     1 |        13 |   10 |   14 |    1 |        1
   8 |     2 |        13 |    5 |   10 |    1 |        2
   9 |     3 |        13 |    8 |    7 |    1 |        3
  10 |     2 |        13 |   11 |   12 |    1 |        2
  11 |     3 |        13 |    6 |   11 |    1 |        3
  12 |     3 |        13 |   12 |   13 |    1 |        3
(12 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single vertex** • When 0 gets the spanning forest starting in aleatory nodes for each tree.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple vertices** • 0 values are ignored • For optimization purposes, any duplicated value is ignored.
Distance	ANY-NUMERIC	Upper limit for the inclusion of the node in the result. • When the value is Negative **throws error**

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT
ANY-NUMERIC:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT, NUMERIC

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
depth	BIGINT	Depth of the node. • 0 when node = start_vid.
start_vid	BIGINT	Identifier of the root vertex. • I n **Multiple Vertices** results are in ascending order.
node	BIGINT	Identifier of node reached using edge.
edge	BIGINT	Identifier of the edge used to arrive to node. • -1 when node = start_vid.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Kruskal's algorithm documentation**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_kruskalDFS

pgr_kruskalDFS — Kruskal algorithm for Minimum Spanning Tree with Depth First Search ordering.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

Visits and extracts the nodes information in Depth First Search ordering of the Minimum Spanning Tree created using Kruskal's algorithm.

The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time: O(E * log E)
- Returned tree nodes from a root vertex are on Depth First Search order
- Depth First Search Running time: O(E + V)

Signatures

```
pgr_kruskalDFS(Edges SQL, Root vid [, max_depth])
pgr_kruskalDFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single vertex

```
pgr_kruskalDFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertex 2

```
SELECT * FROM pgr_kruskalDFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     3 |         2 |    9 |   16 |    1 |        3
   6 |     4 |         2 |   12 |   15 |    1 |        4
   7 |     5 |         2 |   11 |   13 |    1 |        5
   8 |     6 |         2 |    6 |   11 |    1 |        6
   9 |     6 |         2 |   10 |   12 |    1 |        6
  10 |     7 |         2 |    5 |   10 |    1 |        7
  11 |     8 |         2 |    8 |    7 |    1 |        8
  12 |     9 |         2 |    7 |    6 |    1 |        9
  13 |     7 |         2 |   13 |   14 |    1 |        7
(13 rows)
```

Multiple vertices

```
pgr_kruskalDFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertices \{13, 2\} with depth <= 3

```
SELECT * FROM pgr_kruskalDFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[13,2], max_depth := 3
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     3 |         2 |    9 |   16 |    1 |        3
   6 |     0 |        13 |   13 |   -1 |    0 |        0
   7 |     1 |        13 |   10 |   14 |    1 |        1
   8 |     2 |        13 |    5 |   10 |    1 |        2
   9 |     3 |        13 |    8 |    7 |    1 |        3
  10 |     2 |        13 |   11 |   12 |    1 |        2
  11 |     3 |        13 |    6 |   11 |    1 |        3
  12 |     3 |        13 |   12 |   13 |    1 |        3
(12 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single vertex** • When value is 0 then gets the spanning forest starting in aleatory nodes for each tree in the forest.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple vertices** • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Optional Parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit for depth of node in the tree • When value is Negative then **throws error**

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
depth	BIGINT	Depth of the node. • 0 when node = start_vid.

Column	Type	Description
start_vid	BIGINT	Identifier of the root vertex. • I n **Multiple Vertices** results are in ascending order.
node	BIGINT	Identifier of `node` reached using `edge`.
edge	BIGINT	Identifier of the `edge` used to arrive to `node`. • -1 when `node` = `start_vid`.
cost	FLOAT	Cost to traverse `edge`.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `node`.

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Kruskal's algorithm documentation**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**)

Description

Kruskal's algorithm is a greedy minimum spanning tree algorithm that in each cycle finds and adds the edge of the least possible weight that connects any two trees in the forest.

The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time: O(E * log E)

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

See Also

- **Spanning Tree - Category**

- **Boost: Kruskal's algorithm documentation**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

- **Index**
- **Search Page**

Prim - Family of functions

- **pgr_prim**
- **pgr_primBFS**
- **pgr_primDD**
- **pgr_primDFS**

Boost Graph Inside

pgr_prim

`pgr_prim` — Minimum spanning forest of graph using Prim algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

This algorithm finds the minimum spanning forest in a possibly disconnected graph using Prim's algorithm.

The main characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time: O(E*log V)
- EMPTY SET is returned when there are no edges in the graph.

Signatures

Summary

```
pgr_prim(edges_sql)

RETURNS SET OF (edge, cost)
OR EMPTY SET
```

Example:
Minimum Spanning Forest of a subgraph

```
SELECT edge, cost FROM pgr_prim(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table WHERE id < 14'
) ORDER BY edge;
 edge | cost
------+------
    1 |    1
    2 |    1
    3 |    1
    4 |    1
    5 |    1
    6 |    1
    7 |    1
    9 |    1
   10 |    1
   11 |    1
   13 |    1
(11 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (edge, cost)

Column	Type	Description
edge	BIGINT	Identifier of the edge.
cost	FLOAT	Cost to traverse the edge.

See Also

- **Spanning Tree - Category**
- **Prim - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_primBFS

pgr_primBFS — Prim's algorithm for Minimum Spanning Tree with Depth First Search ordering.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

Visits and extracts the nodes information in Breath First Search ordering of the Minimum Spanning Tree created with Prims's algorithm.

The main Characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time: O(E*log V)
- Returned tree nodes from a root vertex are on Breath First Search order
- Breath First Search Running time: O(E + V)

Signatures

```
pgr_primBFS(Edges SQL, Root vid [, max_depth])
pgr_primBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single vertex

```
pgr_primBFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree having as root vertex 2

```
SELECT * FROM pgr_primBFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     1 |         2 |    5 |    4 |    1 |        1
   5 |     2 |         2 |    4 |    3 |    1 |        2
   6 |     2 |         2 |    6 |    5 |    1 |        2
   7 |     2 |         2 |    8 |    7 |    1 |        2
   8 |     2 |         2 |   10 |   10 |    1 |        2
   9 |     3 |         2 |    9 |    9 |    1 |        3
  10 |     3 |         2 |   11 |   11 |    1 |        3
  11 |     3 |         2 |    7 |    6 |    1 |        3
  12 |     3 |         2 |   13 |   14 |    1 |        3
  13 |     4 |         2 |   12 |   13 |    1 |        4
(13 rows)
```

Multiple vertices

```
pgr_primBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertices \{13, 2\} with depth <= 3

```
SELECT * FROM pgr_primBFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[13,2], max_depth := 3
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     1 |         2 |    5 |    4 |    1 |        1
   5 |     2 |         2 |    4 |    3 |    1 |        2
   6 |     2 |         2 |    6 |    5 |    1 |        2
   7 |     2 |         2 |    8 |    7 |    1 |        2
   8 |     2 |         2 |   10 |   10 |    1 |        2
   9 |     3 |         2 |    9 |    9 |    1 |        3
  10 |     3 |         2 |   11 |   11 |    1 |        3
  11 |     3 |         2 |    7 |    6 |    1 |        3
  12 |     3 |         2 |   13 |   14 |    1 |        3
  13 |     0 |        13 |   13 |   -1 |    0 |        0
  14 |     1 |        13 |   10 |   14 |    1 |        1
  15 |     2 |        13 |    5 |   10 |    1 |        2
  16 |     3 |        13 |    2 |    4 |    1 |        3
  17 |     3 |        13 |    8 |    7 |    1 |        3
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single vertex** • When value is 0 then gets the spanning forest starting in aleatory nodes for each tree in the forest.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple vertices** • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Optional Parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit for depth of node in the tree • When value is Negative then **throws error**

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.

Column	Type	Description
depth	BIGINT	Depth of the node. • 0 when node = start_vid.
start_vid	BIGINT	Identifier of the root vertex. • I n **Multiple Vertices** results are in ascending order.
node	BIGINT	Identifier of node reached using edge.
edge	BIGINT	Identifier of the edge used to arrive to node. • -1 when node = start_vid.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

See Also

- **Spanning Tree - Category**
- **Prim - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_primDD

pgr_primDD — Catchament nodes using Prim's algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

Using Prim algorithm, extracts the nodes that have aggregate costs less than or equal to the valuDistance within the calculated minimum spanning tree.

The main Characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time: O(E*log V)
- Returned tree nodes from a root vertex are on Depth First Search order.
- Depth First Search running time: O(E + V)

Signatures

Summary

```
pgr_prim(Edges SQL, root vid, distance)
pgr_prim(Edges SQL, root vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single vertex

```
pgr_primDD(Edges SQL, root vid, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertex 2 with agg_cost <= 3.5

```
SELECT * FROM pgr_primDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2, 3.5
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     2 |         2 |    6 |    5 |    1 |        2
   6 |     3 |         2 |    9 |    9 |    1 |        3
   7 |     3 |         2 |   11 |   11 |    1 |        3
   8 |     1 |         2 |    5 |    4 |    1 |        1
   9 |     2 |         2 |    8 |    7 |    1 |        2
  10 |     3 |         2 |    7 |    6 |    1 |        3
  11 |     2 |         2 |   10 |   10 |    1 |        2
  12 |     3 |         2 |   13 |   14 |    1 |        3
(12 rows)
```

Multiple vertices

```
pgr_primDD(Edges SQL, root vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree starting on vertices \{13, 2\} with agg_cost <= 3.5;

```
SELECT * FROM pgr_primDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[13,2], 3.5
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     2 |         2 |    6 |    5 |    1 |        2
   6 |     3 |         2 |    9 |    9 |    1 |        3
   7 |     3 |         2 |   11 |   11 |    1 |        3
   8 |     1 |         2 |    5 |    4 |    1 |        1
   9 |     2 |         2 |    8 |    7 |    1 |        2
  10 |     3 |         2 |    7 |    6 |    1 |        3
  11 |     2 |         2 |   10 |   10 |    1 |        2
  12 |     3 |         2 |   13 |   14 |    1 |        3
  13 |     0 |        13 |   13 |   -1 |    0 |        0
  14 |     1 |        13 |   10 |   14 |    1 |        1
  15 |     2 |        13 |    5 |   10 |    1 |        2
  16 |     3 |        13 |    2 |    4 |    1 |        3
  17 |     3 |        13 |    8 |    7 |    1 |        3
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single vertex** • When 0 gets the spanning forest starting in aleatory nodes for each tree.

Parameter	Type	Description
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple vertices** • 0 values are ignored • For optimization purposes, any duplicated value is ignored.
Distance	ANY-NUMERIC	Upper limit for the inclusion of the node in the result. • When the value is Negative **throws error**

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERIC:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT, NUMERIC

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
depth	BIGINT	Depth of the node. • 0 when node = start_vid.
start_vid	BIGINT	Identifier of the root vertex. • I n **Multiple Vertices** results are in ascending order.
node	BIGINT	Identifier of node reached using edge.
edge	BIGINT	Identifier of the edge used to arrive to node. • -1 when node = start_vid.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

See Also

- **Spanning Tree - Category**
- **Prim - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_primDFS

`pgr_primDFS` — Prim algorithm for Minimum Spanning Tree with Depth First Search ordering.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**)

Description

Visits and extracts the nodes information in Depth First Search ordering of the Minimum Spanning Tree created using Prims's algorithm.

The main Characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time: O(E*log V)
- Returned tree nodes from a root vertex are on Depth First Search order
- Depth First Search Running time: O(E + V)

Signatures

```
pgr_primDFS(Edges SQL, Root vid [, max_depth])
pgr_primDFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single vertex

```
pgr_primDFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Minimum Spanning Tree having as root vertex 2

```
SELECT * FROM pgr_primDFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     2 |         2 |    6 |    5 |    1 |        2
   6 |     3 |         2 |    9 |    9 |    1 |        3
   7 |     3 |         2 |   11 |   11 |    1 |        3
   8 |     4 |         2 |   12 |   13 |    1 |        4
   9 |     1 |         2 |    5 |    4 |    1 |        1
  10 |     2 |         2 |    8 |    7 |    1 |        2
  11 |     3 |         2 |    7 |    6 |    1 |        3
  12 |     2 |         2 |   10 |   10 |    1 |        2
  13 |     3 |         2 |   13 |   14 |    1 |        3
(13 rows)
```

Multiple vertices

```
pgr_primDFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:

The Minimum Spanning Tree starting on vertices \{13, 2\} with depth <= 3

```
SELECT * FROM pgr_primDFS(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[13,2], max_depth := 3
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    3 |    2 |    1 |        1
   4 |     2 |         2 |    4 |    3 |    1 |        2
   5 |     2 |         2 |    6 |    5 |    1 |        2
   6 |     3 |         2 |    9 |    9 |    1 |        3
   7 |     3 |         2 |   11 |   11 |    1 |        3
   8 |     1 |         2 |    5 |    4 |    1 |        1
   9 |     2 |         2 |    8 |    7 |    1 |        2
  10 |     3 |         2 |    7 |    6 |    1 |        3
  11 |     2 |         2 |   10 |   10 |    1 |        2
  12 |     3 |         2 |   13 |   14 |    1 |        3
  13 |     0 |        13 |   13 |   -1 |    0 |        0
  14 |     1 |        13 |   10 |   14 |    1 |        1
  15 |     2 |        13 |    5 |   10 |    1 |        2
  16 |     3 |        13 |    2 |    4 |    1 |        3
  17 |     3 |        13 |    8 |    7 |    1 |        3
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single vertex** • When value is 0 then gets the spanning forest starting in aleatory nodes for each tree in the forest.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple vertices** • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Optional Parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit for depth of node in the tree • When value is Negative then **throws error**

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
depth	BIGINT	Depth of the node. • 0 when node = start_vid.
start_vid	BIGINT	Identifier of the root vertex. • I n **Multiple Vertices** results are in ascending order.
node	BIGINT	Identifier of node reached using edge.
edge	BIGINT	Identifier of the edge used to arrive to node. • -1 when node = start_vid.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

See Also

- **Spanning Tree - Category**
- **Prim - Family of functions**
- The queries use the **Sample Data** network.
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

- **Index**
- **Search Page**

- **Supported versions:** current(**3.0**)

Description

The prim algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník. It is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

This algorithms find the minimum spanning forest in a possibly disconnected graph; in contrast, the most basic form of Prim's algorithm only finds minimum spanning trees in connected graphs. However, running Prim's algorithm separately for each connected component of the graph, then it is called minimum spanning forest.

The main characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time: O(E*log V)

Note

From boost Graph: "The algorithm as implemented in Boost.Graph does not produce correct results on graphs with parallel edges."

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.

Column	Type	Default	Description
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

See Also

- **Spanning Tree - Category**
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

- **Index**
- **Search Page**

Topology - Family of Functions

The pgRouting's topology of a network, represented with an edge table with source and target attributes and a vertices table associated with it. Depending on the algorithm, you can create a topology or just reconstruct the vertices table, You can analyze the topology, We also provide a function to node an unoded network.

- **pgr_createTopology** - to create a topology based on the geometry.
- **pgr_createVerticesTable** - to reconstruct the vertices table based on the source and target information.
- **pgr_analyzeGraph** - to analyze the edges and vertices of the edge table.
- **pgr_analyzeOneWay** - to analyze directionality of the edges.
- **pgr_nodeNetwork** -to create nodes to a not noded edge table.

Experimental

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

- **pgr_extractVertices – Experimental** - Extracts vertices information based on the source and target.

`pgr_createTopology` — Builds a network topology based on the geometry information.

Availability

- Version 2.0.0
  - Renamed from version 1.x
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The function returns:

- `OK` after the network topology has been built and the vertices table created.
- `FAIL` when the network topology was not built due to an error.

Signatures

```
varchar pgr_createTopology(text edge_table, double precision tolerance,
                text the_geom:='the_geom', text id:='id',
                text source:='source',text target:='target',
                text rows_where:='true', boolean clean:=false)
```

Parameters

The topology creation function accepts the following parameters:

edge_table:
`text` Network table name. (may contain the schema name AS well)
tolerance:
`float8` Snapping tolerance of disconnected edges. (in projection unit)
the_geom:
`text` Geometry column name of the network table. Default value is `the_geom`.
id:
`text` Primary key column name of the network table. Default value is `id`.
source:
`text` Source column name of the network table. Default value is`source`.
target:
`text` Target column name of the network table. Default value is `target`.
rows_where:
`text` Condition to SELECT a subset or rows. Default value is `true` to indicate all rows that where`source` or `target` have a null value, otherwise the condition is used.
clean:
`boolean` Clean any previous topology. Default value is `false`.

Warning

The `edge_table` will be affected

- The `source` column values will change.
- The `target` column values will change.
  - An index will be created, if it doesn't exists, to speed up the process to the following columns:
    - `id`
    - `the_geom`
    - `source`
    - `target`

The function returns:

- `OK` after the network topology has been built.
  - Creates a vertices table: <edge_table>_vertices_pgr.
  - Fills `id` and `the_geom` columns of the vertices table.
  - Fills the source and target columns of the edge table referencing the`id` of the vertices table.
- `FAIL` when the network topology was not built due to an error:
  - A required column of the Network table is not found or is not of the appropriate type.

- The condition is not well formed.
- The names of source , target or id are the same.
- The SRID of the geometry could not be determined.

The Vertices Table

The vertices table is a requirement of the **pgr_analyzeGraph** and the **pgr_analyzeOneWay** functions.

The structure of the vertices table is:

id:
`bigint` Identifier of the vertex.
cnt:
`integer` Number of vertices in the edge_table that reference this vertex. See **pgr_analyzeGraph**.
chk:
`integer` Indicator that the vertex might have a problem. See **pgr_analyzeGraph**.
ein:
`integer` Number of vertices in the edge_table that reference this vertex AS incoming. See **pgr_analyzeOneWay**.
eout:
`integer` Number of vertices in the edge_table that reference this vertex AS outgoing. See **pgr_analyzeOneWay**.
the_geom:
`geometry` Point geometry of the vertex.

Usage when the edge table's columns MATCH the default values:

The simplest way to use pgr_createTopology is:

```
SELECT  pgr_createTopology('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

When the arguments are given in the order described in the parameters:

We get the same result AS the simplest way to use the function.

```
SELECT  pgr_createTopology('edge_table', 0.001,
   'the_geom', 'id', 'source', 'target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

Warning

An error would occur when the arguments are not given in the appropriate order:
In this example, the column `id` of the table `ege_table` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the id column.

```
SELECT  pgr_createTopology('edge_table', 0.001,
    'id', 'the_geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'id', 'the_geom', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  ----> PGR ERROR in pgr_createTopology: Wrong type of Column id:the_geom
NOTICE:  Unexpected error raise_exception
 pgr_createtopology
--------------------
 FAIL
(1 row)
```

When using the named notation

Parameters defined with a default value can be omitted, as long as the value matches the default And The order of the parameters would not matter.

```
SELECT  pgr_createTopology('edge_table', 0.001,
    the_geom:='the_geom', id:='id', source:='source', target:='target');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('edge_table', 0.001,
    source:='source', id:='id', target:='target', the_geom:='the_geom');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('edge_table', 0.001, source:='source');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Selecting rows based on the id.

```
SELECT  pgr_createTopology('edge_table', 0.001, rows_where:='id < 10');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of row with `id = 5`.

```
SELECT  pgr_createTopology('edge_table', 0.001,
    rows_where:='the_geom && (SELECT st_buffer(the_geom, 0.05) FROM edge_table WHERE id=5)');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of the row with `gid` =100 of the table `othertable`.

```
CREATE TABLE otherTable AS  (SELECT 100 AS gid,  st_point(2.5, 2.5) AS other_geom);
SELECT 1
SELECT  pgr_createTopology('edge_table', 0.001,
    rows_where:='the_geom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100)');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Usage when the edge table's columns DO NOT MATCH the default values:

For the following table

```
CREATE TABLE mytable AS (SELECT id AS gid,  the_geom AS mygeom, source AS src , target AS tgt FROM edge_table) ;
SELECT 18
```

Using positional notation:

The arguments need to be given in the order described in the parameters.

Note that this example uses clean flag. So it recreates the whole vertices table.

```
SELECT  pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', clean := TRUE);
 pgr_createtopology
--------------------
 OK
(1 row)
```

Warning

An error would occur when the arguments are not given in the appropriate order:
In this example, the column `gid` of the table `mytable` is passed to the function AS the geometry column, and the geometry column `mygeom` is passed to the function AS the id column.

```
SELECT  pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  ----> PGR ERROR in pgr_createTopology: Wrong type of Column id:mygeom
NOTICE:  Unexpected error raise_exception
 pgr_createtopology
--------------------
 FAIL
(1 row)
```

When using the named notation

In this scenario omitting a parameter would create an error because the default values for the column names do not match the column names of the table. The order of the parameters do not matter:

```
SELECT  pgr_createTopology('mytable', 0.001, the_geom:='mygeom', id:='gid', source:='src', target:='tgt');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Based on id:

```
SELECT  pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', rows_where:='gid < 10');
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT  pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom', rows_where:='gid < 10');
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT  pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt',
   rows_where:='mygeom && (SELECT st_buffer(mygeom, 1) FROM mytable WHERE gid=5)');
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT  pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom',
   rows_where:='mygeom && (SELECT st_buffer(mygeom, 1) FROM mytable WHERE gid=5)');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of the row with `gid` =100 of the table `othertable`.

```
SELECT  pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt',
   rows_where:='mygeom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100)');
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT  pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom',
   rows_where:='mygeom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100)');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Additional Examples

Example:
With full output

This example start a clean topology, with 5 edges, and then its incremented to the rest of the edges.

```
SELECT pgr_createTopology('edge_table',  0.001, rows_where:='id < 6', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'id < 6', clean := t)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  5 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT pgr_createTopology('edge_table',  0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  13 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

The example uses the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr_createVerticesTable** to reconstruct the vertices table based on the source and target information.
- **pgr_analyzeGraph** to analyze the edges and vertices of the edge table.

Indices and tables

- **Index**
- **Search Page**

pgr_extractVertices - Experimental

pgr_extractVertices — Extracts the vertices information based on the source and target.

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

This is an auxiliary function for extracting the vertex information of the set of edges of a graph.

- When the edge identifier is given, then it will also calculate the in and out edges

Signatures

```
pgr_extractVertices(Edges SQL [, dryrun])
RETURNS SETOF (id, in_edges, out_edges, x, y, geom)
```

Example:
Extracting the vertex information

```
SELECT  * FROM pgr_extractVertices(
   'SELECT id, the_geom AS geom
   FROM edge_table');
 id | in_edges | out_edges |      x       |  y  |                  geom
----+----------+-----------+--------------+-----+---------------------------------------
  1 |          | {6}       |            0 |   2 | 01010000000000000000000000000000000000000040
  2 |          | {17}      |          0.5 | 3.5 | 0101000000000000000000E03F0000000000000C40
  3 | {6}      | {7}       |            1 |   2 | 0101000000000000000000F03F0000000000000040
  4 | {17}     |           | 1.999999999999 | 3.5 | 01010000006 8EEFFFFFFFFFF3F0000000000000C40
  5 |          | {1}       |            2 |   0 | 01010000000000000000000040000000000000000
  6 | {1}      | {2,4}     |            2 |   1 | 0101000000000000000000004000000000000F03F
  7 | {4,7}    | {8,10}    |            2 |   2 | 0101000000000000000000040000000000000040
  8 | {10}     | {12,14}   |            2 |   3 | 01010000000000000000000040000000000000840
  9 | {14}     |           |            2 |   4 | 01010000000000000000000040000000000001040
 10 | {2}      | {3,5}     |            3 |   1 | 0101000000000000000000084000000000000F03F
 11 | {5,8}    | {9,11}    |            3 |   2 | 01010000000000000000000840000000000000040
 12 | {11,12}  | {13}      |            3 |   3 | 01010000000000000000000840000000000000840
 13 |          | {18}      |          3.5 | 2.3 | 0101000000000000000000C406666666666660240
 14 | {18}     |           |          3.5 |   4 | 0101000000000000000000C400000000000001040
 15 | {3}      | {16}      |            4 |   1 | 01010000000000000000000104000000000000F03F
 16 | {9,16}   | {15}      |            4 |   2 | 01010000000000000000000104000000000000040
 17 | {13,15}  |           |            4 |   3 | 01010000000000000000000104000000000000840
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	The set of edges of the graph. It is an **Inner Query** as described below.
dryrun	TEXT	Don't process and get in a NOTICE the resulting query.

Inner Query

When line geometry is known

Column	Type	Description
id	BIGINT	(Optional) identifier of the edge.
geom	LINESTRING	LINESTRING geometry of the edge.

This inner query takes precedence over the next two inner query, therefore other columns are ignored when `geom` column appears.

- Ignored columns:
  - `startpoint`
  - `endpoint`
  - `source`
  - `target`

When vertex geometry is known

To use this inner query the column `geom` should not be part of the set of columns.

Column	Type	Description
id	BIGINT	(Optional) identifier of the edge.
startpoint	POINT	POINT geometry of the starting vertex.
endpoint	POINT	POINT geometry of the ending vertex.

This inner query takes precedence over the next inner query, therefore other columns are ignored when `startpoint` and `endpoint` columns appears.

- Ignored columns:
  - `source`
  - `target`

When identifiers of vertices are known

To use this inner query the columns `geom`, `startpoint` and `endpoint` should not be part of the set of columns.

Column	Type	Description
id	BIGINT	(Optional) identifier of the edge.
source	ANY-INTEGER	Identifier of the first end point vertex of the edge.

Column	Type	Description
target	ANY-INTEGER	Identifier of the second end point vertex of the edge.

Result Columns

Rreturns set of (id, in_edges, out_edges, x, y, geom)

Column	Type	Description
id	BIGINT	Identifier of the first end point vertex of the edge.
in_edges	BIGINT[]	Array of identifiers of the edges that have the vertex `id` as *first end point*. • `NULL` When the `id` is not part of the inner query
out_edges	BIGINT[]	Array of identifiers of the edges that have the vertex `id` as *second end point*. • `NULL` When the `id` is not part of the inner query
x	FLOAT	X value of the POINT geometry • `NULL` When no geometry is provided
y	FLOAT	Y value of the POINT geometry • `NULL` When no geometry is provided
geom	POINT	Geometry of the POINT • `NULL` When no geometry is provided

Additional Examples

Example 1:
Dryrun execution

To get the query generated used to get the vertex information, use `dryrun := true`.

The results can be used as base code to make a refinement based on the backend development needs.

```
SELECT  * FROM pgr_extractVertices(
   'SELECT id, the_geom AS geom FROM edge_table',
   dryrun := true);
NOTICE:
      WITH

      main_sql AS (
        SELECT id, the_geom AS geom FROM edge_table
      ),

      the_out AS (
        SELECT id::BIGINT AS out_edge, ST_StartPoint(geom) AS geom
        FROM main_sql
      ),

      agg_out AS (
        SELECT array_agg(out_edge ORDER BY out_edge) AS out_edges, ST_x(geom) AS x, ST_Y(geom) AS y, geom
        FROM the_out
        GROUP BY geom
      ),

      the_in AS (
        SELECT id::BIGINT AS in_edge, ST_EndPoint(geom) AS geom
        FROM main_sql
      ),

      agg_in AS (
        SELECT array_agg(in_edge ORDER BY in_edge) AS in_edges, ST_x(geom) AS x, ST_Y(geom) AS y, geom
        FROM the_in
        GROUP BY geom
      ),

      the_points AS (
        SELECT in_edges, out_edges, coalesce(agg_out.geom, agg_in.geom) AS geom
        FROM agg_out
        FULL OUTER JOIN agg_in USING (x, y)
      )

      SELECT row_number() over(ORDER BY ST_X(geom), ST_Y(geom)) AS id, in_edges, out_edges, ST_X(geom), ST_Y(geom), geom
      FROM the_points;
 id | in_edges | out_edges | x | y | geom
----+----------+-----------+---+---+------
(0 rows)
```

Example 2:
Creating a routing topology

1. Making sure the database does not have the `vertices_table`

```
DROP TABLE IF EXISTS vertices_table;
NOTICE:  table "vertices_table" does not exist, skipping
DROP TABLE
```

2. Cleaning up the columns of the rotuing topology to be created

```
UPDATE edge_table
SET source = NULL, target = NULL,
  x1 = NULL, y1 = NULL,
  x2 = NULL, y2 = NULL;
UPDATE 18
```

3. Creating the vertices table

```
SELECT  * INTO vertices_table
FROM pgr_extractVertices('SELECT id, the_geom AS geom FROM edge_table');
SELECT 17
```

4. Inspection of the vertices table

```
SELECT *
FROM vertices_table;
 id | in_edges | out_edges |       x        |  y  |                    geom
----+----------+-----------+----------------+-----+--------------------------------------------
  1 |          | {6}       |              0 |   2 | 0101000000000000000000000000000000000040
  2 |          | {17}      |            0.5 | 3.5 | 0101000000000000000000E03F0000000000000C40
  3 | {6}      | {7}       |              1 |   2 | 0101000000000000000000F03F0000000000000040
  4 | {17}     |           | 1.999999999999 | 3.5 | 010100000068EEFFFFFFFFFF3F0000000000000C40
  5 |          | {1}       |              2 |   0 | 0101000000000000000000004000000000000000
  6 | {1}      | {2,4}     |              2 |   1 | 010100000000000000000000040000000000000F03F
  7 | {4,7}    | {8,10}    |              2 |   2 | 0101000000000000000000004000000000000040
  8 | {10}     | {12,14}   |              2 |   3 | 010100000000000000000000400000000000000840
  9 | {14}     |           |              2 |   4 | 0101000000000000000000004000000000000001040
 10 | {2}      | {3,5}     |              3 |   1 | 010100000000000000000008400000000000000F03F
 11 | {5,8}    | {9,11}    |              3 |   2 | 0101000000000000000000084000000000000000040
 12 | {11,12}  | {13}      |              3 |   3 | 01010000000000000000000840000000000000000840
 13 |          | {18}      |            3.5 | 2.3 | 010100000000000000000000C406666666666660240
 14 | {18}     |           |            3.5 |   4 | 010100000000000000000000C400000000000001040
 15 | {3}      | {16}      |              4 |   1 | 0101000000000000000000010400000000000000F03F
 16 | {9,16}   | {15}      |              4 |   2 | 010100000000000000000001040000000000000040
 17 | {13,15}  |           |              4 |   3 | 010100000000000000000001040000000000000840
(17 rows)
```

5. Creating the routing topology on the edge table

   Updating the `source` information

```
WITH
  out_going AS (
    SELECT id AS vid, unnest(out_edges) AS eid, x, y
    FROM vertices_table
  )
UPDATE edge_table
SET source = vid, x1 = x, y1 = y
FROM out_going WHERE id = eid;
UPDATE 18
```

   Updating the `target` information

```
WITH
  in_coming AS (
    SELECT id AS vid, unnest(in_edges) AS eid, x, y
    FROM vertices_table
  )
UPDATE edge_table
SET target = vid, x2 = x, y2 = y
FROM in_coming WHERE id = eid;
UPDATE 18
```

6. Inspection of the routing topology

```
SELECT id, source, target, x1, y1, x2, y2
FROM edge_table;
 id | source | target | x1  | y1  |       x2       | y2
----+--------+--------+-----+-----+----------------+----
  6 |      1 |      3 |   0 |   2 |              1 |  2
 17 |      2 |      4 | 0.5 | 3.5 | 1.999999999999 |  3.5
  1 |      5 |      6 |   2 |   0 |              2 |  1
  4 |      6 |      7 |   2 |   1 |              2 |  2
  7 |      3 |      7 |   1 |   2 |              2 |  2
 10 |      7 |      8 |   2 |   2 |              2 |  3
 14 |      8 |      9 |   2 |   3 |              2 |  4
  2 |      6 |     10 |   2 |   1 |              3 |  1
  5 |     10 |     11 |   3 |   1 |              3 |  2
  8 |      7 |     11 |   2 |   2 |              3 |  2
 11 |     11 |     12 |   3 |   2 |              3 |  3
 12 |      8 |     12 |   2 |   3 |              3 |  3
 18 |     13 |     14 | 3.5 | 2.3 |            3.5 |  4
  3 |     10 |     15 |   3 |   1 |              4 |  1
  9 |     11 |     16 |   3 |   2 |              4 |  2
 16 |     15 |     16 |   4 |   1 |              4 |  2
 13 |     12 |     17 |   3 |   3 |              4 |  3
 15 |     16 |     17 |   4 |   2 |              4 |  3
(18 rows)
```

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr_createVerticesTable** to create a topology based on the geometry.

Indices and tables

- **Index**
- **Search Page**

pgr_createVerticesTable

`pgr_createVerticesTable` — Reconstructs the vertices table based on the source and target information.

Availability

- Version 2.0.0
  - Renamed from version 1.x
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The function returns:

- `OK` after the vertices table has been reconstructed.
- `FAIL` when the vertices table was not reconstructed due to an error.

Signatures

```
    pgr_createVerticesTable(edge_table, the_geom, source, target, rows_where)
RETURNS VARCHAR
```

Parameters

The reconstruction of the vertices table function accepts the following parameters:

edge_table:
`text` Network table name. (may contain the schema name as well)
the_geom:
`text` Geometry column name of the network table. Default value is `the_geom`.
source:
`text` Source column name of the network table. Default value is`source`.
target:
`text` Target column name of the network table. Default value is `target`.
rows_where:
`text` Condition to SELECT a subset or rows. Default value is `true` to indicate all rows.

Warning

The edge_table will be affected

- An index will be created, if it doesn't exists, to speed up the process to the following columns:
  - the_geom
  - source
  - target

The function returns:

- OK after the vertices table has been reconstructed.
  - Creates a vertices table: <edge_table>_vertices_pgr.
  - Fills id and the_geom columns of the vertices table based on the source and target columns of the edge table.
- FAIL when the vertices table was not reconstructed due to an error.
  - A required column of the Network table is not found or is not of the appropriate type.
  - The condition is not well formed.
  - The names of source, target are the same.
  - The SRID of the geometry could not be determined.

The Vertices Table

The vertices table is a requierment of the **pgr_analyzeGraph** and the **pgr_analyzeOneWay** functions.

The structure of the vertices table is:

id:
bigint Identifier of the vertex.
cnt:
integer Number of vertices in the edge_table that reference this vertex. See **pgr_analyzeGraph**.
chk:
integer Indicator that the vertex might have a problem. See **pgr_analyzeGraph**.
ein:
integer Number of vertices in the edge_table that reference this vertex as incoming. See **pgr_analyzeOneWay**.
eout:
integer Number of vertices in the edge_table that reference this vertex as outgoing. See **pgr_analyzeOneWay**.
the_geom:
geometry Point geometry of the vertex.
Example 1:
The simplest way to use pgr_createVerticesTable

```
SELECT  pgr_createVerticesTable('edge_table');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                        Edges processed: 18
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Additional Examples

Example 2:
When the arguments are given in the order described in the parameters:

```
SELECT  pgr_createVerticesTable('edge_table', 'the_geom', 'source', 'target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  -----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

We get the same result as the simplest way to use the function.

Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the column source column `source` of the table `mytable` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the source column.

```
SELECT  pgr_createVerticesTable('edge_table', 'source', 'the_geom', 'target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','source','the_geom','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  ----> PGR ERROR in pgr_createVerticesTable: Wrong type of Column source: the_geom
HINT:    ----> Expected type of the_geom is integer, smallint or bigint but USER-DEFINED was found
NOTICE:  Unexpected error raise_exception
 pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

When using the named notation

Example 3:
The order of the parameters do not matter:

```
SELECT  pgr_createVerticesTable('edge_table', the_geom:='the_geom', source:='source', target:='target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  -----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 4:
Using a different ordering

```
SELECT  pgr_createVerticesTable('edge_table', source:='source', target:='target', the_geom:='the_geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  -----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 5:
Parameters defined with a default value can be omitted, as long as the value matches the default:

```
SELECT  pgr_createVerticesTable('edge_table',source:='source');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Example 6:

Selecting rows based on the id.

```
SELECT  pgr_createVerticesTable('edge_table',rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','id < 10')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  9 VERTICES
NOTICE:                                 FOR   10  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 10
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 7:

Selecting the rows where the geometry is near the geometry of row with `id` =5 .

```
SELECT  pgr_createVerticesTable('edge_table',
   rows_where:='the_geom && (select st_buffer(the_geom,0.5) FROM edge_table WHERE id=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','the_geom && (select st_buffer(the_geom,0.5) FROM edge_table WHERE id=5)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  9 VERTICES
NOTICE:                                 FOR   9  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 9
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 8:

Selecting the rows where the geometry is near the geometry of the row with `gid` =100 of the table `othertable`.

```
DROP TABLE IF EXISTS otherTable;
NOTICE:  table "othertable" does not exist, skipping
DROP TABLE
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT 1
SELECT  pgr_createVerticesTable('edge_table',
   rows_where:='the_geom && (select st_buffer(other_geom,0.5) FROM otherTable WHERE gid=100)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','the_geom && (select st_buffer(other_geom,0.5) FROM otherTable WHERE gid=100)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  10 VERTICES
NOTICE:                                 FOR   12  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 12
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Usage when the edge table's columns DO NOT MATCH the default values:

Using the following table

```
DROP TABLE IF EXISTS mytable;
NOTICE:  table "mytable" does not exist, skipping
DROP TABLE
CREATE TABLE mytable AS (SELECT id AS gid, the_geom AS mygeom, source AS src ,target AS tgt FROM edge_table) ;
SELECT 18
```

Using positional notation:

Example 9:
The arguments need to be given in the order described in the parameters:

```
SELECT  pgr_createVerticesTable('mytable', 'mygeom', 'src', 'tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.mytable_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE:  -----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the columnsrc of the table mytable is passed to the function as the geometry column, and the geometry columnmygeom is passed to the function as the source column.

```
SELECT  pgr_createVerticesTable('mytable', 'src', 'mygeom', 'tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','src','mygeom','tgt','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  ----> PGR ERROR in pgr_createVerticesTable: Wrong type of Column source: mygeom
HINT:    ----> Expected type of mygeom is integer, smallint or bigint but USER-DEFINED was found
NOTICE:  Unexpected error raise_exception
 pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

When using the named notation

Example 10:
The order of the parameters do not matter:

```
SELECT  pgr_createVerticesTable('mytable',the_geom:='mygeom',source:='src',target:='tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.mytable_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE:  -----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 11:
Using a different ordering

In this scenario omitting a parameter would create an error because the default values for the column names do not match the column names of the table.

```
SELECT  pgr_createVerticesTable(
   'mytable', source:='src', target:='tgt',
   the_geom:='mygeom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.mytable_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                 FOR   18  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 18
NOTICE:  Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Example 12:

Selecting rows based on the gid. (positional notation)

```
SELECT  pgr_createVerticesTable(
   'mytable', 'mygeom', 'src', 'tgt',
   rows_where:='gid < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','gid < 10')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.mytable_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  9 VERTICES
NOTICE:                                 FOR   10  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 10
NOTICE:  Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 13:

Selecting rows based on the gid. (named notation)

```
SELECT  pgr_createVerticesTable(
   'mytable', source:='src', target:='tgt', the_geom:='mygeom',
   rows_where:='gid < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','gid < 10')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.mytable_vertices_pgr, please wait...
NOTICE:   ----->   VERTICES TABLE CREATED WITH  9 VERTICES
NOTICE:                                 FOR   10  EDGES
NOTICE:   Edges with NULL geometry,source or target: 0
NOTICE:                         Edges processed: 10
NOTICE:  Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE:  ------------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 14:

Selecting the rows where the geometry is near the geometry of row with `gid` = 5.

```
SELECT  pgr_createVerticesTable(
   'mytable', 'mygeom', 'src', 'tgt',
   rows_where := 'the_geom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHERE gid=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','the_geom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHERE gid=5)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Got column "the_geom" does not exist
NOTICE:  ERROR: Condition is not correct, please execute the following query to test your condition
NOTICE:  select * from public.mytable WHERE true AND (the_geom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHERE gid=5)) limit 1
 pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

Example 15:

TBD

```
SELECT  pgr_createVerticesTable(
    'mytable', source:='src', target:='tgt', the_geom:='mygeom',
    rows_where:='mygeom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHERE id=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','mygeom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHERE id=5)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Got column "id" does not exist
NOTICE:  ERROR: Condition is not correct, please execute the following query to test your condition
NOTICE:  select * from public.mytable WHERE true AND (mygeom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHERE id=5)) limit 1
 pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

Example 16:

Selecting the rows where the geometry is near the geometry of the row with `gid` =100 of the table `othertable`.

```
DROP TABLE IF EXISTS otherTable;
DROP TABLE
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT 1
```

```
SELECT pgr_createVerticesTable(
    'mytable', 'mygeom', 'src', 'tgt',
    rows_where:='the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Got column "the_geom" does not exist
NOTICE:  ERROR: Condition is not correct, please execute the following query to test your condition
NOTICE:  select * from public.mytable WHERE true AND (the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)) limit 1
 pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

Example 17:

TBD

```
SELECT  pgr_createVerticesTable(
    'mytable',source:='src',target:='tgt',the_geom:='mygeom',
    rows_where:='the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('mytable','mygeom','src','tgt','the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Got column "the_geom" does not exist
NOTICE:  ERROR: Condition is not correct, please execute the following query to test your condition
NOTICE:  select * from public.mytable WHERE true AND (the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)) limit 1
 pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

The example uses the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr_createTopology** <pgr_create_topology>` to create a topology based on the geometry.
- **pgr_analyzeGraph** to analyze the edges and vertices of the edge table.
- **pgr_analyzeOneWay** to analyze directionality of the edges.

Indices and tables

- **Index**
- **Search Page**

pgr_analyzeGraph

`pgr_analyzeGraph` — Analyzes the network topology.

Availability

- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The function returns:

- OK after the analysis has finished.
- FAIL when the analysis was not completed due to an error.

```
varchar pgr_analyzeGraph(text edge_table, double precision tolerance,
                text the_geom:='the_geom', text id:='id',
                text source:='source',text target:='target',text rows_where:='true')
```

Prerequisites

The edge table to be analyzed must contain a source column and a target column filled with id's of the vertices of the segments and the corresponding vertices table <edge_table>_vertices_pgr that stores the vertices information.

- Use **pgr_createVerticesTable** to create the vertices table.
- Use **pgr_createTopology** to create the topology and the vertices table.

Parameters

The analyze graph function accepts the following parameters:

edge_table:
text Network table name. (may contain the schema name as well)
tolerance:
float8 Snapping tolerance of disconnected edges. (in projection unit)
the_geom:
text Geometry column name of the network table. Default value is the_geom.
id:
text Primary key column name of the network table. Default value is id.
source:
text Source column name of the network table. Default value is source.
target:
text Target column name of the network table. Default value is target.
rows_where:
text Condition to select a subset or rows. Default value is true to indicate all rows.

The function returns:

- OK after the analysis has finished.
  - Uses the vertices table: <edge_table>_vertices_pgr.
  - Fills completely the cnt and chk columns of the vertices table.
  - Returns the analysis of the section of the network defined by rows_where
- FAIL when the analysis was not completed due to an error.
  - The vertices table is not found.
  - A required column of the Network table is not found or is not of the appropriate type.
  - The condition is not well formed.
  - The names of source , target or id are the same.
  - The SRID of the geometry could not be determined.

The Vertices Table

The vertices table can be created with **pgr_createVerticesTable** or **pgr_createTopology**

The structure of the vertices table is:

id:
bigint Identifier of the vertex.
cnt:
integer Number of vertices in the edge_table that reference this vertex.
chk:
integer Indicator that the vertex might have a problem.
ein:
integer Number of vertices in the edge_table that reference this vertex as incoming. See **pgr_analyzeOneWay**.
eout:
integer Number of vertices in the edge_table that reference this vertex as outgoing. See **pgr_analyzeOneWay**.
the_geom:

`geometry` Point geometry of the vertex.

Usage when the edge table's columns MATCH the default values:

The simplest way to use pgr_analyzeGraph is:

```
SELECT  pgr_createTopology('edge_table',0.001, clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := t)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ----------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT  pgr_analyzeGraph('edge_table',0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:              Intersections detected: 1
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
-------------------
 OK
(1 row)
```

When the arguments are given in the order described in the parameters:

```
SELECT  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:              Intersections detected: 1
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
-------------------
 OK
(1 row)
```

We get the same result as the simplest way to use the function.

Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the column `id` of the table `mytable` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the id column.

```
SELECT  pgr_analyzeGraph('edge_table',0.001,'id','the_geom','source','target');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'id','the_geom','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Got function st_srid(bigint) does not exist
NOTICE:  ERROR: something went wrong when checking for SRID of id in table public.edge_table
 pgr_analyzegraph
------------------
 FAIL
(1 row)
```

When using the named notation

The order of the parameters do not matter:

```
SELECT  pgr_analyzeGraph('edge_table',0.001,the_geom:='the_geom',id:='id',source:='source',target:='target');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                  Isolated segments: 2
NOTICE:                        Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:              Intersections detected: 1
NOTICE:                  Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('edge_table',0.001,source:='source',id:='id',target:='target',the_geom:='the_geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                  Isolated segments: 2
NOTICE:                        Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:              Intersections detected: 1
NOTICE:                  Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Parameters defined with a default value can be omitted, as long as the value matches the default:

```
SELECT  pgr_analyzeGraph('edge_table',0.001,source:='source');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                  Isolated segments: 2
NOTICE:                        Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:              Intersections detected: 1
NOTICE:                  Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Selecting rows based on the id. Displays the analysis a the section of the network.

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 10')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                           Dead ends: 4
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:                Intersections detected: 0
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of row with `id` =5 .

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(the_geom,0.05) FROM edge_table WHERE id=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','the_geom && (SELECT st_buffer(the_geom,0.05) FROM edge_table WHERE id=5)')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                           Dead ends: 5
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:                Intersections detected: 0
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of the row with `gid` =100 of the table `othertable`.

```
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT 1
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE gid=100)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','the_geom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE gid=100)')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 10
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:                Intersections detected: 1
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Usage when the edge table's columns DO NOT MATCH the default values:

For the following table

```
CREATE TABLE mytable AS (SELECT id AS gid, source AS src ,target AS tgt , the_geom AS mygeom FROM edge_table);
SELECT 18
SELECT pgr_createTopology('mytable',0.001,'mygeom','gid','src','tgt', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', rows_where := 'true', clean := t)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

Using positional notation:

The arguments need to be given in the order described in the parameters:

```
SELECT  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:                Intersections detected: 1
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the columngid of the table mytable is passed to the function as the geometry column, and the geometry columnmygeom is passed to the function as the id column.

```
SELECT  pgr_analyzeGraph('mytable',0.0001,'gid','mygeom','src','tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.0001,'gid','mygeom','src','tgt','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Got function st_srid(bigint) does not exist
NOTICE:  ERROR: something went wrong when checking for SRID of gid in table public.mytable
 pgr_analyzegraph
------------------
 FAIL
(1 row)
```

When using the named notation

The order of the parameters do not matter:

```
SELECT  pgr_analyzeGraph('mytable',0.001,the_geom:='mygeom',id:='gid',source:='src',target:='tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                   Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                   Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                   Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                   Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

In this scenario omitting a parameter would create an error because the default values for the column names do not match the column names of the table.

Selecting rows using rows_where parameter

Selecting rows based on the id.

```
SELECT  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',rows_where:='gid < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','gid < 10')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                   Isolated segments: 0
NOTICE:                           Dead ends: 4
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                   Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom',rows_where:='gid < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','gid < 10')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                           Dead ends: 4
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Selecting the rows WHERE the geometry is near the geometry of row with `id` =5 .

```
SELECT  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',
   rows_where:='mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 1
NOTICE:                           Dead ends: 5
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 1
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom',
   rows_where:='mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 1
NOTICE:                           Dead ends: 5
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 1
NOTICE:                    Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Selecting the rows WHERE the geometry is near the place='myhouse' of the table `othertable`. (note the use of quote_literal)

```
DROP TABLE IF EXISTS otherTable;
DROP TABLE
CREATE TABLE otherTable AS  (SELECT 'myhouse'::text AS place, st_point(2.5,2.5) AS other_geom) ;
SELECT 1
SELECT  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',
   rows_where:='mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='||quote_literal('myhouse')||')');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='myhouse')')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 10
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom',
   rows_where:='mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='||quote_literal('myhouse')||')');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='myhouse')')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 10
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Additional Examples

```
SELECT  pgr_createTopology('edge_table',0.001, clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := t)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT pgr_analyzeGraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 10')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:               ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                     Isolated segments: 0
NOTICE:                           Dead ends: 4
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id >= 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id >= 10')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:               ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                     Isolated segments: 2
NOTICE:                           Dead ends: 8
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 17');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 17')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:               ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                     Isolated segments: 0
NOTICE:                           Dead ends: 3
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

```
SELECT pgr_createTopology('edge_table', 0.001,rows_where:='id <17', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'id <17', clean := t)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  16 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT pgr_analyzeGraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 3
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

The examples use the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr_analyzeOneWay** to analyze directionality of the edges.
- **pgr_createVerticesTable** to reconstruct the vertices table based on the source and target information.
- **pgr_nodeNetwork** to create nodes to a not noded edge table.

Indices and tables

- **Index**
- **Search Page**

pgr_analyzeOneWay

`pgr_analyzeOneWay` — Analyzes oneway Sstreets and identifies flipped segments.

This function analyzes oneway streets in a graph and identifies any flipped segments.

Availability

- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The analyses of one way segments is pretty simple but can be a powerful tools to identifying some the potential problems created by setting the direction of a segment the wrong way. A node is a *source* if it has edges the exit from that node and no edges enter that node. Conversely, a node is a *sink* if all edges enter the node but none exit that node. For a *source* type node it is logically impossible to exist because no vehicle can exit the node if no vehicle and enter the node. Likewise, if you had a *sink* node you would have an infinite number of vehicle piling up on this node because you can enter it but not leave it.

So why do we care if the are not feasible? Well if the direction of an edge was reversed by mistake we could generate exactly these conditions. Think about a divided highway and on the north bound lane one segment got entered wrong or maybe a sequence of multiple segments got entered wrong or maybe this happened on a round-about. The result would be potentially a *source* and/or a *sink* node.

So by counting the number of edges entering and exiting each node we can identify both *source* and *sink* nodes so that you can look at those areas of your network to make repairs and/or report the problem back to your data vendor.

Prerequisites

The edge table to be analyzed must contain a source column and a target column filled with id's of the vertices of the segments and the corresponding vertices table <edge_table>_vertices_pgr that stores the vertices information.

- Use **pgr_createVerticesTable** to create the vertices table.
- Use **pgr_createTopology** to create the topology and the vertices table.

Signatures

```
text pgr_analyzeOneWay(geom_table text,
                       text[] s_in_rules, text[] s_out_rules,
                       text[] t_in_rules, text[] t_out_rules,
                       text oneway='oneway', text source='source', text target='target',
                       boolean two_way_if_null=true);
```

Parameters

edge_table:
`text` Network table name. (may contain the schema name as well)
s_in_rules:
`text[]` source node **in** rules
s_out_rules:
`text[]` source node **out** rules
t_in_rules:
`text[]` target node **in** rules
t_out_rules:
`text[]` target node **out** rules
oneway:
`text` oneway column name name of the network table. Default value is `oneway`.
source:
`text` Source column name of the network table. Default value is `source`.
target:
`text` Target column name of the network table. Default value is `target`.
two_way_if_null:
`boolean` flag to treat oneway NULL values as bi-directional. Default value is `true`.

Note

It is strongly recommended to use the named notation. See **pgr_createVerticesTable** or **pgr_createTopology** for examples.

The function returns:

- `OK` after the analysis has finished.
  - Uses the vertices table: <edge_table>_vertices_pgr.
  - Fills completely the `ein` and `eout` columns of the vertices table.
- `FAIL` when the analysis was not completed due to an error.
  - The vertices table is not found.
  - A required column of the Network table is not found or is not of the appropriate type.
  - The names of source , target or oneway are the same.

The rules are defined as an array of text strings that if match the `oneway` value would be counted as `true` for the source or target **in** or **out** condition.

The Vertices Table

The vertices table can be created with **pgr_createVerticesTable** or **pgr_createTopology**

The structure of the vertices table is:

id:
`bigint` Identifier of the vertex.
cnt:
`integer` Number of vertices in the edge_table that reference this vertex. See **pgr_analyzeGgraph**.
chk:
`integer` Indicator that the vertex might have a problem. See **pgr_analyzeGraph**.
ein:
`integer` Number of vertices in the edge_table that reference this vertex as incoming.
eout:
`integer` Number of vertices in the edge_table that reference this vertex as outgoing.
the_geom:
`geometry` Point geometry of the vertex.

Additional Examples

```
SELECT pgr_analyzeOneWay('edge_table',
        ARRAY['', 'B', 'TF'],
        ARRAY['', 'B', 'FT'],
        ARRAY['', 'B', 'FT'],
        ARRAY['', 'B', 'TF'],
        oneway:='dir');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeOneway('edge_table',{"",B,TF},{"",B,FT},{"",B,FT},{"",B,TF},'dir','source','target',t)
NOTICE:  Analyzing graph for one way street errors.
NOTICE:  Analysis 25% complete ...
NOTICE:  Analysis 50% complete ...
NOTICE:  Analysis 75% complete ...
NOTICE:  Analysis 100% complete ...
NOTICE:  Found 0 potential problems in directionality
 pgr_analyzeoneway
-------------------
 OK
(1 row)
```

The queries use the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **Graph Analytics** for an overview of the analysis of a graph.
- **pgr_analyzeGraph** to analyze the edges and vertices of the edge table.
- **pgr_createVerticesTable** to reconstruct the vertices table based on the source and target information.

Indices and tables

- **Index**
- **Search Page**

pgr_nodeNetwork

`pgr_nodeNetwork` - Nodes an network edge table.

Author:
Nicolas Ribot
Copyright:
Nicolas Ribot, The source code is released under the MIT-X license.

The function reads edges from a not “noded” network table and writes the “noded” edges into a new table.

```
pgr_nodenetwork(edge_table, tolerance, id, text the_geom, table_ending, rows_where, outall)
RETURNS TEXT
```

Availability

- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The main characteristics are:

A common problem associated with bringing GIS data into pgRouting is the fact that the data is often not “noded” correctly. This will create invalid topologies, which will result in routes that are incorrect.

What we mean by “noded” is that at every intersection in the road network all the edges will be broken into separate road segments. There are cases like an over-pass and under-pass intersection where you can not traverse from the over-pass to the under-pass, but this function does not have the ability to detect and accommodate those situations.

This function reads the `edge_table` table, that has a primary key column `id` and geometry column named `the_geom` and intersect all the segments in it against all the other segments and then creates a table `edge_table_noded`. It uses the `tolerance` for deciding that multiple nodes within the tolerance are considered the same node.

Parameters

edge_table:
`text` Network table name. (may contain the schema name as well)

tolerance:
`float8` tolerance for coincident points (in projection unit)dd
id:
`text` Primary key column name of the network table. Default value is `id`.
the_geom:
`text` Geometry column name of the network table. Default value is `the_geom`.
table_ending:
`text` Suffix for the new table's. Default value is `noded`.

The output table will have for `edge_table_noded`

id:
`bigint` Unique identifier for the table
old_id:
`bigint` Identifier of the edge in original table
sub_id:
`integer` Segment number of the original edge
source:
`integer` Empty source column to be used with **pgr_createTopology** function
target:
`integer` Empty target column to be used with **pgr_createTopology** function
the geom:
`geometry` Geometry column of the noded network

Examples

Let's create the topology for the data in **Sample Data**

```
SELECT pgr_createTopology('edge_table', 0.001, clean := TRUE);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := t)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

Now we can analyze the network.

```
SELECT pgr_analyzegraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                           Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

The analysis tell us that the network has a gap and an intersection. We try to fix the problem using:

```
SELECT pgr_nodeNetwork('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  id: id
NOTICE:  the_geom: the_geom
NOTICE:  table_ending: noded
NOTICE:  rows_where:
NOTICE:  outall: f
NOTICE:  pgr_nodeNetwork('edge_table', 0.001, 'id', 'the_geom', 'noded', '',  f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Processing, please wait .....
NOTICE:    Split Edges: 3
NOTICE:   Untouched Edges: 15
NOTICE:       Total original Edges: 18
NOTICE:   Edges generated: 6
NOTICE:   Untouched Edges: 15
NOTICE:         Total New segments: 21
NOTICE:   New Table: public.edge_table_noded
NOTICE:  ---------------------------------
 pgr_nodenetwork
-----------------
 OK
(1 row)
```

Inspecting the generated table, we can see that edges 13,14 and 18 has been segmented

```
SELECT old_id, sub_id FROM edge_table_noded ORDER BY old_id, sub_id;
 old_id | sub_id
--------+--------
      1 |      1
      2 |      1
      3 |      1
      4 |      1
      5 |      1
      6 |      1
      7 |      1
      8 |      1
      9 |      1
     10 |      1
     11 |      1
     12 |      1
     13 |      1
     13 |      2
     14 |      1
     14 |      2
     15 |      1
     16 |      1
     17 |      1
     18 |      1
     18 |      2
(21 rows)
```

We can create the topology of the new network

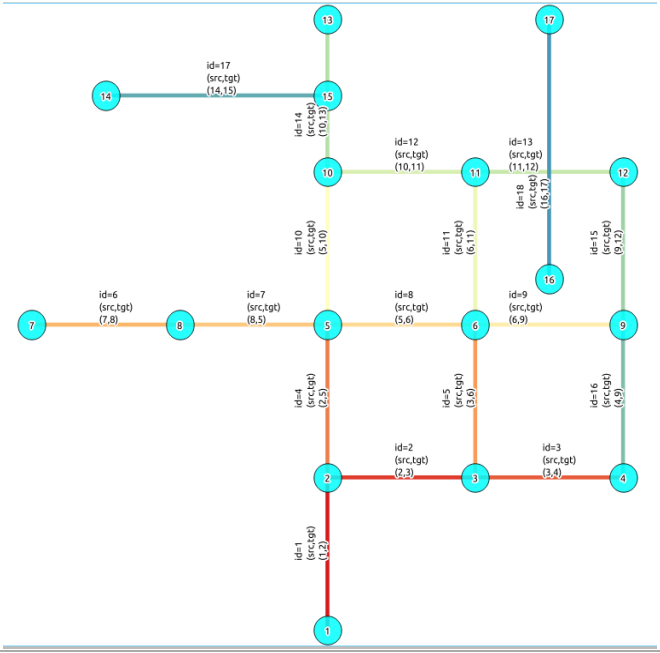
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  21 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table_noded is: public.edge_table_noded_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

Now let’s analyze the new topology

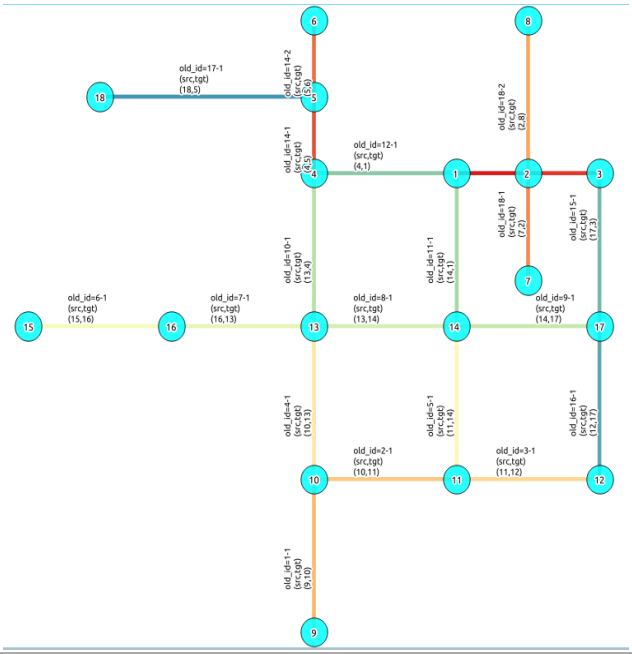
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                           Dead ends: 6
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Images

Before Image



After Image



Comparing the results

Comparing with the Analysis in the original edge_table, we see that.

	Before	**After**
Table name	edge_table	edge_table_noded
Fields	All original fields	Has only basic fields to do a topology analysis
Dead ends	• Edges with 1 dead end: 1,6,24 • Edges with 2 dead ends 17,18 Edge 17's right node is a dead end because there is no other edge sharing that same node. (cnt=1)	Edges with 1 dead end: 1-1 ,6-1,14-2, 18-1 17-1 18-2
Isolated segments	two isolated segments: 17 and 18 both they have 2 dead ends	No Isolated segments • Edge 17 now shares a node with edges 14-1 and 14-2 • Edges 18-1 and 18-2 share a node with edges 13-1 and 13-2
Gaps	There is a gap between edge 17 and 14 because edge 14 is near to the right node of edge 17	Edge 14 was segmented Now edges: 14-1 14-2 17 share the same node The tolerance value was taken in account
Intersections	Edges 13 and 18 were intersecting	Edges were segmented, So, now in the interection's point there is a node and the following edges share it: 13-1 13-2 18-1 18-2

Now, we are going to include the segments 13-1, 13-2 14-1, 14-2 ,18-1 and 18-2 into our edge-table, copying the data for dir,cost,and reverse cost with tho following steps:

- Add a column old_id into edge_table, this column is going to keep track the id of the original edge

- Insert only the segmented edges, that is, the ones whose max(sub_id) >1

```
alter table edge_table drop column if exists old_id;
NOTICE:  column "old_id" of relation "edge_table" does not exist, skipping
ALTER TABLE
alter table edge_table add column old_id integer;
ALTER TABLE
insert into edge_table (old_id, dir, cost, reverse_cost, the_geom)
  (with
    segmented as (select old_id,count(*) as i from edge_table_noded group by old_id)
  select  segments.old_id, dir, cost, reverse_cost, segments.the_geom
    from edge_table as edges join edge_table_noded as segments on (edges.id = segments.old_id)
    where edges.id in (select old_id from segmented where i>1) );
INSERT 0 6
```

We recreate the topology:

```
SELECT pgr_createTopology('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  ------------> TOPOLOGY CREATED FOR  6 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

To get the same analysis results as the topology of edge_table_noded, we do the following query:

```
SELECT pgr_analyzegraph('edge_table', 0.001, rows_where:='id not in (select old_id from edge_table where old_id is not null)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id not in (select old_id from edge_table where old_id is not null)')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                  Isolated segments: 0
NOTICE:                        Dead ends: 6
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:              Intersections detected: 0
NOTICE:                  Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

To get the same analysis results as the original edge_table, we do the following query:

```
SELECT pgr_analyzegraph('edge_table', 0.001, rows_where:='old_id is null');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','old_id is null')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                  Isolated segments: 2
NOTICE:                        Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:              Intersections detected: 1
NOTICE:                  Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

Or we can analyze everything because, maybe edge 18 is an overpass, edge 14 is an under pass and there is also a street level juction, and the same happens with edges 17 and 13.

```
SELECT pgr_analyzegraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, please wait ...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 3
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 5
NOTICE:                     Ring geometries: 0
 pgr_analyzegraph
------------------
 OK
(1 row)
```

See Also

Topology - Family of Functions for an overview of a topology for routing algorithms. **pgr_analyzeOneWay** to analyze directionality of the edges. **pgr_createTopology** to create a topology based on the geometry. **pgr_analyzeGraph** to analyze the edges and vertices of the edge table.

Indices and tables

- **Index**
- **Search Page**

See Also

Indices and tables

- **Index**
- **Search Page**

Traveling Sales Person - Family of functions

- **pgr_TSP** - When input is given as matrix cell information.
- **pgr_TSPeuclidean** - When input are coordinates.

pgr_TSP

- `pgr_TSP` - Using *Simulated Annealing* approximation algorithm

Availability: 2.0.0

- Version 2.3.0
  - Signature change
    - Old signature no longer supported
- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question:

Given a list of cities and the distances between each pair of cities, which is the shortest possible route that visits each city exactly once and returns to the origin city?

See **Simulated Annealing Algorithm** for a complete description of this implementation

Signatures

Summary

```
pgr_TSP(Matrix SQL,
    [start_id], [end_id],
    [max_processing_time],
    [tries_per_temperature], [max_changes_per_temperature], [max_consecutive_non_changes],
    [initial_temperature], [final_temperature], [cooling_factor],
    [randomize])
RETURNS SETOF (seq, node, cost, agg_cost)
```

Example:

Not having a random execution

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_dijkstraCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 14),
        directed := false)
    $$,
    randomize := false);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    1 |    3 |        0
   2 |    4 |    1 |        3
   3 |    9 |    1 |        4
   4 |   12 |    1 |        5
   5 |   11 |    2 |        6
   6 |   13 |    1 |        8
   7 |   10 |    1 |        9
   8 |    5 |    2 |       10
   9 |    7 |    1 |       12
  10 |    8 |    2 |       13
  11 |    6 |    1 |       15
  12 |    3 |    1 |       16
  13 |    2 |    1 |       17
  14 |    1 |    0 |       18
(14 rows)
```

Parameters

Parameter	Description
Matrix SQL	an SQL query, described in the **Inner query**

Optional Parameters

Parameter	Type	Default	Description
start_vid	BIGINT	*0*	The greedy part of the implementation will use this identifier.
end_vid	BIGINT	*0*	Last visiting vertex before returning to start_vid.
max_processing_time	FLOAT	*+infinity*	Stop the annealing processing when the value is reached.
tries_per_temperature	INTEGER	*500*	Maximum number of times a neighbor(s) is searched in each temperature.
max_changes_per_temperature	INTEGER	*60*	Maximum number of times the solution is changed in each temperature.
max_consecutive_non_changes	INTEGER	*100*	Maximum number of consecutive times the solution is not changed in each temperature.
initial_temperature	FLOAT	*100*	Starting temperature.
final_temperature	FLOAT	*0.1*	Ending temperature.
cooling_factor	FLOAT	*0.9*	Value between between 0 and 1 (not including) used to calculate the next temperature.
randomize	BOOLEAN	*true*	Choose the random seed - true: Use current time as seed - false: Use *1* as seed. Using this value will get the same results with the same data in each execution.

Inner query

Matrix SQL: an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex.
end_vid	BIGINT	Identifier of the ending vertex.
agg_cost	FLOAT	Cost for going from start_vid to end_vid

Can be Used with **Cost Matrix - Category** functions with *directed := false*.

If using *directed := true*, the resulting non symmetric matrix must be converted to symmetric by fixing the non symmetric values according to your application needs.

Result Columns

Returns SET OF `(seq, node, cost, agg_cost)`

Column	Type	Description
seq	`INTEGER`	Row sequence.
node	`BIGINT`	Identifier of the node/coordinate/point.
cost	`FLOAT`	Cost to traverse from the current `node` to the next `node` in the path sequence. • `0` for the last row in the path sequence.
agg_cost	`FLOAT`	Aggregate cost from the `node` at `seq = 1` to the current node. • `0` for the first row in the path sequence.

Additional Examples

Example:
Start from vertex 7

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_dijkstraCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 14),
        directed := false
    )
    $$,
    start_id := 7,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    7 |    1 |        0
   2 |    8 |    1 |        1
   3 |    5 |    1 |        2
   4 |    2 |    1 |        3
   5 |    1 |    2 |        4
   6 |    3 |    1 |        6
   7 |    4 |    1 |        7
   8 |    9 |    1 |        8
   9 |   12 |    1 |        9
  10 |   11 |    1 |       10
  11 |   10 |    1 |       11
  12 |   13 |    3 |       12
  13 |    6 |    3 |       15
  14 |    7 |    0 |       18
(14 rows)
```

Example:
Using with points of interest.

To generate a symmetric matrix:

- the **side** information of pointsOfInterset is ignored by not including it in the query
- and **directed := false**

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_withPointsCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
        'SELECT pid, edge_id, fraction from pointsOfInterest',
        array[-1, 3, 5, 6, -6], directed := false)
    $$,
    start_id := 5,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    5 |    1 |        0
   2 |    6 |    1 |        1
   3 |    3 |  1.6 |        2
   4 |   -1 |  1.3 |      3.6
   5 |   -6 |  0.3 |      4.9
   6 |    5 |    0 |      5.2
(6 rows)
```

The queries use the **Sample Data** network.

See Also

- **Traveling Sales Person - Family of functions**
- **Simulated annaeling algorithm for beginners**
- **Wikipedia: Traveling Salesman Problem**
- **Wikipedia: Simulated annealing**

Indices and tables

- **Index**
- **Search Page**

pgr_TSPeuclidean

`pgr_TSPeuclidean` - Using *Simulated Annealing* approximation algorithm

Availability

- Version 3.0.0
  - Name change from pgr_eucledianTSP
- Version 2.3.0
  - New **Official** function

Support

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3**

Description

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question:

Given a list of cities and the distances between each pair of cities, which is the shortest possible route that visits each city exactly once and returns to the origin city?

See **Simulated Annealing Algorithm** for a complete description of this implementation

Signatures

Summary

```
pgr_TSPeuclidean(Coordinates SQL,
    [start_id], [end_id],
    [max_processing_time],
    [tries_per_temperature], [max_changes_per_temperature], [max_consecutive_non_changes],
    [initial_temperature], [final_temperature], [cooling_factor],
    [randomize])
 RETURNS SETOF (seq, node, cost, agg_cost)
```

Example:
Not having a random execution

```
 SELECT * FROM pgr_TSPeuclidean(
    $$
    SELECT id, st_X(the_geom) AS x, st_Y(the_geom)AS y  FROM edge_table_vertices_pgr
    $$,
    randomize := false);
 seq | node |      cost      |   agg_cost
-----+------+----------------+---------------
   1 |    1 |  1.41421356237 |            0
   2 |    3 |              1 | 1.41421356237
   3 |    4 |              1 | 2.41421356237
   4 |    9 |              1 | 3.41421356237
   5 |    6 | 0.583095189485 | 4.41421356237
   6 |   16 | 0.860232526704 | 4.99730875186
   7 |   12 |  1.11803398875 | 5.85754127856
   8 |   17 |  1.11803398875 | 6.97557526731
   9 |   11 |              1 | 8.09360925606
  10 |   10 |            0.5 | 9.09360925606
  11 |   15 |            0.5 | 9.59360925606
  12 |   13 |  1.58113883008 | 10.0936092561
  13 |   14 |  1.58113883008 | 11.6747480861
  14 |    7 |              1 | 13.2558869162
  15 |    8 |              1 | 14.2558869162
  16 |    5 |              1 | 15.2558869162
  17 |    2 |              1 | 16.2558869162
  18 |    1 |              0 | 17.2558869162
(18 rows)
```

Parameters

Parameter	Description
Coordinates SQL	an SQL query, described in the **Inner query**

Optional Parameters

Parameter	Type	Default	Description
start_vid	BIGINT	*0*	The greedy part of the implementation will use this identifier.
end_vid	BIGINT	*0*	Last visiting vertex before returning to start_vid.
max_processing_time	FLOAT	*+infinity*	Stop the annealing processing when the value is reached.
tries_per_temperature	INTEGER	*500*	Maximum number of times a neighbor(s) is searched in each temperature.
max_changes_per_temperature	INTEGER	*60*	Maximum number of times the solution is changed in each temperature.
max_consecutive_non_changes	INTEGER	*100*	Maximum number of consecutive times the solution is not changed in each temperature.
initial_temperature	FLOAT	*100*	Starting temperature.
final_temperature	FLOAT	*0.1*	Ending temperature.
cooling_factor	FLOAT	*0.9*	Value between between 0 and 1 (not including) used to calculate the next temperature.
randomize	BOOLEAN	*true*	Choose the random seed • true: Use current time as seed • false: Use *1* as seed. Using this value will get the same results with the same data in each execution.

Inner query

Coordinates SQL: an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
id	BIGINT	(optional) Identifier of the coordinate. • When missing the coordinates will receive an **id** starting from 1, in the order given.
x	FLOAT	X value of the coordinate.
y	FLOAT	Y value of the coordinate.

Result Columns

Returns SET OF `(seq, node, cost, agg_cost)`

Column	Type	Description
seq	INTEGER	Row sequence.
node	BIGINT	Identifier of the node/coordinate/point.
cost	FLOAT	Cost to traverse from the current `node` to the next `node` in the path sequence. • `0` for the last row in the path sequence.
agg_cost	FLOAT	Aggregate cost from the `node` at `seq = 1` to the current node. • `0` for the first row in the path sequence.

Additional Examples

Example:
Try 3 times per temperature with cooling factor of 0.5, not having a random execution

```
SELECT* from pgr_TSPeuclidean(
    $$
    SELECT id, st_X(the_geom) AS x, st_Y(the_geom) AS y FROM edge_table_vertices_pgr
    $$,
    tries_per_temperature := 3,
    cooling_factor := 0.5,
    randomize := false);
 seq | node |      cost      |   agg_cost
-----+------+----------------+---------------
   1 |    1 |  1.41421356237 |             0
   2 |    3 |              1 | 1.41421356237
   3 |    4 |              1 | 2.41421356237
   4 |    9 | 0.583095189485 | 3.41421356237
   5 |   16 | 0.583095189485 | 3.99730875186
   6 |    6 |              1 | 4.58040394134
   7 |    5 |              1 | 5.58040394134
   8 |    8 |              1 | 6.58040394134
   9 |    7 |  1.58113883008 | 7.58040394134
  10 |   14 |            1.5 | 9.16154277143
  11 |   15 |            0.5 | 10.6615427714
  12 |   13 |            1.5 | 11.1615427714
  13 |   17 |  1.11803398875 | 12.6615427714
  14 |   12 |              1 | 13.7795767602
  15 |   11 |              1 | 14.7795767602
  16 |   10 |              2 | 15.7795767602
  17 |    2 |              1 | 17.7795767602
  18 |    1 |              0 | 18.7795767602
(18 rows)
```

Example:

Skipping the Simulated Annealing & showing some process information

```
SET client_min_messages TO DEBUG1;
SET
SELECT* from pgr_TSPeuclidean(
    $$
    SELECT id, st_X(the_geom) AS x, st_Y(the_geom) AS y FROM edge_table_vertices_pgr
    $$,
    tries_per_temperature := 0,
    randomize := false);
DEBUG:  Processing Information
Initializing tsp class ---> tsp.greedyInitial ---> tsp.annealing ---> OK

Cycle(100)  total changes =0 0 were because  delta energy < 0
Total swaps: 3
Total slides: 0
Total reverses: 0
Times best tour changed: 4
Best cost reached = 18.7796
 seq | node |      cost      |   agg_cost
-----+------+----------------+---------------
   1 |    1 |  1.41421356237 |             0
   2 |    3 |              1 | 1.41421356237
   3 |    4 |              1 | 2.41421356237
   4 |    9 | 0.583095189485 | 3.41421356237
   5 |   16 | 0.583095189485 | 3.99730875186
   6 |    6 |              1 | 4.58040394134
   7 |    5 |              1 | 5.58040394134
   8 |    8 |              1 | 6.58040394134
   9 |    7 |  1.58113883008 | 7.58040394134
  10 |   14 |            1.5 | 9.16154277143
  11 |   15 |            0.5 | 10.6615427714
  12 |   13 |            1.5 | 11.1615427714
  13 |   17 |  1.11803398875 | 12.6615427714
  14 |   12 |              1 | 13.7795767602
  15 |   11 |              1 | 14.7795767602
  16 |   10 |              2 | 15.7795767602
  17 |    2 |              1 | 17.7795767602
  18 |    1 |              0 | 18.7795767602
(18 rows)
```

The queries use the **Sample Data** network.

See Also

- **Traveling Sales Person - Family of functions**
- **Simulated annaeling algorithm for beginners**
- **Wikipedia: Traveling Salesman Problem**
- **Wikipedia: Simulated annealing**

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3**

Table of Contents

General Information

Problem Definition

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question:

Given a list of cities and the distances between each pair of cities, which is the shortest possible route that visits each city exactly once and returns to the origin city?

Origin

The traveling sales person problem was studied in the 18th century by mathematicians
Sir William Rowam Hamilton and **Thomas Penyngton Kirkman**.

A discussion about the work of Hamilton & Kirkman can be found in the book **Graph Theory (Biggs et al. 1976)**.

- ISBN-13: 978-0198539162
- ISBN-10: 0198539169

It is believed that the general form of the TSP have been first studied by Kalr Menger in Vienna and Harvard. The problem was later promoted by Hassler, Whitney & Merrill at Princeton. A detailed description about the connection between Menger & Whitney, and the development of the TSP can be found in **On the history of combinatorial optimization (till 1960)**

Characteristics

- The travel costs are symmetric:
  - traveling costs from city A to city B are just as much as traveling from B to A.
- This problem is an NP-hard optimization problem.
- To calculate the number of different tours through n cities:
  - Given a starting city,
  - There are n-1 choices for the second city,
  - And n-2 choices for the third city, etc.
  - Multiplying these together we get (n-1)! = (n-1) (n-2) . . 1.
  - Now since our travel costs do not depend on the direction we take around the tour:
    - this number by 2
    - (n-1)!/2.

Simulated Annealing Algorithm

The simulated annealing algorithm was originally inspired from the process of annealing in metal work.

Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties.

Pseudocode

Given an initial solution, the simulated annealing process, will start with a high temperature and gradually cool down until the desired temperature is reached.

For each temperature, a neighbouring new solution **newSolution** is calculated. The higher the temperature the higher the probability of accepting the new solution as a possible bester solution.

Once the desired temperature is reached, the best solution found is returned

```
Solution = initial_solution;

temperature = initial_temperature;
while (temperature > final_temperature) {

    do tries_per_temperature times {
        newSolution = neighbour(solution);
        If P(E(solution), E(newSolution), T) >= random(0, 1)
            solution = newSolution;
    }

    temperature = temperature * cooling factor;
}

Output: the best solution
```

pgRouting Implementation

pgRouting's implementation adds some extra parameters to allow some exit controls within the simulated annealing process.

- `max_changes_per_temperature`:
  - Limits the number of changes in the solution per temperature
  - Count is reset to `0` when **temperature** changes
  - Count is increased by :math:`1` when **solution** changes
- `max_consecutive_non_changes`:
  - Limits the number of consecutive non changes per temperature
  - Count is reset to `0` when **solution** changes
  - Count is increased by :math:`1` when **solution** changes
- `max_processing_time`:
  - Limits the time the simulated annealing is performed.

```
Solution = initial_solution;

temperature = initial_temperature;
WHILE (temperature > final_temperature) {

    DO tries_per_temperature times {
        newSolution = neighbour(solution);
        If Probability(E(solution), E(newSolution), T) >= random(0, 1)
            solution = newSolution;

        BREAK DO WHEN:
            max_changes_per_temperature is reached
            OR max_consecutive_non_changes is reached
    }

    temperature = temperature * cooling factor;
    BREAK WHILE WHEN:
        no changes were done in the current temperature
        OR max_processing_time has being reached
}

Output: the best solution found
```

Choosing parameters

There is no exact rule on how the parameters have to be chose, it will depend on the special characteristics of the problem.

- If the computational time is crucial, then limit execution time with **max_processing_time**.
- Make the **tries_per_temperture** depending on the number of cities (`n`), for example:
  - Useful to estimate the time it takes to do one cycle: use *1*
    - this will help to set a reasonable **max_processing_time**
  - `n * (n-1)`
  - `500 * n`
- For a faster decreasing the temperature set **cooling_factor** to a smaller number, and set to a higher number for a slower decrease.
- When for the same given data the same results are needed, set **randomize** to *false*.
  - When estimating how long it takes to do one cycle: use *false*

A recommendation is to play with the values and see what fits to the particular data.

Description of the Control Parameters

The control parameters are optional, and have a default value.

Parameter	Type	Default	Description
start_vid	`BIGINT`	*0*	The greedy part of the implementation will use this identifier.
end_vid	`BIGINT`	*0*	Last visiting vertex before returning to start_vid.
max_processing_time	`FLOAT`	*+infinity*	Stop the annealing processing when the value is reached.

Parameter	Type	Default	Description
tries_per_temperature	INTEGER	*500*	Maximum number of times a neighbor(s) is searched in each temperature.
max_changes_per_temperature	INTEGER	*60*	Maximum number of times the solution is changed in each temperature.
max_consecutive_non_changes	INTEGER	*100*	Maximum number of consecutive times the solution is not changed in each temperature.
initial_temperature	FLOAT	*100*	Starting temperature.
final_temperature	FLOAT	*0.1*	Ending temperature.
cooling_factor	FLOAT	*0.9*	Value between between 0 and 1 (not including) used to calculate the next temperature.
randomize	BOOLEAN	*true*	Choose the random seed • true: Use current time as seed • false: Use *1* as seed. Using this value will get the same results with the same data in each execution.

Description of the return columns

Returns SET OF (seq, node, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Row sequence.
node	BIGINT	Identifier of the node/coordinate/point.
cost	FLOAT	Cost to traverse from the current `node` to the next `node` in the path sequence. • `0` for the last row in the path sequence.
agg_cost	FLOAT	Aggregate cost from the `node` at `seq = 1` to the current node. • `0` for the first row in the path sequence.

See Also

References

- **Simulated annaeling algorithm for beginners**
- **Wikipedia: Traveling Salesman Problem**
- **Wikipedia: Simulated annealing**

Indices and tables

- **Index**
- **Search Page**

Spanning Tree - Category

- **Kruskal - Family of functions**
- **Prim - Family of functions**

A spanning tree of an undirected graph is a tree that includes all the vertices of G with the minimum possible number of edges.

For a disconnected graph, there there is no single tree, but a spanning forest, consisting of a spanning tree of each connected component.

- **Supported versions:** current(**3.0**)

See Also

- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

- **Index**
- **Search Page**

K shortest paths - Category

- **pgr_KSP** - Yen's algorithm based on pgr_dijkstra

Proposed

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

- **pgr_withPointsKSP - Proposed** - Yen's algorithm based on pgr_withPoints

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4**

Indices and tables

- **Index**
- **Search Page**

pgr_trsp - Turn Restriction Shortest Path (TRSP)

`pgr_trsp` — Returns the shortest path with support for turn restrictions.

Availability

- Version 2.1.0
  - New *Via* **prototypes**
    - pgr_trspViaVertices
    - pgr_trspViaEdges
- Version 2.0.0
  - **Official** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

The turn restricted shorthest path (TRSP) is a shortest path algorithm that can optionally take into account complicated turn restrictions like those found in real world navigable road networks. Performamnce wise it is nearly as fast as the A* search but has many additional features like it works with edges rather than the nodes of the network. Returns a set of (seq, id1, id2, cost) or (seq, id1, id2, id3, cost) rows, that make up a path.

```
pgr_trsp(sql text, source integer, target integer,
                directed boolean, has_rcost boolean [,restrict_sql text]);
RETURNS SETOF (seq, id1, id2, cost)
```

```
pgr_trsp(sql text, source_edge integer, source_pos float8,
                   target_edge integer, target_pos float8,
                directed boolean, has_rcost boolean [,restrict_sql text]);
RETURNS SETOF (seq, id1, id2, cost)
```

```
pgr_trspViaVertices(sql text, vids integer[],
                directed boolean, has_rcost boolean
                [, turn_restrict_sql text]);
RETURNS SETOF (seq, id1, id2, id3, cost)
```

```
pgr_trspViaEdges(sql text, eids integer[], pcts float8[],
                directed boolean, has_rcost boolean
                [, turn_restrict_sql text]);
RETURNS SETOF (seq, id1, id2, id3, cost)
```

The main characteristics are:

The Turn Restricted Shortest Path algorithm (TRSP) is similar to the shooting star in that you can specify turn restrictions.

The TRSP setup is mostly the same as **Dijkstra shortest path** with the addition of an optional turn restriction table. This provides an easy way of adding turn restrictions to a road network by placing them in a separate table.

sql:
a SQL query, which should return a set of rows with the following columns:

```
SELECT id, source, target, cost, [,reverse_cost] FROM edge_table
```

id:
`int4` identifier of the edge
source:
`int4` identifier of the source vertex
target:
`int4` identifier of the target vertex
cost:
`float8` value, of the edge traversal cost. A negative cost will prevent the edge from being inserted in the graph.
reverse_cost:
(optional) the cost for the reverse traversal of the edge. This is only used when the `directed` and `has_rcost` parameters are `true` (see the above remark about negative costs).

source:
`int4` **NODE id** of the start point
target:
`int4` **NODE id** of the end point
directed:
`true` if the graph is directed
has_rcost:
if `true`, the `reverse_cost` column of the SQL generated set of rows will be used for the cost of the traversal of the edge in the opposite direction.
restrict_sql:
(optional) a SQL query, which should return a set of rows with the following columns:

```
SELECT to_cost, target_id, via_path FROM restrictions
```

to_cost:
`float8` turn restriction cost
target_id:
`int4` target id
via_path:
`text` comma separated list of edges in the reverse order of `rule`

Another variant of TRSP allows to specify **EDGE id** of source and target together with a fraction to interpolate the position:

source_edge:
`int4` **EDGE id** of the start edge
source_pos:
`float8` fraction of 1 defines the position on the start edge
target_edge:
`int4` **EDGE id** of the end edge
target_pos:
`float8` fraction of 1 defines the position on the end edge

Returns set of:

seq:
row sequence
id1:
node ID
id2:
edge ID (`-1` for the last row)
cost:
cost to traverse from `id1` using `id2`

Support for Vias

Warning

The Support for Vias functions are prototypes. Not all corner cases are being considered.

We also have support for vias where you can say generate a from A to B to C, etc. We support both methods above only you pass an array of vertices or and array of edges and percentage position along the edge in two arrays.

sql:
a SQL query, which should return a set of rows with the following columns:

```
SELECT id, source, target, cost, [,reverse_cost] FROM edge_table
```

id:
int4 identifier of the edge
source:
int4 identifier of the source vertex
target:
int4 identifier of the target vertex
cost:
float8 value, of the edge traversal cost. A negative cost will prevent the edge from being inserted in the graph.
reverse_cost:
(optional) the cost for the reverse traversal of the edge. This is only used when the directed and has_rcost parameters are true (see the above remark about negative costs).

vids:
int4[] An ordered array of **NODE id** the path will go through from start to end.
directed:
true if the graph is directed
has_rcost:
if true, the reverse_cost column of the SQL generated set of rows will be used for the cost of the traversal of the edge in the opposite direction.
restrict_sql:
(optional) a SQL query, which should return a set of rows with the following columns:

```
SELECT to_cost, target_id, via_path FROM restrictions
```

to_cost:
float8 turn restriction cost
target_id:
int4 target id
via_path:
text commar separated list of edges in the reverse order of rule

Another variant of TRSP allows to specify **EDGE id** together with a fraction to interpolate the position:

eids:
int4 An ordered array of **EDGE id** that the path has to traverse
pcts:
float8 An array of fractional positions along the respective edges in eids, where 0.0 is the start of the edge and 1.0 is the end of the eadge.

Returns set of:

seq:
row sequence
id1:
route ID
id2:
node ID
id3:
edge ID (-1 for the last row)
cost:
cost to traverse from id2 using id3

Additional Examples

Example:
Without turn restrictions

```
SELECT * FROM pgr_trsp(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
        7, 12, false, false
    );
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   7 |   6 |    1
   1 |   8 |   7 |    1
   2 |   5 |   8 |    1
   3 |   6 |   9 |    1
   4 |   9 |  15 |    1
   5 |  12 |  -1 |    0
(6 rows)
```

Example:
With turn restrictions

Then a query with turn restrictions is created as:

```
SELECT * FROM pgr_trsp(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
        2, 7, false, false,
        'SELECT to_cost, target_id::int4,
        from_edge || coalesce("," || via_path, "") AS via_path
        FROM restrictions'
    );
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   4 |    1
   1 |   5 |  10 |    1
   2 |  10 |  12 |    1
   3 |  11 |  11 |    1
   4 |   6 |   8 |    1
   5 |   5 |   7 |    1
   6 |   8 |   6 |    1
   7 |   7 |  -1 |    0
(8 rows)

SELECT * FROM pgr_trsp(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
        7, 11, false, false,
        'SELECT to_cost, target_id::int4,
        from_edge || coalesce("," || via_path, "") AS via_path
        FROM restrictions'
    );
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   7 |   6 |    1
   1 |   8 |   7 |    1
   2 |   5 |   8 |    1
   3 |   6 |   9 |    1
   4 |   9 |  15 |    1
   5 |  12 |  13 |    1
   6 |  11 |  -1 |    0
(7 rows)
```

An example query using vertex ids and via points:

```
SELECT * FROM pgr_trspViaVertices(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
        ARRAY[2,7,11]::INTEGER[],
        false,  false,
        'SELECT to_cost, target_id::int4, from_edge ||
        coalesce(","||via_path,"") AS via_path FROM restrictions');
 seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+------
   1 |   1 |   2 |   4 |    1
   2 |   1 |   5 |  10 |    1
   3 |   1 |  10 |  12 |    1
   4 |   1 |  11 |  11 |    1
   5 |   1 |   6 |   8 |    1
   6 |   1 |   5 |   7 |    1
   7 |   1 |   8 |   6 |    1
   8 |   2 |   7 |   6 |    1
   9 |   2 |   8 |   7 |    1
  10 |   2 |   5 |   8 |    1
  11 |   2 |   6 |   9 |    1
  12 |   2 |   9 |  15 |    1
  13 |   2 |  12 |  13 |    1
  14 |   2 |  11 |  -1 |    0
(14 rows)
```

An example query using edge ids and vias:

```
SELECT * FROM pgr_trspViaEdges(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost,
        reverse_cost FROM edge_table',
        ARRAY[2,7,11]::INTEGER[],
        ARRAY[0.5, 0.5, 0.5]::FLOAT[],
        true,
        true,
        'SELECT to_cost, target_id::int4, FROM_edge ||
        coalesce(","||via_path,"") AS via_path FROM restrictions');
 seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+------
   1 |   1 |  -1 |   2 |  0.5
   2 |   1 |   2 |   4 |    1
   3 |   1 |   5 |   8 |    1
   4 |   1 |   6 |   9 |    1
   5 |   1 |   9 |  16 |    1
   6 |   1 |   4 |   3 |    1
   7 |   1 |   3 |   5 |    1
   8 |   1 |   6 |   8 |    1
   9 |   1 |   5 |   7 |    1
  10 |   2 |   5 |   8 |    1
  11 |   2 |   6 |   9 |    1
  12 |   2 |   9 |  16 |    1
  13 |   2 |   4 |   3 |    1
  14 |   2 |   3 |   5 |    1
  15 |   2 |   6 |  11 |  0.5
(15 rows)
```

The queries use the **Sample Data** network.

Known Issues

Introduction

pgr_trsp code has issues that are not being fixed yet, but as time passes and new functionality is added to pgRouting with wrappers to **hide** the issues, not to fix them.

For clarity on the queries:

- _pgr_trsp (internal_function) is the original code
- pgr_trsp (lower case) represents the wrapper calling the original code
- pgr_TRSP (upper case) represents the wrapper calling the replacement function, depending on the function, it can be:
  - pgr_dijkstra
  - pgr_dijkstraVia
  - pgr_withPoints
  - _pgr_withPointsVia (internal function)

The restrictions

The restriction used in the examples does not have to do anything with the graph:

- No vertex has id: 25, 32 or 33
- No edge has id: 25, 32 or 33

A restriction is assigned as:

```
SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path;
 to_cost | target_id | via_path
---------+-----------+----------
     100 |        25 | 32, 33
(1 row)
```

The back end code has that same restriction as follows

```
SELECT 1 AS id, 100::float AS cost, 25::INTEGER AS target_id, ARRAY[33, 32, 25] AS path;
 id | cost | target_id |    path
----+------+-----------+------------
  1 |  100 |        25 | {33,32,25}
(1 row)
```

therefore the shortest path expected are as if there was no restriction involved

The “Vertices” signature version

```
pgr_trsp(sql text, source integer, target integer,
        directed boolean, has_rcost boolean [,restrict_sql text]);
```

Different ways to represent ‘no path found`

- Sometimes represents with EMPTY SET a no path found
- Sometimes represents with Error a no path found

Returning EMPTY SET to represent no path found

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 15, true, true
);
 seq | id1 | id2 | cost
-----+-----+-----+------
(0 rows)
```

pgr_trsp calls **pgr_dijkstra** when there are no restrictions which returns *EMPTY SET* when a path is not found

```
SELECT * FROM pgr_dijkstra(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 15
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

Throwing EXCEPTION to represent no path found

```
SELECT * FROM pgr_trsp(
     $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
     1, 15, true, true,
     $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR:  Error computing path: Path Not Found
```

pgr_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, which will throw an EXCEPTION to represent no path found.

Routing from/to same location

When routing from location `1` to the same location `1`, no path is needed to reach the destination, its already there. Therefore is expected to return an *EMPTY SET* or an *EXCEPTION* depending on the parameters

- Sometimes represents with EMPTY SET no path found (expected)
- Sometimes represents with EXCEPTION no path found (expected)
- Sometimes finds a path (not expected)

Returning expected EMPTY SET to represent no path found

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    1, 1,  true, true
);
 seq | id1 | id2 | cost
-----+-----+-----+------
(0 rows)
```

`pgr_trsp` calls **pgr_dijkstra** when there are no restrictions which returns the expected to return *EMPTY SET* to represent no path found.

Returning expected EXCEPTION to represent no path found

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    14, 14, true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR:  Error computing path: Path Not Found
```

In this case `pgr_trsp` calls the original code when there are restrictions, even if they have nothing to do with the graph, in this case that code throws the expected EXCEPTION

Returning unexpected path

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    1, 1, true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   1 |   1 |    1
   1 |   2 |   4 |    1
   2 |   5 |   8 |    1
   3 |   6 |   9 |    1
   4 |   9 |  16 |    1
   5 |   4 |   3 |    1
   6 |   3 |   2 |    1
   7 |   2 |   1 |    1
   8 |   1 |  -1 |    0
(9 rows)
```

In this case `pgr_trsp` calls the original code when there are restrictions, even if they have nothing to do with the graph, in this case that code finds an unexpected path.

User contradictions

`pgr_trsp` unlike other pgRouting functions does not autodectect the existence of `reverse_cost` column. Therefor it has `has_rcost` parameter to check the existence of `reverse_cost` column. Contradictions happen:

- When the reverse_cost is missing, and the flag *has_rcost* is set to true
- When the reverse_cost exists, and the flag *has_rcost* is set to false

When the reverse_cost is missing, and the flag *has_rcost* is set to true.

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table$$,
    2, 3, false, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR:  Error, reverse_cost is used, but query did't return 'reverse_cost' column
```

An EXCEPTION is thrown.

When the reverse_cost exists, and the flag *has_rcost* is set to false

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    2, 3, false, false,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   4 |    1
   1 |   5 |   8 |    1
   2 |   6 |   5 |    1
   3 |   3 |  -1 |    0
(4 rows)
```

The `reverse_cost` column will be effectively removed and will cost execution time

The “Edges” signature version

```
pgr_trsp(sql text, source_edge integer, source_pos float8,
        target_edge integer, target_pos float8,
        directed boolean, has_rcost boolean [,restrict_sql text]);
```

Different ways to represent ‘no path found`

- Sometimes represents with EMPTY SET a no path found
- Sometimes represents with EXCEPTION a no path found

Returning EMPTY SET to represent no path found

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 0.5, 17, 0.5, true, true
);
 seq | id1 | id2 | cost
-----+-----+-----+------
(0 rows)
```

pgr_trsp calls **pgr_withPoints - Proposed** when there are no restrictions which returns *EMPTY SET* when a path is not found

Throwing EXCEPTION to represent no path found

```
SELECT * FROM _pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 0.5, 17, 0.5, true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR:  Error computing path: Path Not Found
```

pgr_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, which will throw an EXCEPTION to represent no path found.

Paths with equal number of vertices and edges

A path is made of *N* vertices and *N - 1* edges.

- Sometimes returns *N* vertices and *N - 1* edges.
- Sometimes returns *N - 1* vertices and *N - 1* edges.

Returning *N* vertices and *N - 1* edges.

```
SELECT * FROM pgr_TRSP(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 0.5, 1, 0.8, true, true
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  -1 |   1 |  0.3
   1 |  -2 |  -1 |    0
(2 rows)
```

pgr_trsp calls **pgr_withPoints - Proposed** when there are no restrictions which returns the correct number of rows that will include all the vertices. The last row will have a `-1` on the edge column to indicate the edge number is invalidu for that row.

Returning *N - 1* vertices and *N - 1* edges.

```
SELECT * FROM pgr_TRSP(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 0.5, 1, 0.8, true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  -1 |   1 |  0.3
(1 row)
```

pgr_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, and will not return the last vertex of the path.

Routing from/to same location

When routing from the same edge and position to the same edge and position, no path is needed to reach the destination, its already there. Therefore is expected to return an *EMPTY SET* or an *EXCEPTION* depending on the parameters, non of which is happening.

A path with 2 vertices and edge cost 0

```
SELECT * FROM pgr_TRSP(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 0.5, 1, 0.5, true, true
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  -1 |   1 |    0
   1 |  -2 |  -1 |    0
(2 rows)
```

pgr_trsp calls **pgr_withPoints - Proposed** setting the first (edge, position) with a differenct point id from the second (edge, position) making them different points. But the cost using the edge, is 0.

A path with 1 vertices and edge cost 0

```
SELECT * FROM pgr_TRSP(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    1, 0.5, 1, 0.5, true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  -1 |   1 |    0
(1 row)
```

pgr_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, and will not have the row for the vertex -2.

User contradictions

`pgr_trsp` unlike other pgRouting functions does not autodectect the existence of`reverse_cost` column. Therefor it has `has_rcost` parameter to check the existence of `reverse_cost` column. Contradictions happen:

- When the reverse_cost is missing, and the flag*has_rcost* is set to true
- When the reverse_cost exists, and the flag*has_rcost* is set to false

When the reverse_cost is missing, and the flag *has_rcost* is set to true.

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table$$,
    1, 0.5, 1, 0.8, false, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR:  Error, reverse_cost is used, but query did't return 'reverse_cost' column
```

An EXCEPTION is thrown.

When the reverse_cost exists, and the flag *has_rcost* is set to false

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    1, 0.5, 1, 0.8, false, false,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  -1 |   1 |  0.3
(1 row)
```

The `reverse_cost` column will be effectively removed and will cost execution time

Using a points of interest table

Given a set of points of interest:

```
SELECT * FROM pointsOfInterest;
 pid |  x  |  y  | edge_id | side | fraction |                  the_geom                  |                  newpoint
-----+-----+-----+---------+------+----------+--------------------------------------------+--------------------------------------------
   1 | 1.8 | 0.4 |       1 | l    |      0.4 | 0101000000CDCCCCCCCCCCFC3F9A9999999999D93F | 0101000000000000000000000409A9999999999D93F
   2 | 4.2 | 2.4 |      15 | r    |      0.4 | 0101000000CDCCCCCCCCCC1040333333333330340 | 0101000000000000000000104033333333333330340
   3 | 2.6 | 3.2 |      12 | l    |      0.6 | 0101000000CDCCCCCCCCCC04409A99999999990940 | 0101000000CDCCCCCCCCCC044000000000000000840
   4 | 0.3 | 1.8 |       6 | r    |      0.3 | 0101000000333333333333D33FCDCCCCCCCCCCFC3F | 0101000000333333333333D33F0000000000000040
   5 | 2.9 | 1.8 |       5 | l    |      0.8 | 0101000000333333333330740CDCCCCCCCCCCFC3F | 0101000000000000000000000840CDCCCCCCCCCCFC3F
   6 | 2.2 | 1.7 |       4 | b    |      0.7 | 01010000009A99999999901403333333333FB3F | 0101000000000000000000004033333333333FB3F
(6 rows)
```

Using pgr_trsp

```
SELECT * FROM pgr_TRSP(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    (SELECT edge_id::INTEGER  FROM pointsOfInterest WHERE pid = 1),
    (SELECT fraction  FROM pointsOfInterest WHERE pid = 1),
    (SELECT edge_id::INTEGER  FROM pointsOfInterest WHERE pid = 6),
    (SELECT fraction  FROM pointsOfInterest WHERE pid = 6),
    true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  -1 |   1 |  0.6
   1 |   2 |   4 |  0.7
(2 rows)
```

On *pgr_trsp*, to be able to use the table information:

- Each parameter has to be extracted explicitly from the table
- Regardles of the point pid original value
  - will always be -1 for the first point
  - will always be -2 for the second point
    - the row reaching point -2 will not be shown

Using pgr_withPoints - Proposed

```
SELECT * FROM pgr_withPoints(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    $$SELECT pid, edge_id, fraction FROM pointsOfInterest$$,
    -1, -6
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   -1 |    1 |  0.6 |        0
   2 |        2 |    2 |    4 |  0.7 |      0.6
   3 |        3 |   -6 |   -1 |    0 |      1.3
(3 rows)
```

Suggestion: use **pgr_withPoints - Proposed** when there are no turn restrictions:

- Results are more complete
- Column names are meaningful

Routing from a vertex to a point

Solving a shortest path from vertex 6 to pid 1 using a points of interest table

Using pgr_trsp

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    8, 1,
    (SELECT edge_id::INTEGER  FROM pointsOfInterest WHERE pid = 1),
    (SELECT fraction  FROM pointsOfInterest WHERE pid = 1),
    true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   6 |   8 |    1
   1 |   5 |   4 |    1
   2 |   2 |   1 |  0.6
(3 rows)
```

- Vertex 6 is on edge 8 at 1 fraction

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
    11, 0,
    (SELECT edge_id::INTEGER  FROM pointsOfInterest WHERE pid = 1),
    (SELECT fraction  FROM pointsOfInterest WHERE pid = 1),
    true, true,
    $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   6 |   8 |    1
   1 |   5 |   4 |    1
   2 |   2 |   1 |  0.6
(3 rows)
```

- Vertex 6 is also edge 11 at 0 fraction

Using pgr_withPoints - Proposed

```
SELECT * FROM pgr_withPoints(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edge_table$$,
    $$SELECT pid, edge_id, fraction FROM pointsOfInterest$$,
    6, -1
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    8 |    1 |        0
   2 |        2 |    5 |    4 |    1 |        1
   3 |        3 |    2 |    1 |  0.6 |        2
   4 |        4 |   -1 |   -1 |    0 |      2.6
(4 rows)
```

Suggestion: use **pgr_withPoints - Proposed** when there are no turn restrictions:

- No need to choose where the vertex is located.
- Results are more complete
- Column names are meaningful

prototypes

`pgr_trspViaVertices` and `pgr_trspViaEdges` were added to pgRouting as prototypes

These functions use the pgr_trsp functions inheriting all the problems mentioned above. When there are no restrictions and have a routing “via” problem with vertices:

- **pgr_dijkstraVia - Proposed**

See Also

Indices and tables

- **Index**
- **Search Page**

Cost - Category

- **pgr_aStarCost**
- **pgr_dijkstraCost**

Proposed

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

- **pgr_withPointsCost - Proposed**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4**

General Information

Characteristics

The main Characteristics are:

- Each function works as part of the family it belongs to.

- It does not return a path.
- Returns the sum of the costs of the resulting path(s) for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of *(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* int the non included values *(v, v)* is *0*.
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values *(u, v)* is \infty.
- Let be the case the values returned are stored in a table, so the unique index would be the pair *(start_vid, end_vid)*.
- Depending on the function and its parameters, the results can be symmetric.
  - The *agg_cost* of *(u, v)* is the same as for *(v, u)*.
- Any duplicated value in the *start_vids* or in *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending

See Also

Indices and tables

- **Index**
- **Search Page**

Cost Matrix - Category

- **pgr_aStarCostMatrix**
- **pgr_dijkstraCostMatrix**

proposed

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

- **pgr_withPointsCostMatrix - proposed**

pgr_withPointsCostMatrix - proposed

`pgr_withPointsCostMatrix` - Calculates the shortest path and returns only the aggregate cost of the shortest path(s) found, for the combination of points given.

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **proposed** function
- **Supported versions:** current(**3.0**)
- **Unsupported versions: 2.6 2.5 2.4 2.3**

Description

- **TBD**

Signatures

Summary

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Note

There is no **details** flag, unlike the other members of the withPoints family of functions.

Using default

The minimal signature:

- Is for a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

Cost matrix for points \{1, 6\} and vertices \{3, 6\} on a **directed** graph

```
SELECT * FROM pgr_withPointsCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction from pointsOfInterest',
    array[-1, 3, 6, -6]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
        -6 |      -1 |      1.3
        -6 |       3 |      4.3
        -6 |       6 |      1.3
        -1 |      -6 |      1.3
        -1 |       3 |      5.6
        -1 |       6 |      2.6
         3 |      -6 |      1.7
         3 |      -1 |      1.6
         3 |       6 |        1
         6 |      -6 |      1.3
         6 |      -1 |      2.6
         6 |       3 |        3
(12 rows)
```

Complete Signature

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vids,
    directed:=true, driving_side:='b')
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

Cost matrix for points \{1, 6\} and vertices \{3, 6\} on an **undirected** graph

- Returning a **symmetrical** cost matrix
- Using the default **side** value on the **points_sql** query
- Using the default **driving_side** value

```
SELECT * FROM pgr_withPointsCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction from pointsOfInterest',
    array[-1, 3, 6, -6], directed := false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
        -6 |      -1 |      1.3
        -6 |       3 |      1.7
        -6 |       6 |      1.3
        -1 |      -6 |      1.3
        -1 |       3 |      1.6
        -1 |       6 |      2.6
         3 |      -6 |      1.7
         3 |      -1 |      1.6
         3 |       6 |        1
         6 |      -6 |      1.3
         6 |      -1 |      2.6
         6 |       3 |        1
(12 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
points_sql	TEXT	Points SQL query as described above.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of starting vertices. When negative: is a point's pid.
directed	BOOLEAN	(optional). When `false` the graph is considered as Undirected. Default is `true` which considers the graph as Directed.
driving_side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: • In the right or left or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.

Returns SET OF `(start_vid, end_vid, agg_cost)`

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vetrices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in the query.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `end_vid`.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the Points SQL query

points_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Description

Column	Type	Description
pid	ANY-INTEGER	(optional) Identifier of the point. • If column present, it can not be NULL. • If column not present, a sequential identifier will be given automatically.
edge_id	ANY-INTEGER	Identifier of the “closest” edge to the point.
fraction	ANY-NUMERICAL	Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	(optional) Value in [‘b’, ‘r’, ‘l’, NULL] indicating if the point is: • In the right, left of the edge or • If it doesn’t matter with ‘b’ or NULL. • If column not present ‘b’ is considered.

Where:

ANY-INTEGER:
smallint, int, bigint
ANY-NUMERICAL:
smallint, int, bigint, real, float

Additional Examples

Example:
pgr_TSP using pgr_withPointsCostMatrix for points \{1, 6\} and vertices \{3, 6\} on an **undirected** graph

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_withPointsCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
        'SELECT pid, edge_id, fraction from pointsOfInterest',
        array[-1, 3, 6, -6], directed := false);
    $$,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |   -6 |  1.3 |        0
   2 |   -1 |  1.6 |      1.3
   3 |    3 |    1 |      2.9
   4 |    6 |  1.3 |      3.9
   5 |   -6 |    0 |      5.2
(5 rows)
```

See Also

- **pgr_withPoints - Proposed**
- **Cost Matrix - Category**
- **pgr_TSP**
- *sampledata* network.

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions: 2.5 2.4**

General Information

Synopsis

Traveling Sales Person - Family of functions needs as input a symmetric cost matrix and no edge*(u, v)* must value \infty.

This collection of functions will return a cost matrix in form of a table.

Characteristics

The main Characteristics are:

- Can be used as input to **pgr_TSP**.
  - **directly:**

when the resulting matrix is symmetric and there is no\infty value.

- It will be the users responsibility to make the matrix symmetric.
  - By using geometric or harmonic average of the non symmetric values.
  - By using max or min the non symmetric values.
  - By setting the upper triangle to be the mirror image of the lower triangle.
  - By setting the lower triangle to be the mirror image of the upper triangle.
- It is also the users responsibility to fix an\infty value.

- Each function works as part of the family it belongs to.
- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of*(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* int the non included values*(v, v)* is *0*.
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values*(u, v)* is \infty.
- Let be the case the values returned are stored in a table, so the unique index would be the pair*(start_vid, end_vid)*.
- Depending on the function and its parameters, the results can be symmetric.
  - The *agg_cost* of *(u, v)* is the same as for*(v, u)*.
- Any duplicated value in the *start_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: approximately O(| start_vids | * (V \log V + E))

See Also

- **Traveling Sales Person - Family of functions**

Indices and tables

- **Index**
- **Search Page**

Driving Distance - Category

- **pgr_drivingDistance** - Driving Distance based on Dijkstra's algorithm
- **pgr_primDD** - Driving Distance based on Prim's algorithm
- **pgr_kruskalDD** - Driving Distance based on Kruskal's algorithm
- Post pocessing
  - **pgr_alphaShape** - Alpha shape computation

Proposed

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

- **pgr_withPointsDD - Proposed** - Driving Distance based on pgr_withPoints

pgr_alphaShape

`pgr_alphaShape` — Polygon part of an alpha shape.

Availability

- Version 3.0.0
  - Breaking change on signature
  - Old signature no longer supported

- Version 2.1.0
  - Added alpha argument with default 0 (use optimal value)
  - Support to return multiple outer/inner ring
- Version 2.0.0
  - **Official** function
  - Renamed from version 1.x

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2** **2.1** **2.0**

Description

Returns the polygon part of an alpha shape.

Characteristics

- Input is a *geometry* and returns a *geometry*
- Uses PostGis ST_DelaunyTriangles
- Instead of using CGAL's definition of *alpha* it use the `spoon_radius`
  - spoon_radius = \sqrt alpha
- A Triangle area is considered part of the alpha shape when circumcenter\ radius < spoon_radius
- When the total number of points is less than 3, returns an EMPTY geometry

Signatures

Summary

```
pgr_alphaShape(geometry,   [spoon_radius])
RETURNS geometry
```

Example: passing a geometry collection with spoon radius 1.5 using the return variable `geom`

```
SELECT ST_Area(pgr_alphaShape((SELECT ST_Collect(the_geom) FROM edge_table_vertices_pgr), 1.5));
 st_area
---------
    9.75
(1 row)
```

Parameters

Parameter	Type	Default	Description
geometry	geometry		Geometry with at least 3 points
spoon_radius	FLOAT		The radius of the spoon

Return Value

Kind of geometry	Description
GEOMETRY COLLECTION	A Geometry collection of Polygons

See Also

- **pgr_drivingDistance**
- **Sample Data** network.
- **ST_ConcaveHull**

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4**

See Also

Indices and tables

- **Index**
- **Search Page**

See Also

Indices and tables

- **Index**
- **Search Page**

All Pairs - Family of Functions

- **pgr_floydWarshall** - Floyd-Warshall's algorithm.
- **pgr_johnson** - Johnson's algorithm

aStar - Family of functions

- **pgr_aStar** - A* algorithm for the shortest path.
- **pgr_aStarCost** - Get the aggregate cost of the shortest paths.
- **pgr_aStarCostMatrix** - Get the cost matrix of the shortest paths.

Bidirectional A* - Family of functions

- **pgr_bdAstar** - Bidirectional A* algorithm for obtaining paths.
- **pgr_bdAstarCost** - Bidirectional A* algorithm to calculate the cost of the paths.
- **pgr_bdAstarCostMatrix** - Bidirectional A* algorithm to calculate a cost matrix of paths.

Bidirectional Dijkstra - Family of functions

- **pgr_bdDijkstra** - Bidirectional Dijkstra algorithm for the shortest paths.
- **pgr_bdDijkstraCost** - Bidirectional Dijkstra to calculate the cost of the shortest paths
- **pgr_bdDijkstraCostMatrix** - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

Components - Family of functions

- **pgr_connectedComponents** - Connected components of an undirected graph.
- **pgr_strongComponents** - Strongly connected components of a directed graph.
- **pgr_biconnectedComponents** - Biconnected components of an undirected graph.
- **pgr_articulationPoints** - Articulation points of an undirected graph.
- **pgr_bridges** - Bridges of an undirected graph.

Contraction - Family of functions

- **pgr_contraction**

Dijkstra - Family of functions

- **pgr_dijkstra** - Dijkstra's algorithm for the shortest paths.
- **pgr_dijkstraCost** - Get the aggregate cost of the shortest paths.
- **pgr_dijkstraCostMatrix** - Use pgr_dijkstra to create a costs matrix.
- **pgr_drivingDistance** - Use pgr_dijkstra to calculate catchament information.
- **pgr_KSP** - Use Yen algorithm with pgr_dijkstra to get the K shortest paths.

Flow - Family of functions

- **pgr_maxFlow** - Only the Max flow calculation using Push and Relabel algorithm.
- **pgr_boykovKolmogorov** - Boykov and Kolmogorov with details of flow on edges.
- **pgr_edmondsKarp** - Edmonds and Karp algorithm with details of flow on edges.
- **pgr_pushRelabel** - Push and relabel algorithm with details of flow on edges.
- Applications
  - **pgr_edgeDisjointPaths** - Calculates edge disjoint paths between two groups of vertices.
  - **pgr_maxCardinalityMatch** - Calculates a maximum cardinality matching in a graph.

Kruskal - Family of functions

- **pgr_kruskal**
- **pgr_kruskalBFS**
- **pgr_kruskalDD**
- **pgr_kruskalDFS**

Prim - Family of functions

- **pgr_prim**
- **pgr_primBFS**
- **pgr_primDD**
- **pgr_primDFS**

Topology - Family of Functions

- **pgr_createTopology** - to create a topology based on the geometry.
- **pgr_createVerticesTable** - to reconstruct the vertices table based on the source and target information.
- **pgr_analyzeGraph** - to analyze the edges and vertices of the edge table.
- **pgr_analyzeOneWay** - to analyze directionality of the edges.
- **pgr_nodeNetwork** -to create nodes to a not noded edge table.

Traveling Sales Person - Family of functions

- **pgr_TSP** - When input is given as matrix cell information.
- **pgr_TSPeuclidean** - When input are coordinates.

pgr_trsp - Turn Restriction Shortest Path (TRSP) - Turn Restriction Shortest Path (TRSP)

Functions by categories

Cost - Category

- **pgr_aStarCost**
- **pgr_dijkstraCost**

Cost Matrix - Category

- **pgr_aStarCostMatrix**
- **pgr_dijkstraCostMatrix**

Driving Distance - Category

- **pgr_drivingDistance** - Driving Distance based on Dijkstra's algorithm
- **pgr_primDD** - Driving Distance based on Prim's algorithm
- **pgr_kruskalDD** - Driving Distance based on Kruskal's algorithm
- Post pocessing
  - **pgr_alphaShape** - Alpha shape computation

K shortest paths - Category

- **pgr_KSP** - Yen's algorithm based on pgr_dijkstra

Spanning Tree - Category

- **Kruskal - Family of functions**
- **Prim - Family of functions**

Available Functions but not official pgRouting functions

- **Proposed Functions**
- **Experimental Functions**

Proposed Functions

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

Families

Dijkstra - Family of functions

- **pgr_dijkstraVia - Proposed** - Get a route of a seuence of vertices.

withPoints - Family of functions

- **pgr_withPoints - Proposed** - Route from/to points anywhere on the graph.

- **pgr_withPointsCost - Proposed** - Costs of the shortest paths.
- **pgr_withPointsCostMatrix - proposed** - Costs of the shortest paths.
- **pgr_withPointsKSP - Proposed** - K shortest paths.
- **pgr_withPointsDD - Proposed** - Driving distance.

categories

Cost - Category

- **pgr_withPointsCost - Proposed**

Cost Matrix - Category

- **pgr_withPointsCostMatrix - proposed**

Driving Distance - Category

- **pgr_withPointsDD - Proposed** - Driving Distance based on pgr_withPoints

K shortest paths - Category

- **pgr_withPointsKSP - Proposed** - Yen's algorithm based on pgr_withPoints

withPoints - Family of functions

When points are also given as input:

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

- **pgr_withPoints - Proposed** - Route from/to points anywhere on the graph.
- **pgr_withPointsCost - Proposed** - Costs of the shortest paths.
- **pgr_withPointsCostMatrix - proposed** - Costs of the shortest paths.
- **pgr_withPointsKSP - Proposed** - K shortest paths.
- **pgr_withPointsDD - Proposed** - Driving distance.

pgr_withPoints - Proposed

`pgr_withPoints` - Returns the shortest path in a graph with additional temporary vertices.

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2**

Description

Modify the graph to include points defined by points_sql. Using Dijkstra algorithm, find the shortest path(s)

The main characteristics are:

- Process is done only on edges with positive costs.
- Vertices of the graph are:
  - **positive** when it belongs to the edges_sql
  - **negative** when it belongs to the points_sql
- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path. - The agg_cost the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path: - The agg_cost the non included values (u, v) is ∞
- For optimization purposes, any duplicated value in the start_vids or end_vids are ignored.
- The returned values are ordered: - start_vid ascending - end_vid ascending
- Running time: O(|start_vids|\times(V \log V + E))

Signatures

Summary

```
pgr_withPoints(edges_sql, points_sql, from_vid,  to_vid  [, directed] [, driving_side] [, details])
pgr_withPoints(edges_sql, points_sql, from_vid,  to_vids [, directed] [, driving_side] [, details])
pgr_withPoints(edges_sql, points_sql, from_vids, to_vid  [, directed] [, driving_side] [, details])
pgr_withPoints(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, [start_vid,] [end_vid,] node, edge, cost, agg_cost)
```

Using defaults

```
pgr_withPoints(edges_sql, points_sql, from_vid, to_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Example:

From point 1 to point 3

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of points_sql query.

```
SELECT * FROM pgr_withPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -3);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   -1 |    1 |  0.6 |        0
   2 |        2 |    2 |    4 |    1 |      0.6
   3 |        3 |    5 |   10 |    1 |      1.6
   4 |        4 |   10 |   12 |  0.6 |      2.6
   5 |        5 |   -3 |   -1 |    0 |      3.2
(5 rows)
```

One to One

```
pgr_withPoints(edges_sql, points_sql, from_vid,  to_vid  [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Example:

From point 1 to vertex 3 with details of passing points

```
SELECT * FROM pgr_withPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 3,
    details := true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   -1 |    1 |  0.6 |        0
   2 |        2 |    2 |    4 |  0.7 |      0.6
   3 |        3 |   -6 |    4 |  0.3 |      1.3
   4 |        4 |    5 |    8 |    1 |      1.6
   5 |        5 |    6 |    9 |    1 |      2.6
   6 |        6 |    9 |   16 |    1 |      3.6
   7 |        7 |    4 |    3 |    1 |      4.6
   8 |        8 |    3 |   -1 |    0 |      5.6
(8 rows)
```

One to Many

```
pgr_withPoints(edges_sql, points_sql, from_vid,  to_vids [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
```

Example:

From point 1 to point 3 and vertex 5

```
SELECT * FROM pgr_withPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, ARRAY[-3,5]);
 seq | path_seq | end_pid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |      -3 |   -1 |    1 |  0.6 |        0
   2 |        2 |      -3 |    2 |    4 |    1 |      0.6
   3 |        3 |      -3 |    5 |   10 |    1 |      1.6
   4 |        4 |      -3 |   10 |   12 |  0.6 |      2.6
   5 |        5 |      -3 |   -3 |   -1 |    0 |      3.2
   6 |        1 |       5 |   -1 |    1 |  0.6 |        0
   7 |        2 |       5 |    2 |    4 |    1 |      0.6
   8 |        3 |       5 |    5 |   -1 |    0 |      1.6
(8 rows)
```

Many to One

```
pgr_withPoints(edges_sql, points_sql, from_vids, to_vid  [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
```

Example:

From point 1 and vertex 2 to point 3

```
SELECT * FROM pgr_withPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], -3);
 seq | path_seq | start_pid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |        -1 |   -1 |    1 |  0.6 |        0
   2 |        2 |        -1 |    2 |    4 |    1 |      0.6
   3 |        3 |        -1 |    5 |   10 |    1 |      1.6
   4 |        4 |        -1 |   10 |   12 |  0.6 |      2.6
   5 |        5 |        -1 |   -3 |   -1 |    0 |      3.2
   6 |        1 |         2 |    2 |    4 |    1 |        0
   7 |        2 |         2 |    5 |   10 |    1 |        1
   8 |        3 |         2 |   10 |   12 |  0.6 |        2
   9 |        4 |         2 |   -3 |   -1 |    0 |      2.6
(9 rows)
```

Many to Many

```
pgr_withPoints(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
```

Example:

From point 1 and vertex 2 to point 3 and vertex 7

```
SELECT * FROM pgr_withPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], ARRAY[-3,7]);
 seq | path_seq | start_pid | end_pid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |        -1 |      -3 |   -1 |    1 |  0.6 |        0
   2 |        2 |        -1 |      -3 |    2 |    4 |    1 |      0.6
   3 |        3 |        -1 |      -3 |    5 |   10 |    1 |      1.6
   4 |        4 |        -1 |      -3 |   10 |   12 |  0.6 |      2.6
   5 |        5 |        -1 |      -3 |   -3 |   -1 |    0 |      3.2
   6 |        1 |        -1 |       7 |   -1 |    1 |  0.6 |        0
   7 |        2 |        -1 |       7 |    2 |    4 |    1 |      0.6
   8 |        3 |        -1 |       7 |    5 |    7 |    1 |      1.6
   9 |        4 |        -1 |       7 |    8 |    6 |    1 |      2.6
  10 |        5 |        -1 |       7 |    7 |   -1 |    0 |      3.6
  11 |        1 |         2 |      -3 |    2 |    4 |    1 |        0
  12 |        2 |         2 |      -3 |    5 |   10 |    1 |        1
  13 |        3 |         2 |      -3 |   10 |   12 |  0.6 |        2
  14 |        4 |         2 |      -3 |   -3 |   -1 |    0 |      2.6
  15 |        1 |         2 |       7 |    2 |    4 |    1 |        0
  16 |        2 |         2 |       7 |    5 |    7 |    1 |        1
  17 |        3 |         2 |       7 |    8 |    6 |    1 |        2
  18 |        4 |         2 |       7 |    7 |   -1 |    0 |        3
(18 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
points_sql	TEXT	Points SQL query as described above.
start_vid	ANY-INTEGER	Starting vertex identifier. When negative: is a point's pid.
end_vid	ANY-INTEGER	Ending vertex identifier. When negative: is a point's pid.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of starting vertices. When negative: is a point's pid.
end_vids	ARRAY[ANY-INTEGER]	Array of identifiers of ending vertices. When negative: is a point's pid.
directed	BOOLEAN	(optional). When `false` the graph is considered as Undirected. Default is`true` which considers the graph as Directed.
driving_side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: • In the right or left or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.
details	BOOLEAN	(optional). When `true` the results will include the points in points_sql that are in the path. Default is `false` which ignores other points of the points_sql.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the Points SQL query

points_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
pid	ANY-INTEGER	(optional) Identifier of the point. • If column present, it can not be NULL. • If column not present, a sequential identifier will be given automatically.
edge_id	ANY-INTEGER	Identifier of the “closest” edge to the point.
fraction	ANY-NUMERICAL	Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	(optional) Value in [‘b’, ‘r’, ‘l’, NULL] indicating if the point is: • In the right, left of the edge or • If it doesn’t matter with ‘b’ or NULL. • If column not present ‘b’ is considered.

Where:

ANY-INTEGER:
smallint, int, bigint
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

Column	Type	Description
seq	INTEGER	Row sequence.
path_seq	INTEGER	Path sequence that indicates the relative position on the path.
start_vid	BIGINT	Identifier of the starting vertex. When negative: is a point’s pid.
end_vid	BIGINT	Identifier of the ending vertex. When negative: is a point’s pid.
node	BIGINT	Identifier of the node: • A positive value indicates the node is a vertex of edges_sql. • A negative value indicates the node is a point of points_sql.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. • `-1` for the last row in the path sequence.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next `node` in the path sequence. • `0` for the last row in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_pid` to `node`. • `0` for the first row in the path sequence.

Additional Examples

Example:
Which path (if any) passes in front of point `6` or vertex `6` with **right** side driving topology.

```
SELECT ('(' || start_pid || ' => ' || end_pid ||') at ' || path_seq || 'th step:')::TEXT AS path_at,
       CASE WHEN edge = -1 THEN ' visits'
          ELSE ' passes in front of'
       END as status,
       CASE WHEN node < 0 THEN 'Point'
          ELSE 'Vertex'
       END as is_a,
       abs(node) as id
    FROM pgr_withPoints(
       'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
       'SELECT pid, edge_id, fraction, side from pointsOfInterest',
       ARRAY[1,-1], ARRAY[-2,-3,-6,3,6],
       driving_side := 'r',
       details := true)
    WHERE node IN (-6,6);
        path_at         |       status        | is_a  | id
------------------------+---------------------+-------+----
 (-1 => -6) at 4th step: |  visits             | Point |  6
 (-1 => -3) at 4th step: |  passes in front of | Point |  6
 (-1 => -2) at 4th step: |  passes in front of | Point |  6
 (-1 => -2) at 6th step: |  passes in front of | Vertex |  6
 (-1 => 3) at 4th step:  |  passes in front of | Point |  6
 (-1 => 3) at 6th step:  |  passes in front of | Vertex |  6
 (-1 => 6) at 4th step:  |  passes in front of | Point |  6
 (-1 => 6) at 6th step:  |  visits             | Vertex |  6
 (1 => -6) at 3th step:  |  visits             | Point |  6
 (1 => -3) at 3th step:  |  passes in front of | Point |  6
 (1 => -2) at 3th step:  |  passes in front of | Point |  6
 (1 => -2) at 5th step:  |  passes in front of | Vertex |  6
 (1 => 3) at 3th step:   |  passes in front of | Point |  6
 (1 => 3) at 5th step:   |  passes in front of | Vertex |  6
 (1 => 6) at 3th step:   |  passes in front of | Point |  6
 (1 => 6) at 5th step:   |  visits             | Vertex |  6
(16 rows)
```

Example:

Which path (if any) passes in front of point 6 or vertex 6 with **left** side driving topology.

```
SELECT ('(' || start_pid || ' => ' || end_pid ||') at ' || path_seq || 'th step:')::TEXT AS path_at,
       CASE WHEN edge = -1 THEN ' visits'
          ELSE ' passes in front of'
       END as status,
       CASE WHEN node < 0 THEN 'Point'
          ELSE 'Vertex'
       END as is_a,
       abs(node) as id
    FROM pgr_withPoints(
       'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
       'SELECT pid, edge_id, fraction, side from pointsOfInterest',
       ARRAY[1,-1], ARRAY[-2,-3,-6,3,6],
       driving_side := 'l',
       details := true)
    WHERE node IN (-6,6);
        path_at         |       status        | is_a  | id
------------------------+---------------------+-------+----
 (-1 => -6) at 3th step: |  visits             | Point |  6
 (-1 => -3) at 3th step: |  passes in front of | Point |  6
 (-1 => -2) at 3th step: |  passes in front of | Point |  6
 (-1 => -2) at 5th step: |  passes in front of | Vertex |  6
 (-1 => 3) at 3th step:  |  passes in front of | Point |  6
 (-1 => 3) at 5th step:  |  passes in front of | Vertex |  6
 (-1 => 6) at 3th step:  |  passes in front of | Point |  6
 (-1 => 6) at 5th step:  |  visits             | Vertex |  6
 (1 => -6) at 4th step:  |  visits             | Point |  6
 (1 => -3) at 4th step:  |  passes in front of | Point |  6
 (1 => -2) at 4th step:  |  passes in front of | Point |  6
 (1 => -2) at 6th step:  |  passes in front of | Vertex |  6
 (1 => 3) at 4th step:   |  passes in front of | Point |  6
 (1 => 3) at 6th step:   |  passes in front of | Vertex |  6
 (1 => 6) at 4th step:   |  passes in front of | Point |  6
 (1 => 6) at 6th step:   |  visits             | Vertex |  6
(16 rows)
```

Example:

From point 1 and vertex 2 to point 3 to vertex 7 on an **undirected** graph, with details.

```
SELECT * FROM pgr_withPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], ARRAY[-3,7],
    directed := false,
    details := true);
 seq | path_seq | start_pid | end_pid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |        -1 |      -3 |   -1 |    1 |  0.6 |        0
   2 |        2 |        -1 |      -3 |    2 |    4 |  0.7 |      0.6
   3 |        3 |        -1 |      -3 |   -6 |    4 |  0.3 |      1.3
   4 |        4 |        -1 |      -3 |    5 |   10 |    1 |      1.6
   5 |        5 |        -1 |      -3 |   10 |   12 |  0.6 |      2.6
   6 |        6 |        -1 |      -3 |   -3 |   -1 |    0 |      3.2
   7 |        1 |        -1 |       7 |   -1 |    1 |  0.6 |        0
   8 |        2 |        -1 |       7 |    2 |    4 |  0.7 |      0.6
   9 |        3 |        -1 |       7 |   -6 |    4 |  0.3 |      1.3
  10 |        4 |        -1 |       7 |    5 |    7 |    1 |      1.6
  11 |        5 |        -1 |       7 |    8 |    6 |  0.7 |      2.6
  12 |        6 |        -1 |       7 |   -4 |    6 |  0.3 |      3.3
  13 |        7 |        -1 |       7 |    7 |   -1 |    0 |      3.6
  14 |        1 |         2 |      -3 |    2 |    4 |  0.7 |        0
  15 |        2 |         2 |      -3 |   -6 |    4 |  0.3 |      0.7
  16 |        3 |         2 |      -3 |    5 |   10 |    1 |        1
  17 |        4 |         2 |      -3 |   10 |   12 |  0.6 |        2
  18 |        5 |         2 |      -3 |   -3 |   -1 |    0 |      2.6
  19 |        1 |         2 |       7 |    2 |    4 |  0.7 |        0
  20 |        2 |         2 |       7 |   -6 |    4 |  0.3 |      0.7
  21 |        3 |         2 |       7 |    5 |    7 |    1 |        1
  22 |        4 |         2 |       7 |    8 |    6 |  0.7 |        2
  23 |        5 |         2 |       7 |   -4 |    6 |  0.3 |      2.7
  24 |        6 |         2 |       7 |    7 |   -1 |    0 |        3
(24 rows)
```

The queries use the **Sample Data** network

See Also

- **withPoints - Family of functions**

Indices and tables

- **Index**
- **Search Page**

pgr_withPointsCost - Proposed

`pgr_withPointsCost` - Calculates the shortest path and returns only the aggregate cost of the shortest path(s) found, for the combination of points given.

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2**

Description

Modify the graph to include points defined by points_sql. Using Dijkstra algorithm, return only the aggregate cost of the shortest path(s) found.

The main characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of vertices in the modified graph.
- Vertices of the graph are:
  - **positive** when it belongs to the edges_sql
  - **negative** when it belongs to the points_sql
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of*(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* in the non included values*(v, v)* is *0*
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values*(u, v)* is \infty
- If the values returned are stored in a table, the unique index would be the pair:*(start_vid, end_vid)*.
- For **undirected** graphs, the results are **symmetric**.
  - The *agg_cost* of *(u, v)* is the same as for*(v, u)*.
- For optimization purposes, any duplicated value in the*start_vids* or *end_vids* is ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: O(| start_vids | * (V \log V + E))

Signatures

Summary

```
pgr_withPointsCost(edges_sql, points_sql, from_vid,  to_vid  [, directed] [, driving_side])
pgr_withPointsCost(edges_sql, points_sql, from_vid,  to_vids [, directed] [, driving_side])
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vid  [, directed] [, driving_side])
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Note

There is no **details** flag, unlike the other members of the withPoints family of functions.

Using defaults

```
pgr_withPointsCost(edges_sql, points_sql, start_vid, end_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

From point 1 to point 3

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -3);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
(1 row)
```

One to One

```
pgr_withPointsCost(edges_sql, points_sql, from_vid,  to_vid  [, directed] [, driving_side])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example:

From point 1 to vertex 3 on an **undirected** graph.

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 3,
    directed := false);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |       3 |      1.6
(1 row)
```

One to Many

```
pgr_withPointsCost(edges_sql, points_sql, from_vid,  to_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

From point 1 to point 3 and vertex 5 on a **directed** graph.

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, ARRAY[-3,5]);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
        -1 |       5 |      1.6
(2 rows)
```

Many to One

```
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vid  [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

From point 1 and vertex 2 to point 3 on a **directed** graph.

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], -3);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
         2 |      -3 |      2.6
(2 rows)
```

Many to Many

```
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example:

From point 1 and vertex 2 to point 3 and vertex 7 on a **directed** graph.

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], ARRAY[-3,7]);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
        -1 |       7 |      3.6
         2 |      -3 |      2.6
         2 |       7 |        3
(4 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
points_sql	TEXT	Points SQL query as described above.

Parameter	Type	Description
start_vid	ANY-INTEGER	Starting vertex identifier. When negative: is a point's pid.
end_vid	ANY-INTEGER	Ending vertex identifier. When negative: is a point's pid.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of starting vertices. When negative: is a point's pid.
end_vids	ARRAY[ANY-INTEGER]	Array of identifiers of ending vertices. When negative: is a point's pid.
directed	BOOLEAN	(optional). When false the graph is considered as Undirected. Default is true which considers the graph as Directed.
driving_side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: • In the right or left or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the Points SQL query

points_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
pid	ANY-INTEGER	(optional) Identifier of the point. • If column present, it can not be NULL. • If column not present, a sequential identifier will be given automatically.
edge_id	ANY-INTEGER	Identifier of the "closest" edge to the point.
fraction	ANY-NUMERICAL	Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: • In the right, left of the edge or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.

Where:

ANY-INTEGER:
smallint, int, bigint
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. When negative: is a point's pid.

Column	Type	Description
end_vid	BIGINT	Identifier of the ending point. When negative: is a point's pid.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
From point 1 and vertex 2 to point 3 and vertex 7, with **right** side driving topology

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], ARRAY[-3,7],
    driving_side := 'l');
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
        -1 |       7 |      3.6
         2 |      -3 |      2.6
         2 |       7 |        3
(4 rows)
```

Example:
From point 1 and vertex 2 to point 3 and vertex 7, with **left** side driving topology

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], ARRAY[-3,7],
    driving_side := 'r');
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |        4
        -1 |       7 |      4.4
         2 |      -3 |      2.6
         2 |       7 |        3
(4 rows)
```

Example:
From point 1 and vertex 2 to point 3 and vertex 7, does not matter driving side.

```
SELECT * FROM pgr_withPointsCost(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    ARRAY[-1,2], ARRAY[-3,7],
    driving_side := 'b');
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
        -1 |       7 |      3.6
         2 |      -3 |      2.6
         2 |       7 |        3
(4 rows)
```

The queries use the **Sample Data** network.

See Also

- **withPoints - Family of functions**

Indices and tables

- **Index**
- **Search Page**

pgr_withPointsKSP - Proposed

pgr_withPointsKSP - Find the K shortest paths using Yen's algorithm.

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL

- Name might not change. (But still can)
- Signature might not change. (But still can)
- Functionality might not change. (But still can)
- pgTap tests have being done. But might need more.
- Documentation might need refinement.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **proposed** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2**

Description

Modifies the graph to include the points defined in the `points_sql` and using Yen algorithm, finds the `K` shortest paths.

Signatures

Summary

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K [, directed] [, heap_paths] [, driving_side] [, details])
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

Using defaults

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K)
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

Example:
From point `1` to point `2` in `2` cycles

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of the query.
- No **heap paths** are returned.

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -2, 2);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.6 |        0
   2 |       1 |        2 |    2 |    4 |    1 |      0.6
   3 |       1 |        3 |    5 |    8 |    1 |      1.6
   4 |       1 |        4 |    6 |    9 |    1 |      2.6
   5 |       1 |        5 |    9 |   15 |  0.4 |      3.6
   6 |       1 |        6 |   -2 |   -1 |    0 |        4
   7 |       2 |        1 |   -1 |    1 |  0.6 |        0
   8 |       2 |        2 |    2 |    4 |    1 |      0.6
   9 |       2 |        3 |    5 |    8 |    1 |      1.6
  10 |       2 |        4 |    6 |   11 |    1 |      2.6
  11 |       2 |        5 |   11 |   13 |    1 |      3.6
  12 |       2 |        6 |   12 |   15 |  0.6 |      4.6
  13 |       2 |        7 |   -2 |   -1 |    0 |      5.2
(13 rows)
```

Complete Signature

Finds the `K` shortest paths depending on the optional parameters setup.

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K [, directed] [, heap_paths] [, driving_side] [, details])
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

Example:

From point 1 to vertex 6 in 2 cycles with details.

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 6, 2, details := true);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.6 |        0
   2 |       1 |        2 |    2 |    4 |  0.7 |      0.6
   3 |       1 |        3 |   -6 |    4 |  0.3 |      1.3
   4 |       1 |        4 |    5 |    8 |    1 |      1.6
   5 |       1 |        5 |    6 |   -1 |    0 |      2.6
   6 |       2 |        1 |   -1 |    1 |  0.6 |        0
   7 |       2 |        2 |    2 |    4 |  0.7 |      0.6
   8 |       2 |        3 |   -6 |    4 |  0.3 |      1.3
   9 |       2 |        4 |    5 |   10 |    1 |      1.6
  10 |       2 |        5 |   10 |   12 |  0.6 |      2.6
  11 |       2 |        6 |   -3 |   12 |  0.4 |      3.2
  12 |       2 |        7 |   11 |   13 |    1 |      3.6
  13 |       2 |        8 |   12 |   15 |  0.6 |      4.6
  14 |       2 |        9 |   -2 |   15 |  0.4 |      5.2
  15 |       2 |       10 |    9 |    9 |    1 |      5.6
  16 |       2 |       11 |    6 |   -1 |    0 |      6.6
(16 rows)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
points_sql	TEXT	Points SQL query as described above.
start_pid	ANY-INTEGER	Starting point id.
end_pid	ANY-INTEGER	Ending point id.
K	INTEGER	Number of shortest paths.
directed	BOOLEAN	(optional). When false the graph is considered as Undirected. Default is true which considers the graph as Directed.
heap_paths	BOOLEAN	(optional). When true the paths calculated to get the shortests paths will be returned also. Default is false only the K shortest paths are returned.
driving_side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: • In the right or left or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.
details	BOOLEAN	(optional). When true the results will include the driving distance to the points with in the distance. Default is false which ignores other points of the points_sql.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the Points SQL query

points_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
pid	ANY-INTEGER	(optional) Identifier of the point. • If column present, it can not be NULL. • If column not present, a sequential identifier will be given automatically.
edge_id	ANY-INTEGER	Identifier of the "closest" edge to the point.
fraction	ANY-NUMERICAL	Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: • In the right, left of the edge or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.

Where:

ANY-INTEGER:
smallint, int, bigint
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

Column	Type	Description
seq	INTEGER	Row sequence.
path_seq	INTEGER	Relative position in the path of node and edge. Has value 1 for the beginning of a path.
path_id	INTEGER	Path identifier. The ordering of the paths: For two paths i, j if i < j then agg_cost(i) <= agg_cost(j).
node	BIGINT	Identifier of the node in the path. Negative values are the identifiers of a point.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. • `-1` for the last row in the path sequence.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next `node` in the path sequence. • `0` for the last row in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_pid` to `node`. • `0` for the first row in the path sequence.

Additional Examples

Example:
Left side driving topology from point `1` to point `2` in `2` cycles, with details

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -2, 2,
    driving_side := 'l', details := true);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.6 |        0
   2 |       1 |        2 |    2 |    4 |  0.7 |      0.6
   3 |       1 |        3 |   -6 |    4 |  0.3 |      1.3
   4 |       1 |        4 |    5 |    8 |    1 |      1.6
   5 |       1 |        5 |    6 |   11 |    1 |      2.6
   6 |       1 |        6 |   11 |   13 |    1 |      3.6
   7 |       1 |        7 |   12 |   15 |  0.6 |      4.6
   8 |       1 |        8 |   -2 |   -1 |    0 |      5.2
   9 |       2 |        1 |   -1 |    1 |  0.6 |        0
  10 |       2 |        2 |    2 |    4 |  0.7 |      0.6
  11 |       2 |        3 |   -6 |    4 |  0.3 |      1.3
  12 |       2 |        4 |    5 |    8 |    1 |      1.6
  13 |       2 |        5 |    6 |    9 |    1 |      2.6
  14 |       2 |        6 |    9 |   15 |    1 |      3.6
  15 |       2 |        7 |   12 |   15 |  0.6 |      4.6
  16 |       2 |        8 |   -2 |   -1 |    0 |      5.2
(16 rows)
```

Example:

Right side driving topology from point 1 to point 2 in 2 cycles, with heap paths and details

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -2, 2,
    heap_paths := true, driving_side := 'r', details := true);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.4 |        0
   2 |       1 |        2 |    1 |    1 |    1 |      0.4
   3 |       1 |        3 |    2 |    4 |  0.7 |      1.4
   4 |       1 |        4 |   -6 |    4 |  0.3 |      2.1
   5 |       1 |        5 |    5 |    8 |    1 |      2.4
   6 |       1 |        6 |    6 |    9 |    1 |      3.4
   7 |       1 |        7 |    9 |   15 |  0.4 |      4.4
   8 |       1 |        8 |   -2 |   -1 |    0 |      4.8
   9 |       2 |        1 |   -1 |    1 |  0.4 |        0
  10 |       2 |        2 |    1 |    1 |    1 |      0.4
  11 |       2 |        3 |    2 |    4 |  0.7 |      1.4
  12 |       2 |        4 |   -6 |    4 |  0.3 |      2.1
  13 |       2 |        5 |    5 |    8 |    1 |      2.4
  14 |       2 |        6 |    6 |   11 |    1 |      3.4
  15 |       2 |        7 |   11 |   13 |    1 |      4.4
  16 |       2 |        8 |   12 |   15 |    1 |      5.4
  17 |       2 |        9 |    9 |   15 |  0.4 |      6.4
  18 |       2 |       10 |   -2 |   -1 |    0 |      6.8
  19 |       3 |        1 |   -1 |    1 |  0.4 |        0
  20 |       3 |        2 |    1 |    1 |    1 |      0.4
  21 |       3 |        3 |    2 |    4 |  0.7 |      1.4
  22 |       3 |        4 |   -6 |    4 |  0.3 |      2.1
  23 |       3 |        5 |    5 |   10 |    1 |      2.4
  24 |       3 |        6 |   10 |   12 |  0.6 |      3.4
  25 |       3 |        7 |   -3 |   12 |  0.4 |        4
  26 |       3 |        8 |   11 |   13 |    1 |      4.4
  27 |       3 |        9 |   12 |   15 |    1 |      5.4
  28 |       3 |       10 |    9 |   15 |  0.4 |      6.4
  29 |       3 |       11 |   -2 |   -1 |    0 |      6.8
(29 rows)
```

The queries use the **Sample Data** network.

See Also

- **withPoints - Family of functions**

Indices and tables

- **Index**
- **Search Page**

pgr_withPointsDD - Proposed

`pgr_withPointsDD` - Returns the driving distance from a starting point.

Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change. (But still can)
  - Signature might not change. (But still can)
  - Functionality might not change. (But still can)
  - pgTap tests have being done. But might need more.
  - Documentation might need refinement.

Boost Graph Inside

Availability

- Version 2.2.0

- New **proposed** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2**

Description

Modify the graph to include points and using Dijkstra algorithm, extracts all the nodes and points that have costs less than or equal to the value `distance` from the starting point. The edges extracted will conform the corresponding spanning tree.

Signatures

Summary

```
pgr_withPointsDD(edges_sql, points_sql, from_vids, distance [, directed] [, driving_side] [, details] [, equicost])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Using defaults

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of the query.

```
pgr_withPointsDD(edges_sql, points_sql, start_vid, distance)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example:
From point 1 with agg_cost <= 3.8

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of the query.

```
SELECT * FROM pgr_withPointsDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 3.8);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |    0 |        0
   2 |    1 |    1 |  0.4 |      0.4
   3 |    2 |    1 |  0.6 |      0.6
   4 |    5 |    4 |    1 |      1.6
   5 |    6 |    8 |    1 |      2.6
   6 |    8 |    7 |    1 |      2.6
   7 |   10 |   10 |    1 |      2.6
   8 |    7 |    6 |    1 |      3.6
   9 |    9 |    9 |    1 |      3.6
  10 |   11 |   11 |    1 |      3.6
  11 |   13 |   14 |    1 |      3.6
(11 rows)
```

Single vertex

Finds the driving distance depending on the optional parameters setup.

```
pgr_withPointsDD(edges_sql, points_sql, from_vid,  distance [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example:
Right side driving topology, from point 1 with agg_cost <= 3.8

```
SELECT * FROM pgr_withPointsDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 3.8,
    driving_side := 'r',
    details := true);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |    0 |        0
   2 |    1 |    1 |  0.4 |      0.4
   3 |    2 |    1 |    1 |      1.4
   4 |   -6 |    4 |  0.7 |      2.1
   5 |    5 |    4 |  0.3 |      2.4
   6 |    6 |    8 |    1 |      3.4
   7 |    8 |    7 |    1 |      3.4
   8 |   10 |   10 |    1 |      3.4
(8 rows)
```

Multiple vertices

Finds the driving distance depending on the optional parameters setup.

```
pgr_withPointsDD(edges_sql, points_sql, from_vids, distance [, directed] [, driving_side] [, details] [, equicost])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Parameters

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
points_sql	TEXT	Points SQL query as described above.
start_vid	ANY-INTEGER	Starting point id
distance	ANY-NUMERICAL	Distance from the start_pid
directed	BOOLEAN	(optional). When `false` the graph is considered as Undirected. Default is `true` which considers the graph as Directed.
driving_side	CHAR	(optional). Value in ['b', 'r', 'l', NULL] indicating if the driving side is: • In the right or left or • If it doesn't matter with 'b' or NULL. • If column not present 'b' is considered.
details	BOOLEAN	(optional). When `true` the results will include the driving distance to the points with in the `distance`. Default is `false` which ignores other points of the points_sql.
equicost	BOOLEAN	(optional). When `true` the nodes will only appear in the closest start_v list. Default is `false` which resembles several calls using the single starting point signatures. Tie brakes are arbitrary.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Description of the Points SQL query

points_sql:

an SQL query, which should return a set of rows with the following columns:

Column	Type	Description
pid	ANY-INTEGER	(optional) Identifier of the point. • If column present, it can not be NULL. • If column not present, a sequential identifier will be given automatically.
edge_id	ANY-INTEGER	Identifier of the “closest” edge to the point.
fraction	ANY-NUMERICAL	Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	(optional) Value in [‘b’, ‘r’, ‘l’, NULL] indicating if the point is: • In the right, left of the edge or • If it doesn’t matter with ‘b’ or NULL. • If column not present ‘b’ is considered.

Where:

ANY-INTEGER:
smallint, int, bigint
ANY-NUMERICAL:
smallint, int, bigint, real, float

Result Columns

Column	Type	Description
seq	INT	row sequence.
node	BIGINT	Identifier of the node within the Distance from start_pid. If details =: true a negative value is the identifier of a point.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. • -1 when start_vid = node.
cost	FLOAT	Cost to traverse edge. • 0 when start_vid = node.
agg_cost	FLOAT	Aggregate cost from start_vid to node. • 0 when start_vid = node.

Additional Examples

Examples for queries marked as directed with cost and reverse_cost columns.

The examples in this section use the following **Network for queries marked as directed and cost and reverse_cost columns are used**

Example:
Left side driving topology from point 1 with agg_cost <= 3.8, with details

```
SELECT * FROM pgr_withPointsDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 3.8,
    driving_side := 'l',
    details := true);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |    0 |        0
   2 |    2 |    1 |  0.6 |      0.6
   3 |   -6 |    4 |  0.7 |      1.3
   4 |    5 |    4 |  0.3 |      1.6
   5 |    1 |    1 |    1 |      1.6
   6 |    6 |    8 |    1 |      2.6
   7 |    8 |    7 |    1 |      2.6
   8 |   10 |   10 |    1 |      2.6
   9 |   -3 |   12 |  0.6 |      3.2
  10 |   -4 |    6 |  0.7 |      3.3
  11 |    7 |    6 |  0.3 |      3.6
  12 |    9 |    9 |    1 |      3.6
  13 |   11 |   11 |    1 |      3.6
  14 |   13 |   14 |    1 |      3.6
(14 rows)
```

Example:
From point 1 with agg_cost <= 3.8, does not matter driving side, with details

```
SELECT * FROM pgr_withPointsDD(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 3.8,
    driving_side := 'b',
    details := true);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |    0 |        0
   2 |    1 |    1 |  0.4 |      0.4
   3 |    2 |    1 |  0.6 |      0.6
   4 |   -6 |    4 |  0.7 |      1.3
   5 |    5 |    4 |  0.3 |      1.6
   6 |    6 |    8 |    1 |      2.6
   7 |    8 |    7 |    1 |      2.6
   8 |   10 |   10 |    1 |      2.6
   9 |   -3 |   12 |  0.6 |      3.2
  10 |   -4 |    6 |  0.7 |      3.3
  11 |    7 |    6 |  0.3 |      3.6
  12 |    9 |    9 |    1 |      3.6
  13 |   11 |   11 |    1 |      3.6
  14 |   13 |   14 |    1 |      3.6
(14 rows)
```

The queries use the **Sample Data** network.

See Also

- **pgr_drivingDistance** - Driving distance using dijkstra.
- **pgr_alphaShape** - Alpha shape computation.

Indices and tables

- **Index**
- **Search Page**

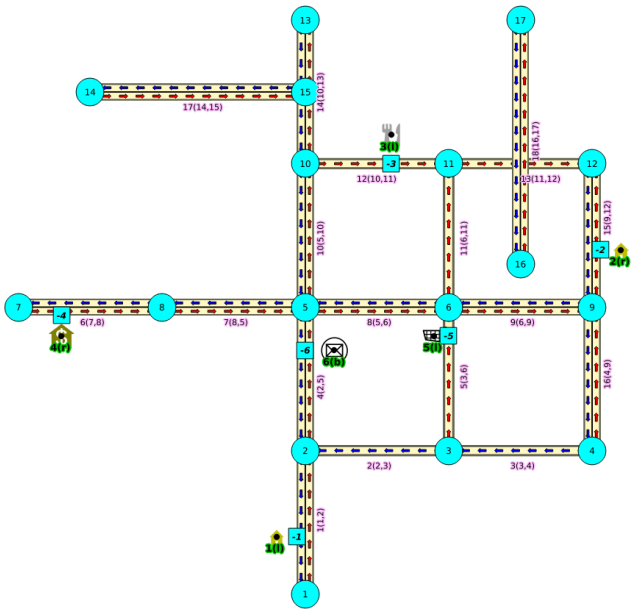
Previous versions of this page

- **Supported versions:** current(**3.0**) **2.6**
- **Unsupported versions:** **2.5** **2.4** **2.3** **2.2**

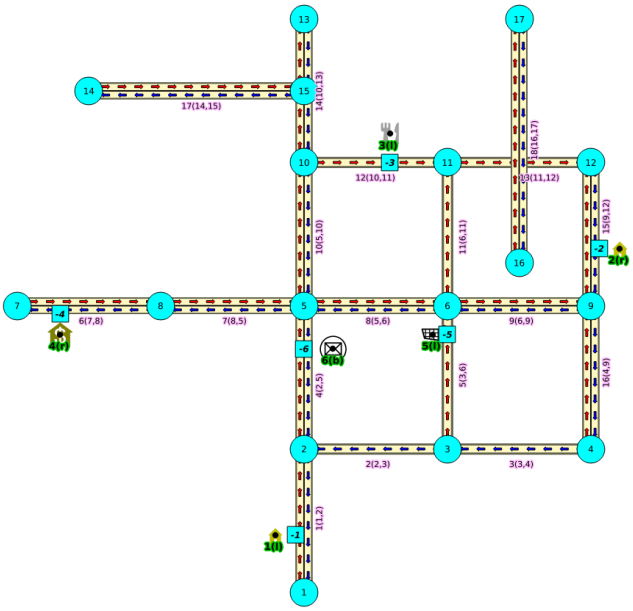
Images

The squared vertices are the temporary vertices, The temporary vertices are added according to the driving side, The following images visually show the differences on how depending on the driving side the data is interpreted.

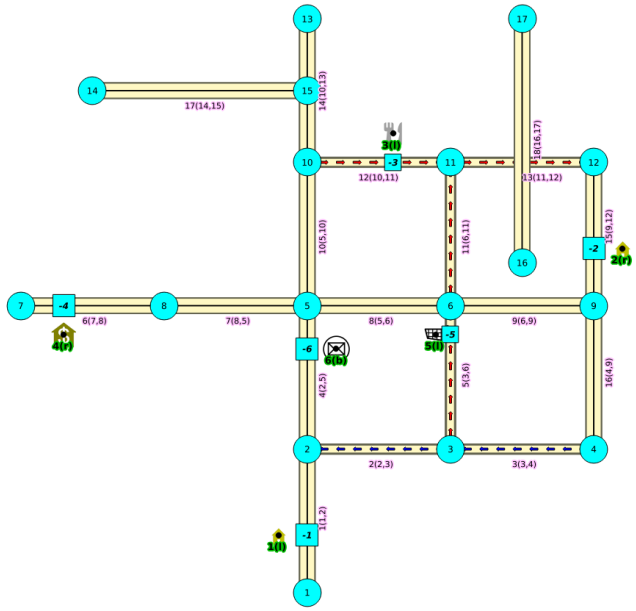
Right driving side



Left driving side



doesn’t matter the driving side



Introduction

This family of functions was thought for routing vehicles, but might as well work for some other application that we can not think of.

The with points family of function give you the ability to route between arbitrary points located outside the original graph.

When given a point identified with a *pid* that its being mapped to and edge with an identifier*edge_id*, with a *fraction* along that edge (from the source to the target of the edge) and some additional information about which *side* of the edge the point is on, then routing from arbitrary points more accurately reflect routing vehicles in road networks,

I talk about a family of functions because it includes different functionalities.

- pgr_withPoints is pgr_dijkstra based
- pgr_withPointsCost is pgr_dijkstraCost based
- pgr_withPointsKSP is pgr_ksp based
- pgr_withPointsDD is pgr_drivingDistance based

In all this functions we have to take care of as many aspects as possible:

- Must work for routing:
  - Cars (directed graph)
  - Pedestrians (undirected graph)
- Arriving at the point:
  - In either side of the street.

    - Compulsory arrival on the side of the street where the point is located.
- Countries with:
    - Right side driving
    - Left side driving
- Some points are:
    - Permanent, for example the set of points of clients stored in a table in the data base
    - Temporal, for example points given through a web application
- The numbering of the points are handled with negative sign.
    - Original point identifiers are to be positive.
    - Transformation to negative is done internally.
    - For results for involving vertices identifiers
        - positive sign is a vertex of the original graph
        - negative sign is a point of the temporary points

The reason for doing this is to avoid confusion when there is a vertex with the same number as identifier as the points identifier.

Graph & edges

- Let G_d(V,E) where V is the set of vertices and E is the set of edges be the original directed graph.
    - An edge of the original *edges_sql* is (id, source, target, cost, reverse_cost) will generate internally
        - (id, source, target, cost)
        - (id, target, source, reverse_cost)

Point Definition

- A point is defined by the quadruplet: (pid, eid, fraction, side)
    - **pid** is the point identifier
    - **eid** is an edge id of the *edges_sql*
    - **fraction** represents where the edge *eid* will be cut.
    - **side** Indicates the side of the edge where the point is located.

Creating Temporary Vertices in the Graph

For edge (15, 9,12 10, 20), & lets insert point (2, 12, 0.3, r)

On a right hand side driving network

From first image above:

- We can arrive to the point only via vertex 9.
- It only affects the edge (15, 9,12, 10) so that edge is removed.
- Edge (15, 12,9, 20) is kept.
- Create new edges:
    - (15, 9,-1, 3) edge from vertex 9 to point 1 has cost 3
    - (15, -1,12, 7) edge from point 1 to vertex 12 has cost 7

On a left hand side driving network

From second image above:

- We can arrive to the point only via vertex 12.
- It only affects the edge (15, 12,9 20) so that edge is removed.
- Edge (15, 9,12, 10) is kept.
- Create new edges:
    - (15, 12,-1, 14) edge from vertex 12 to point 1 has cost 14
    - (15, -1,9, 6) edge from point 1 to vertex 9 has cost 6

Remember:
that fraction is from vertex 9 to vertex 12

When driving side does not matter

From third image above:

- We can arrive to the point either via vertex 12 or via vertex 9
- Edge (15, 12,9 20) is removed.
- Edge (15, 9,12, 10) is removed.
- Create new edges:
    - (15, 12,-1, 14) edge from vertex 12 to point 1 has cost 14
    - (15, -1,9, 6) edge from point 1 to vertex 9 has cost 6

- (15, 9,-1, 3) edge from vertex 9 to point 1 has cost 3
- (15, -1,12, 7) edge from point 1 to vertex 12 has cost 7

See Also

Indices and tables

- **Index**
- **Search Page**

See Also

- **Experimental Functions**

Indices and tables

- **Index**
- **Search Page**

Experimental Functions

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Families

Flow - Family of functions

- **pgr_maxFlowMinCost - Experimental** - Details of flow and cost on edges.
- **pgr_maxFlowMinCost_Cost - Experimental** - Only the Min Cost calculation.

Chinese Postman Problem - Family of functions (Experimental)

- **pgr_chinesePostman - Experimental**
- **pgr_chinesePostmanCost - Experimental**

Topology - Family of Functions

- **pgr_extractVertices – Experimental** - Extracts vertices information based on the source and target.

Transformation - Family of functions (Experimental)

- **pgr_lineGraph - Experimental** - Transformation algorithm for generating a Line Graph.
- **pgr_lineGraphFull - Experimental** - Transformation algorithm for generating a Line Graph out of each vertex in the input graph.

Chinese Postman Problem - Family of functions (Experimental)

- **pgr_chinesePostman - Experimental**
- **pgr_chinesePostmanCost - Experimental**

pgr_chinesePostman - Experimental

`pgr_chinesePostman` — Calculates the shortest circuit path which contains every edge in a directed graph and starts and ends on the same vertex.

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions** current(**3.0**)

Description

The main characteristics are:

- Process is done only on edges with **positive** costs.
- Running time: O(E * (E + V * logV))
- Graph must be connected.
- Returns `EMPTY SET` on a disconnected graph

Signatures

```
pgr_chinesePostman(edges_sql)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

```
SELECT * FROM pgr_chinesePostman(
    'SELECT id,
     source, target,
     cost, reverse_cost FROM edge_table where id < 17'
);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |   34 |       34
   2 |    3 |    2 |    1 |        1
   3 |    2 |    1 |    1 |        2
   4 |    1 |    1 |    1 |        3
   5 |    2 |    4 |    1 |        4
   6 |    5 |    4 |    1 |        5
   7 |    2 |    4 |    1 |        6
   8 |    5 |    7 |    1 |        7
   9 |    8 |    6 |    1 |        8
  10 |    7 |    6 |    1 |        9
  11 |    8 |    7 |    1 |       10
  12 |    5 |    8 |    1 |       11
  13 |    6 |    8 |    1 |       12
  14 |    5 |   10 |    1 |       13
  15 |   10 |   10 |    1 |       14
  16 |    5 |   10 |    1 |       15
  17 |   10 |   14 |    1 |       16
  18 |   13 |   14 |    1 |       17
  19 |   10 |   12 |    1 |       18
  20 |   11 |   13 |    1 |       19
  21 |   12 |   15 |    1 |       20
  22 |    9 |    9 |    1 |       21
  23 |    6 |    9 |    1 |       22
  24 |    9 |   15 |    1 |       23
  25 |   12 |   15 |    1 |       24
  26 |    9 |   16 |    1 |       25
  27 |    4 |   16 |    1 |       26
  28 |    9 |   16 |    1 |       27
  29 |    4 |    3 |    1 |       28
  30 |    3 |    5 |    1 |       29
  31 |    6 |   11 |    1 |       30
  32 |   11 |   13 |    1 |       31
  33 |   12 |   15 |    1 |       32
  34 |    9 |   16 |    1 |       33
  35 |    4 |   -1 |    0 |       34
(35 rows)
```

Parameters

Column	Type	Default	Description
edges_sql	TEXT		The edges SQL query as described in **Inner query**.

Inner query

An Edges SQL that represents a **directed** graph with the following columns

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.

Column	Type	Description
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

See Also

- **Chinese Postman Problem - Family of functions (Experimental)**

Indices and tables

- **Index**
- **Search Page**

pgr_chinesePostmanCost - Experimental

`pgr_chinesePostmanCost` — Calculates the minimum costs of a circuit path which contains every edge in a directed graph and starts and ends on the same vertex.

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions** current(**3.0**)

Description

The main characteristics are:

- Process is done only on edges with **positive** costs.
- Running time: O(E * (E + V * logV))
- Graph must be connected.
- [TBD] Return value when the graph if disconnected

Signatures

```
pgr_chinesePostmanCost(edges_sql)
RETURNS FLOAT
```

Example:

```
SELECT * FROM pgr_chinesePostmanCost(
    'SELECT id,
     source, target,
     cost, reverse_cost FROM edge_table where id < 17'
);
 pgr_chinesepostmancost
------------------------
                     34
(1 row)
```

Parameters

Column	Type	Default	Description
edges_sql	TEXT		The edges SQL query as described in **Inner query**.

Inner query

An Edges SQL that represents a **directed** graph with the following columns

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Type	Description
FLOAT	Minimum costs of a circuit path.

See Also

- **Chinese Postman Problem - Family of functions (Experimental)**

Indices and tables

- **Index**
- **Search Page**

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.

- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Versions of this page

- **Supported versions:** current(**3.0**)

Description

The main characteristics are:

- Process is done only on edges with **positive** costs.
- Running time: O(E * (E + V * logV))
- Graph must be connected.

Parameters

Column	Type	Default	Description
edges_sql	TEXT		The edges SQL query as described in **Inner query**.

Inner query

An Edges SQL that represents a **directed** graph with the following columns

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

See Also

Indices and tables

- **Index**
- **Search Page**

categories

Vehicle Routing Functions - Category (Experimental)

- Pickup and delivery problem
  - **pgr_pickDeliver - Experimental** - Pickup & Delivery using a Cost Matrix
  - **pgr_pickDeliverEuclidean - Experimental** - Pickup & Delivery with Euclidean distances
- Distribution problem

- **pgr_vrpOneDepot - Experimental** - From a single depot, distributes orders

Vehicle Routing Functions - Category (Experimental)

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

- Pickup and delivery problem
  - **pgr_pickDeliver - Experimental** - Pickup & Delivery using a Cost Matrix
  - **pgr_pickDeliverEuclidean - Experimental** - Pickup & Delivery with Euclidean distances
- Distribution problem
  - **pgr_vrpOneDepot - Experimental** - From a single depot, distributes orders

Contents

pgr_pickDeliver - Experimental

`pgr_pickDeliver` - Pickup and delivery Vehicle Routing Problem

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Synopsis

Problem: Distribute and optimize the pickup-delivery pairs into a fleet of vehicles.

- Optimization problem is NP-hard.
- pickup and Delivery with time windows.
- All vehicles are equal.
  - Same Starting location.
  - Same Ending location which is the same as Starting location.
  - All vehicles travel at the same speed.
- A customer is for doing a pickup or doing a deliver.
  - has an open time.
  - has a closing time.
  - has a service time.
  - has an (x, y) location.
- There is a customer where to deliver a pickup.
  - travel time between customers is distance / speed
  - pickup and delivery pair is done with the same vehicle.
  - A pickup is done before the delivery.

Characteristics

- All trucks depart at time 0.
- No multiple time windows for a location.
- Less vehicle used is considered better.
- Less total duration is better.
- Less wait time is better.
- the algorithm will raise an exception when
  - If there is a pickup-deliver pair than violates time window
  - The speed, max_cycles, ma_capacity have illegal values
- Six different initial will be optimized - the best solution found will be result

Signature

```
pgr_pickDeliver(orders_sql, vehicles_sql, matrix_sql [, factor, max_cycles, initial_sol])
RETURNS SET OF (seq, vehicle_number, vehicle_id, stop, order_id, stop_type, cargo,
                travel_time, arrival_time, wait_time, service_time, departure_time)
```

Parameters

The parameters are:

```
orders_sql, vehicles_sql, matrix_sql [, factor, max_cycles, initial_sol]
```

Column	Type	Default	Description
orders_sql	TEXT		**Pick & Deliver Orders SQL** query contianing the orders to be processed.
vehicles_sql	TEXT		**Pick & Deliver Vehicles SQL** query containing the vehicles to be used.
matrix_sql	TEXT		**Pick & Deliver Matrix SQL** query containing the distance or travel times.
factor	NUMERIC	1	Travel time multiplier. See **Factor Handling**
max_cycles	INTEGER	10	Maximum number of cycles to perform on the optimization.
initial_sol	INTEGER	4	Initial solution to be used. - 1 One order per truck - 2 Push front order. - 3 Push back order. - 4 Optimize insert. - 5 Push back order that allows more orders to be inserted at the back - 6 Push front order that allows more orders to be inserted at the front

Pick & Deliver Orders SQL

A *SELECT* statement that returns the following columns:

```
id, demand
p_node_id, p_open, p_close, [p_service, ]
d_node_id, d_open, d_close, [d_service, ]
```

where:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the pick-delivery order pair.
demand	ANY-NUMERICAL		Number of units in the order
p_open	ANY-NUMERICAL		The time, relative to 0, when the pickup location opens.
p_close	ANY-NUMERICAL		The time, relative to 0, when the pickup location closes.
d_service	ANY-NUMERICAL	0	The duration of the loading at the pickup location.
d_open	ANY-NUMERICAL		The time, relative to 0, when the delivery location opens.
d_close	ANY-NUMERICAL		The time, relative to 0, when the delivery location closes.
d_service	ANY-NUMERICAL	0	The duration of the loading at the delivery location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Description
p_node_id	ANY-INTEGER	The node identifier of the pickup, must match a node identifier in the matrix table.
d_node_id	ANY-INTEGER	The node identifier of the delivery, must match a node identifier in the matrix table.

Pick & Deliver Vehicles SQL

A *SELECT* statement that returns the following columns:

```
id, capacity
start_node_id, start_open, start_close [, start_service, ]
[ end_node_id, end_open, end_close, end_service ]
```

where:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the pick-delivery order pair.
capacity	ANY-NUMERICAL		Number of units in the order
speed	ANY-NUMERICAL	*1*	Average speed of the vehicle.
start_open	ANY-NUMERICAL		The time, relative to 0, when the starting location opens.
start_close	ANY-NUMERICAL		The time, relative to 0, when the starting location closes.
start_service	ANY-NUMERICAL	*0*	The duration of the loading at the starting location.
end_open	ANY-NUMERICAL	*start_open*	The time, relative to 0, when the ending location opens.
end_close	ANY-NUMERICAL	*start_close*	The time, relative to 0, when the ending location closes.
end_service	ANY-NUMERICAL	*start_service*	The duration of the loading at the ending location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Default	Description
start_node_id	ANY-INTEGER		The node identifier of the starting location, must match a node identifier in the matrix table.
end_node_id	ANY-INTEGER	*start_node_id*	The node identifier of the ending location, must match a node identifier in the matrix table.

Pick & Deliver Matrix SQL

A *SELECT* statement that returns the following columns:

Warning

TODO

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Example

This example use the following data: TODO put link

```
SELECT * FROM pgr_pickDeliver(
    $$ SELECT * FROM orders ORDER BY id $$,
    $$ SELECT * FROM vehicles $$,
    $$ SELECT * from pgr_dijkstraCostMatrix(
        ' SELECT * FROM edge_table ', ARRAY[3, 4, 5, 8, 9, 11])
    $$
);
 seq | vehicle_seq | vehicle_id | stop_seq | stop_type | stop_id | order_id | cargo | travel_time | arrival_time | wait_time | service_time | departure_time
-----+-------------+------------+----------+-----------+---------+----------+-------+-------------+--------------+-----------+--------------+----------------
   1 |           1 |          1 |        1 |         1 |       6 |       -1 |     0 |           0 |            0 |         0 |            0 |              0
   2 |           1 |          1 |        2 |         2 |       5 |        3 |    30 |           1 |            1 |         1 |            3 |              5
   3 |           1 |          1 |        3 |         3 |      11 |        3 |     0 |           2 |            7 |         0 |            3 |             10
   4 |           1 |          1 |        4 |         2 |       9 |        2 |    20 |           2 |           12 |         0 |            2 |             14
   5 |           1 |          1 |        5 |         3 |       4 |        2 |     0 |           1 |           15 |         0 |            3 |             18
   6 |           1 |          1 |        6 |         6 |       6 |       -1 |     0 |           4 |           22 |         0 |            0 |             22
   7 |           2 |          1 |        1 |         1 |       6 |       -1 |     0 |           0 |            0 |         0 |            0 |              0
   8 |           2 |          1 |        2 |         2 |       3 |        1 |    10 |           5 |            5 |         0 |            3 |              8
   9 |           2 |          1 |        3 |         3 |       8 |        1 |     0 |           3 |           11 |         0 |            3 |             14
  10 |           2 |          1 |        4 |         6 |       6 |       -1 |     0 |           0 |           14 |         0 |            0 |             14
  11 |          -2 |          0 |        0 |        -1 |      -1 |       -1 |    -1 |          18 |           -1 |         1 |           17 |             36
(11 rows)
```

See Also

- **Vehicle Routing Functions - Category (Experimental)**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_pickDeliverEuclidean - Experimental

`pgr_pickDeliverEuclidean` - Pickup and delivery Vehicle Routing Problem

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL

- Name might change.
- Signature might change.
- Functionality might change.
- pgTap tests might be missing.
- Might need c/c++ coding.
- May lack documentation.
- Documentation if any might need to be rewritten.
- Documentation examples might need to be automatically generated.
- Might need a lot of feedback from the comunity.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - Replaces `pgr_gsoc_vrppdtw`
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6** **2.5** **2.4** **2.3** **2.2** **2.1**

Synopsis

Problem: Distribute and optimize the pickup-delivery pairs into a fleet of vehicles.

- Optimization problem is NP-hard.
- Pickup and Delivery:
  - capacitated
  - with time windows.
- The vehicles
  - have (x, y) start and ending locations.
  - have a start and ending service times.
  - have opening and closing times for the start and ending locations.
- An order is for doing a pickup and a a deliver.
  - has (x, y) pickup and delivery locations.
  - has opening and closing times for the pickup and delivery locations.
  - has a pickup and deliver service times.
- There is a customer where to deliver a pickup.
  - travel time between customers is distance / speed
  - pickup and delivery pair is done with the same vehicle.
  - A pickup is done before the delivery.

Characteristics

- No multiple time windows for a location.
- Less vehicle used is considered better.
- Less total duration is better.
- Less wait time is better.
- Six different optional different initial solutions
  - the best solution found will be result

Signature

```
pgr_pickDeliverEuclidean(orders_sql, vehicles_sql [,factor, max_cycles, initial_sol])
RETURNS SET OF (seq, vehicle_seq, vehicle_id, stop_seq, stop_type, order_id,
    cargo, travel_time, arrival_time, wait_time, service_time, departure_time)
```

Parameters

The parameters are:

```
orders_sql, vehicles_sql [,factor, max_cycles, initial_sol]
```

Where:

Column	Type	Default	Description
orders_sql	TEXT		**Pick & Deliver Orders SQL** query containing the orders to be processed.

Column	Type	Default	Description
vehicles_sql	TEXT		**Pick & Deliver Vehicles SQL** query containing the vehicles to be used.
factor	NUMERIC	1	(Optional) Travel time multiplier. See **Factor Handling**
max_cycles	INTEGER	10	(Optional) Maximum number of cycles to perform on the optimization.
initial_sol	INTEGER	4	(Optional) Initial solution to be used. • 1 One order per truck • 2 Push front order. • 3 Push back order. • 4 Optimize insert. • 5 Push back order that allows more orders to be inserted at the back • 6 Push front order that allows more orders to be inserted at the front

Pick & Deliver Orders SQL

A *SELECT* statement that returns the following columns:

```
id, demand
p_x, p_y, p_open, p_close, [p_service, ]
d_x, d_y, d_open, d_close, [d_service, ]
```

Where:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the pick-delivery order pair.
demand	ANY-NUMERICAL		Number of units in the order
p_open	ANY-NUMERICAL		The time, relative to 0, when the pickup location opens.
p_close	ANY-NUMERICAL		The time, relative to 0, when the pickup location closes.
d_service	ANY-NUMERICAL	0	The duration of the loading at the pickup location.
d_open	ANY-NUMERICAL		The time, relative to 0, when the delivery location opens.
d_close	ANY-NUMERICAL		The time, relative to 0, when the delivery location closes.
d_service	ANY-NUMERICAL	0	The duration of the loading at the delivery location.

For the euclidean implementation, pick up and delivery (x,y) locations are needed:

Column	Type	Description
p_x	ANY-NUMERICAL	x value of the pick up location
p_y	ANY-NUMERICAL	y value of the pick up location
d_x	ANY-NUMERICAL	x value of the delivery location
d_y	ANY-NUMERICAL	y value of the delivery location

Pick & Deliver Vehicles SQL

A *SELECT* statement that returns the following columns:

```
id, capacity
start_x, start_y, start_open, start_close [, start_service, ]
[ end_x, end_y, end_open, end_close, end_service ]
```

where:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the pick-delivery order pair.
capacity	ANY-NUMERICAL		Number of units in the order
speed	ANY-NUMERICAL	*1*	Average speed of the vehicle.
start_open	ANY-NUMERICAL		The time, relative to 0, when the starting location opens.
start_close	ANY-NUMERICAL		The time, relative to 0, when the starting location closes.
start_service	ANY-NUMERICAL	*0*	The duration of the loading at the starting location.
end_open	ANY-NUMERICAL	*start_open*	The time, relative to 0, when the ending location opens.
end_close	ANY-NUMERICAL	*start_close*	The time, relative to 0, when the ending location closes.
end_service	ANY-NUMERICAL	*start_service*	The duration of the loading at the ending location.

For the euclidean implementation, starting and ending(x,y) locations are needed:

Column	Type	Default	Description

Column	Type	Default	Description
start_x	ANY-NUMERICAL		x value of the coordinate of the starting location.
start_y	ANY-NUMERICAL		y value of the coordinate of the starting location.
end_x	ANY-NUMERICAL	*start_x*	x value of the coordinate of the ending location.
end_y	ANY-NUMERICAL	*start_y*	y value of the coordinate of the ending location.

Description of the result (TODO Disussion: Euclidean & Matrix)

```
RETURNS SET OF
   (seq, vehicle_seq, vehicle_id, stop_seq, stop_type,
      travel_time, arrival_time, wait_time, service_time,  departure_time)
   UNION
   (summary row)
```

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
vehicle_seq	INTEGER	Sequential value starting from **1** for current vehicles. The n_{th} vehicle in the solution.
vehicle_id	BIGINT	Current vehicle identifier.
stop_seq	INTEGER	Sequential value starting from **1** for the stops made by the current vehicle. Them_{th} stop of the current vehicle.
stop_type	INTEGER	Kind of stop location the vehicle is at: • `1`: Starting location • `2`: Pickup location • `3`: Delivery location • `6`: Ending location
order_id	BIGINT	Pickup-Delivery order pair identifier. • `-1`: When no order is involved on the current stop location.
cargo	FLOAT	Cargo units of the vehicle when leaving the stop.
travel_time	FLOAT	Travel time from previous `stop_seq` to current `stop_seq`. • `0` When `stop_type = 1`
arrival_time	FLOAT	Previous `departure_time` plus current `travel_time`.
wait_time	FLOAT	Time spent waiting for current *location* to open.
service_time	FLOAT	Service time at current *location*.
departure_time	FLOAT	arrival_time + wait_time + service_time. • When `stop_type = 6` has the *total_time* used for the current vehicle.

Summary Row

Warning

TODO: Review the summary

Column	Type	Description
seq	INTEGER	Continues the Sequential value
vehicle_seq	INTEGER	`-2` to indicate is a summary row
vehicle_id	BIGINT	*Total Capacity Violations* in the solution.
stop_seq	INTEGER	*Total Time Window Violations* in the solution.
stop_type	INTEGER	`-1`
order_id	BIGINT	`-1`
cargo	FLOAT	`-1`
travel_time	FLOAT	*total_travel_time* The sum of all the*travel_time*
arrival_time	FLOAT	`-1`
wait_time	FLOAT	*total_waiting_time* The sum of all the*wait_time*
service_time	FLOAT	*total_service_time* The sum of all the*service_time*
departure_time	FLOAT	*total_solution_time* = total_travel_time + total_wait_time + total_service_time.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Example

This example use the following data: TODO put link

```
SELECT * FROM pgr_pickDeliverEuclidean(
    'SELECT * FROM orders ORDER BY id',
    'SELECT * from vehicles'
);
 seq | vehicle_seq | vehicle_id | stop_seq | stop_type | order_id | cargo |   travel_time   | arrival_time   | wait_time | service_time | departure_time
-----+-------------+------------+----------+-----------+----------+-------+-----------------+----------------+-----------+--------------+----------------
   1 |           1 |          1 |        1 |         1 |       -1 |     0 |               0 |              0 |         0 |            0 |              0
   2 |           1 |          1 |        2 |         2 |        3 |    30 |               1 |              1 |         1 |            3 |              5
   3 |           1 |          1 |        3 |         3 |        3 |     0 | 1.41421356237 | 6.41421356237 |         0 |            3 |  9.41421356237
   4 |           1 |          1 |        4 |         2 |        2 |    20 | 1.41421356237 | 10.8284271247 |         0 |            2 |  12.8284271247
   5 |           1 |          1 |        5 |         3 |        2 |     0 |               1 | 13.8284271247 |         0 |            3 |  16.8284271247
   6 |           1 |          1 |        6 |         6 |       -1 |     0 | 1.41421356237 | 18.2426406871 |         0 |            0 |  18.2426406871
   7 |           2 |          1 |        1 |         1 |       -1 |     0 |               0 |              0 |         0 |            0 |              0
   8 |           2 |          1 |        2 |         2 |        1 |    10 |               1 |              1 |         1 |            3 |              5
   9 |           2 |          1 |        3 |         3 |        1 |     0 |  2.2360679775 |  7.2360679775 |         0 |            3 |  10.2360679775
  10 |           2 |          1 |        4 |         6 |       -1 |     0 |               2 | 12.2360679775 |         0 |            0 |  12.2360679775
  11 |          -2 |          0 |        0 |        -1 |       -1 |    -1 | 11.4787086646 |             -1 |         2 |           17 |  30.4787086646
(11 rows)
```

See Also

- **Vehicle Routing Functions - Category (Experimental)**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_vrpOneDepot - Experimental

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

No documentation available

Availability

- Version 2.1.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6 2.5 2.4 2.3 2.2 2.1**
- **TBD**

Description

- TBD

Signatures

- TBD

Parameters

- TBD

Inner query

- TBD

Result Columns

- TBD

Additional Example:

```
BEGIN;
BEGIN
SET client_min_messages TO NOTICE;
SET
SELECT * FROM pgr_vrpOneDepot(
    'SELECT * FROM solomon_100_RC_101',
    'SELECT * FROM vrp_vehicles',
    'SELECT * FROM vrp_distance',
    1);
 oid | opos | vid | tarrival | tdepart
-----+------+-----+----------+---------
  -1 |    1 |   1 |        0 |       0
  46 |    2 |   1 |        0 |       0
  47 |    3 |   1 |        0 |       0
   9 |    4 |   1 |        0 |       0
 101 |    5 |   1 |        0 |       0
  71 |    6 |   1 |        0 |       0
   8 |    7 |   1 |        0 |       0
   6 |    8 |   1 |        0 |       0
   5 |    9 |   1 |        0 |       0
   4 |   10 |   1 |        0 |       0
   2 |   11 |   1 |        0 |       0
   6 |   12 |   1 |       40 |      51
  46 |   13 |   1 |       54 |      64
   8 |   14 |   1 |       73 |      89
   9 |   15 |   1 |       94 |     104
  47 |   16 |   1 |      107 |     123
   4 |   17 |   1 |      129 |     139
   2 |   18 |   1 |      142 |     155
   5 |   19 |   1 |      162 |     172
  71 |   20 |   1 |      191 |     201
 101 |   21 |   1 |      205 |     215
  -1 |   22 |   1 |      234 |     234
  -1 |    1 |   3 |        0 |       0
  16 |    2 |   3 |        0 |       0
  10 |    3 |   3 |        0 |       0
  81 |    4 |   3 |        0 |       0
  48 |    5 |   3 |        0 |       0
  15 |    6 |   3 |        0 |       0
  14 |    7 |   3 |        0 |       0
  17 |    8 |   3 |        0 |       0
  11 |    9 |   3 |        0 |       0
  18 |   10 |   3 |        0 |       0
  15 |   11 |   3 |       35 |      45
  48 |   12 |   3 |       48 |      58
  17 |   13 |   3 |       64 |      82
  16 |   14 |   3 |       84 |      94
  10 |   15 |   3 |      103 |     113
  11 |   16 |   3 |      118 |     129
  14 |   17 |   3 |      136 |     152
  18 |   18 |   3 |      163 |     173
  81 |   19 |   3 |      220 |     230
  -1 |   20 |   3 |      238 |     238
  -1 |    1 |  12 |        0 |       0
  49 |    2 |  12 |        0 |       0
  26 |    3 |  12 |        0 |       0
  21 |    4 |  12 |        0 |       0
  65 |    5 |  12 |        0 |       0
  23 |    6 |  12 |        0 |       0
  50 |    7 |  12 |        0 |       0
  96 |    8 |  12 |        0 |       0
```

93	9	12	0	0
52	10	12	0	0
93	11	12	15	25
96	12	12	31	44
65	13	12	57	67
52	14	12	81	93
50	15	12	108	118
23	16	12	122	132
21	17	12	134	144
26	18	12	156	166
49	19	12	173	183
-1	20	12	228	228
-1	1	9	0	0
73	2	9	0	0
92	3	9	0	0
55	4	9	0	0
44	5	9	0	0
42	6	9	0	0
41	7	9	0	0
39	8	9	0	0
37	9	9	0	0
73	10	9	27	40
37	11	9	60	70
39	12	9	76	86
41	13	9	91	101
42	14	9	112	122
44	15	9	132	142
55	16	9	167	177
92	17	9	196	206
-1	18	9	218	218
-1	1	5	0	0
66	2	5	0	0
78	3	5	0	0
25	4	5	0	0
24	5	5	0	0
22	6	5	0	0
20	7	5	0	0
19	8	5	0	0
66	9	5	11	21
24	10	5	56	75
20	11	5	81	91
22	12	5	97	107
19	13	5	111	121
25	14	5	132	158
78	15	5	177	187
-1	16	5	237	237
-1	1	7	0	0
32	2	7	0	0
30	3	7	0	0
28	4	7	0	0
94	5	7	0	0
33	6	7	0	0
35	7	7	0	0
27	8	7	0	0
28	9	7	57	72
30	10	7	77	87
32	11	7	89	99
27	12	7	108	125
35	13	7	136	146
33	14	7	152	162
94	15	7	196	206
-1	16	7	227	227
-1	1	4	0	0
84	2	4	0	0
91	3	4	0	0
53	4	4	0	0
75	5	4	0	0
59	6	4	0	0
100	7	4	0	0
84	8	4	19	45
53	9	4	57	67
100	10	4	72	87
91	11	4	103	113
75	12	4	145	155
59	13	4	166	176
-1	14	4	224	224
-1	1	10	0	0
56	2	10	0	0
40	3	10	0	0
69	4	10	0	0
7	5	10	0	0
45	6	10	0	0
43	7	10	0	0
43	8	10	34	44
40	9	10	49	59
45	10	10	64	74
7	11	10	112	122
69	12	10	148	158
56	13	10	168	178
-1	14	10	192	192
-1	1	6	0	0
70	2	6	0	0
99	3	6	0	0
87	4	6	0	0
58	5	6	0	0

```
  57 |   6 |  6 |       0 |      0
  67 |   7 |  6 |       0 |      0
  70 |   8 |  6 |       9 |     51
  99 |   9 |  6 |      56 |     66
  87 |  10 |  6 |      94 |    104
  58 |  11 |  6 |     113 |    123
  67 |  12 |  6 |     140 |    150
  57 |  13 |  6 |     159 |    169
  -1 |  14 |  6 |     187 |    187
  -1 |   1 | 14 |       0 |      0
  63 |   2 | 14 |       0 |      0
  86 |   3 | 14 |       0 |      0
  90 |   4 | 14 |       0 |      0
  85 |   5 | 14 |       0 |      0
  77 |   6 | 14 |       0 |      0
  63 |   7 | 14 |      29 |     62
  77 |   8 | 14 |      86 |     96
  86 |   9 | 14 |     107 |    117
  85 |  10 | 14 |     125 |    135
  90 |  11 | 14 |     161 |    171
  -1 |  12 | 14 |     224 |    224
  -1 |   1 |  8 |       0 |      0
  34 |   2 |  8 |       0 |      0
  31 |   3 |  8 |       0 |      0
  29 |   4 |  8 |       0 |      0
  38 |   5 |  8 |       0 |      0
  36 |   6 |  8 |       0 |      0
  34 |   7 |  8 |      51 |     61
  29 |   8 |  8 |      70 |     80
  31 |   9 |  8 |      84 |     94
  38 |  10 |  8 |     151 |    161
  36 |  11 |  8 |     165 |    175
  -1 |  12 |  8 |     219 |    219
  -1 |   1 |  2 |       0 |      0
  79 |   2 |  2 |       0 |      0
  61 |   3 |  2 |       0 |      0
  12 |   4 |  2 |       0 |      0
  13 |   5 |  2 |       0 |      0
  13 |   6 |  2 |      32 |     74
  12 |   7 |  2 |      79 |     89
  79 |   8 |  2 |     105 |    115
  61 |   9 |  2 |     123 |    165
  -1 |  10 |  2 |     192 |    192
  -1 |   1 | 11 |       0 |      0
  51 |   2 | 11 |       0 |      0
  62 |   3 | 11 |       0 |      0
  82 |   4 | 11 |       0 |      0
  95 |   5 | 11 |       0 |      0
  62 |   6 | 11 |      16 |     76
  82 |   7 | 11 |      84 |     96
  95 |   8 | 11 |     109 |    119
  51 |   9 | 11 |     139 |    149
  -1 |  10 | 11 |     184 |    184
  -1 |   1 | 13 |       0 |      0
  60 |   2 | 13 |       0 |      0
  76 |   3 | 13 |       0 |      0
  88 |   4 | 13 |       0 |      0
  98 |   5 | 13 |       0 |      0
  60 |   6 | 13 |      42 |     52
  76 |   7 | 13 |      68 |     84
  88 |   8 | 13 |     104 |    114
  98 |   9 | 13 |     124 |    134
  -1 |  10 | 13 |     182 |    182
  -1 |   1 | 20 |       0 |      0
  83 |   2 | 20 |       0 |      0
  80 |   3 | 20 |       0 |      0
  74 |   4 | 20 |       0 |      0
  89 |   5 | 20 |       0 |      0
  83 |   6 | 20 |      15 |     52
  89 |   7 | 20 |      67 |     77
  74 |   8 | 20 |      99 |    109
  80 |   9 | 20 |     118 |    128
  -1 |  10 | 20 |     167 |    167
  -1 |   1 | 15 |       0 |      0
  97 |   2 | 15 |       0 |      0
  64 |   3 | 15 |       0 |      0
  72 |   4 | 15 |       0 |      0
  68 |   5 | 15 |       0 |      0
  64 |   6 | 15 |      39 |     49
  68 |   7 | 15 |      71 |     81
  72 |   8 | 15 |      94 |    104
  97 |   9 | 15 |     114 |    142
  -1 |  10 | 15 |     158 |    158
  -1 |   1 | 17 |       0 |      0
  54 |   2 | 17 |       0 |      0
   3 |   3 | 17 |       0 |      0
   3 |   4 | 17 |      31 |     60
  54 |   5 | 17 |      85 |    101
  -1 |   6 | 17 |     121 |    121
  -1 |   0 |  0 |      -1 |   3432
(235 rows)

ROLLBACK;
ROLLBACK
```

Data

```
DROP TABLE IF EXISTS solomon_100_RC_101 cascade;
CREATE TABLE solomon_100_RC_101 (
    id integer NOT NULL PRIMARY KEY,
    order_unit integer,
    open_time integer,
    close_time integer,
    service_time integer,
    x float8,
    y float8
);

COPY solomon_100_RC_101
(id, x, y, order_unit, open_time, close_time, service_time) FROM stdin;
1 40.000000 50.000000 0 0 240 0
2 25.000000 85.000000 20 145 175 10
3 22.000000 75.000000 30 50 80 10
4 22.000000 85.000000 10 109 139 10
5 20.000000 80.000000 40 141 171 10
6 20.000000 85.000000 20 41 71 10
7 18.000000 75.000000 20 95 125 10
8 15.000000 75.000000 20 79 109 10
9 15.000000 80.000000 10 91 121 10
10 10.000000 35.000000 20 91 121 10
11 10.000000 40.000000 30 119 149 10
12 8.000000 40.000000 40 59 89 10
13 8.000000 45.000000 20 64 94 10
14 5.000000 35.000000 10 142 172 10
15 5.000000 45.000000 10 35 65 10
16 2.000000 40.000000 20 58 88 10
17 0.000000 40.000000 20 72 102 10
18 0.000000 45.000000 20 149 179 10
19 44.000000 5.000000 20 87 117 10
20 42.000000 10.000000 40 72 102 10
21 42.000000 15.000000 10 122 152 10
22 40.000000 5.000000 10 67 97 10
23 40.000000 15.000000 40 92 122 10
24 38.000000 5.000000 30 65 95 10
25 38.000000 15.000000 10 148 178 10
26 35.000000 5.000000 20 154 184 10
27 95.000000 30.000000 30 115 145 10
28 95.000000 35.000000 20 62 92 10
29 92.000000 30.000000 10 62 92 10
30 90.000000 35.000000 10 67 97 10
31 88.000000 30.000000 10 74 104 10
32 88.000000 35.000000 20 61 91 10
33 87.000000 30.000000 10 131 161 10
34 85.000000 25.000000 10 51 81 10
35 85.000000 35.000000 30 111 141 10
36 67.000000 85.000000 20 139 169 10
37 65.000000 85.000000 40 43 73 10
38 65.000000 82.000000 10 124 154 10
39 62.000000 80.000000 30 75 105 10
40 60.000000 80.000000 10 37 67 10
41 60.000000 85.000000 30 85 115 10
42 58.000000 75.000000 20 92 122 10
43 55.000000 80.000000 10 33 63 10
44 55.000000 85.000000 20 128 158 10
45 55.000000 82.000000 10 64 94 10
46 20.000000 82.000000 10 37 67 10
47 18.000000 80.000000 10 113 143 10
48 2.000000 45.000000 10 45 75 10
49 42.000000 5.000000 10 151 181 10
50 42.000000 12.000000 10 104 134 10
51 72.000000 35.000000 30 116 146 10
52 55.000000 20.000000 19 83 113 10
53 25.000000 30.000000 3 52 82 10
54 20.000000 50.000000 5 91 121 10
55 55.000000 60.000000 16 139 169 10
56 30.000000 60.000000 16 140 170 10
57 50.000000 35.000000 19 130 160 10
58 30.000000 25.000000 23 96 126 10
59 15.000000 10.000000 20 152 182 10
60 10.000000 20.000000 19 42 72 10
61 15.000000 60.000000 17 155 185 10
62 45.000000 65.000000 9 66 96 10
63 65.000000 35.000000 3 52 82 10
64 65.000000 20.000000 6 39 69 10
65 45.000000 30.000000 17 53 83 10
66 35.000000 40.000000 16 11 41 10
67 41.000000 37.000000 16 133 163 10
68 64.000000 42.000000 9 70 100 10
69 40.000000 60.000000 21 144 174 10
70 31.000000 52.000000 27 41 71 10
71 35.000000 69.000000 23 180 210 10
72 65.000000 55.000000 14 65 95 10
73 63.000000 65.000000 8 30 60 10
74 2.000000 60.000000 5 77 107 10
75 20.000000 20.000000 8 141 171 10
76 5.000000 5.000000 16 74 104 10
77 60.000000 12.000000 31 75 105 10
78 23.000000 3.000000 7 150 180 10
```

```
79 8.000000 56.000000 27 90 120 10
80 6.000000 68.000000 30 89 119 10
81 47.000000 47.000000 13 192 222 10
82 49.000000 58.000000 10 86 116 10
83 27.000000 43.000000 9 42 72 10
84 37.000000 31.000000 14 35 65 10
85 57.000000 29.000000 18 96 126 10
86 63.000000 23.000000 2 87 117 10
87 21.000000 24.000000 28 87 117 10
88 12.000000 24.000000 13 90 120 10
89 24.000000 58.000000 19 67 97 10
90 67.000000 5.000000 25 144 174 10
91 37.000000 47.000000 6 86 116 10
92 49.000000 42.000000 13 167 197 10
93 53.000000 43.000000 14 14 44 10
94 61.000000 52.000000 3 178 208 10
95 57.000000 48.000000 23 95 125 10
96 56.000000 37.000000 6 34 64 10
97 55.000000 54.000000 26 132 162 10
98 4.000000 18.000000 35 120 150 10
99 26.000000 52.000000 9 46 76 10
100 26.000000 35.000000 15 77 107 10
101 31.000000 67.000000 3 180 210 10
\.

drop table if exists vrp_vehicles cascade;
create table vrp_vehicles (
    vehicle_id integer not null primary key,
    capacity integer,
    case_no integer
);

copy vrp_vehicles (vehicle_id, capacity, case_no) from stdin;
1 200 5
2 200 5
3 200 5
4 200 5
5 200 5
6 200 5
7 200 5
8 200 5
9 200 5
10 200 5
11 200 5
12 200 5
13 200 5
14 200 5
15 200 5
16 200 5
17 200 5
18 200 5
19 200 5
20 200 5
\.

drop table if exists vrp_distance cascade;
create table vrp_distance (
    src_id integer,
    dest_id integer,
    cost Float8,
    distance Float8,
    traveltime Float8
);

copy vrp_distance (src_id, dest_id, cost, distance, traveltime) from stdin;
1 2 38.078866 38.078866 38.078866
1 3 30.805844 30.805844 30.805844
1 4 39.357337 39.357337 39.357337
1 5 36.055513 36.055513 36.055513
1 6 40.311289 40.311289 40.311289
1 7 33.301652 33.301652 33.301652
1 8 35.355339 35.355339 35.355339
1 9 39.051248 39.051248 39.051248
1 10 33.541020 33.541020 33.541020
1 11 31.622777 31.622777 31.622777
1 12 33.526109 33.526109 33.526109
1 13 32.388269 32.388269 32.388269
1 14 38.078866 38.078866 38.078866
1 15 35.355339 35.355339 35.355339
1 16 39.293765 39.293765 39.293765
1 17 41.231056 41.231056 41.231056
1 18 40.311289 40.311289 40.311289
1 19 45.177428 45.177428 45.177428
1 20 40.049969 40.049969 40.049969
1 21 35.057096 35.057096 35.057096
1 22 45.000000 45.000000 45.000000
1 23 35.000000 35.000000 35.000000
1 24 45.044423 45.044423 45.044423
1 25 35.057096 35.057096 35.057096
1 26 45.276926 45.276926 45.276926
1 27 58.523500 58.523500 58.523500
1 28 57.008771 57.008771 57.008771
1 29 55.713553 55.713553 55.713553
1 30 52.201533 52.201533 52.201533
1 31 52.000000 52.000000 52.000000
```

1 32 50.289164 50.289164 50.289164
1 33 51.078371 51.078371 51.078371
1 34 51.478151 51.478151 51.478151
1 35 47.434165 47.434165 47.434165
1 36 44.204072 44.204072 44.204072
1 37 43.011626 43.011626 43.011626
1 38 40.607881 40.607881 40.607881
1 39 37.202150 37.202150 37.202150
1 40 36.055513 36.055513 36.055513
1 41 40.311289 40.311289 40.311289
1 42 30.805844 30.805844 30.805844
1 43 33.541020 33.541020 33.541020
1 44 38.078866 38.078866 38.078866
1 45 35.341194 35.341194 35.341194
1 46 37.735925 37.735925 37.735925
1 47 37.202150 37.202150 37.202150
1 48 38.327536 38.327536 38.327536
1 49 45.044423 45.044423 45.044423
1 50 38.052595 38.052595 38.052595
1 51 35.341194 35.341194 35.341194
1 52 33.541020 33.541020 33.541020
1 53 25.000000 25.000000 25.000000
1 54 20.000000 20.000000 20.000000
1 55 18.027756 18.027756 18.027756
1 56 14.142136 14.142136 14.142136
1 57 18.027756 18.027756 18.027756
1 58 26.925824 26.925824 26.925824
1 59 47.169906 47.169906 47.169906
1 60 42.426407 42.426407 42.426407
1 61 26.925824 26.925824 26.925824
1 62 15.811388 15.811388 15.811388
1 63 29.154759 29.154759 29.154759
1 64 39.051248 39.051248 39.051248
1 65 20.615528 20.615528 20.615528
1 66 11.180340 11.180340 11.180340
1 67 13.038405 13.038405 13.038405
1 68 25.298221 25.298221 25.298221
1 69 10.000000 10.000000 10.000000
1 70 9.219544 9.219544 9.219544
1 71 19.646883 19.646883 19.646883
1 72 25.495098 25.495098 25.495098
1 73 27.459060 27.459060 27.459060
1 74 39.293765 39.293765 39.293765
1 75 36.055513 36.055513 36.055513
1 76 57.008771 57.008771 57.008771
1 77 42.941821 42.941821 42.941821
1 78 49.979996 49.979996 49.979996
1 79 32.557641 32.557641 32.557641
1 80 38.470768 38.470768 38.470768
1 81 7.615773 7.615773 7.615773
1 82 12.041595 12.041595 12.041595
1 83 14.764823 14.764823 14.764823
1 84 19.235384 19.235384 19.235384
1 85 27.018512 27.018512 27.018512
1 86 35.468296 35.468296 35.468296
1 87 32.202484 32.202484 32.202484
1 88 38.209946 38.209946 38.209946
1 89 17.888544 17.888544 17.888544
1 90 52.478567 52.478567 52.478567
1 91 4.242641 4.242641 4.242641
1 92 12.041595 12.041595 12.041595
1 93 14.764823 14.764823 14.764823
1 94 21.095023 21.095023 21.095023
1 95 17.117243 17.117243 17.117243
1 96 20.615528 20.615528 20.615528
1 97 15.524175 15.524175 15.524175
1 98 48.166378 48.166378 48.166378
1 99 14.142136 14.142136 14.142136
1 100 20.518285 20.518285 20.518285
1 101 19.235384 19.235384 19.235384
2 1 38.078866 38.078866 38.078866
2 3 10.440307 10.440307 10.440307
2 4 3.000000 3.000000 3.000000
2 5 7.071068 7.071068 7.071068
2 6 5.000000 5.000000 5.000000
2 7 12.206556 12.206556 12.206556
2 8 14.142136 14.142136 14.142136
2 9 11.180340 11.180340 11.180340
2 10 52.201533 52.201533 52.201533
2 11 47.434165 47.434165 47.434165
2 12 48.104054 48.104054 48.104054
2 13 43.462628 43.462628 43.462628
2 14 53.851648 53.851648 53.851648
2 15 44.721360 44.721360 44.721360
2 16 50.537115 50.537115 50.537115
2 17 51.478151 51.478151 51.478151
2 18 47.169906 47.169906 47.169906
2 19 82.225300 82.225300 82.225300
2 20 76.902536 76.902536 76.902536
2 21 72.034714 72.034714 72.034714
2 22 81.394103 81.394103 81.394103
2 23 71.589105 71.589105 71.589105
2 24 81.049368 81.049368 81.049368
2 25 71.196910 71.196910 71.196910
2 26 80.622577 80.622577 80.622577

2 27 89.022469 89.022469 89.022469
2 28 86.023253 86.023253 86.023253
2 29 86.683332 86.683332 86.683332
2 30 82.006097 82.006097 82.006097
2 31 83.630138 83.630138 83.630138
2 32 80.430094 80.430094 80.430094
2 33 82.879430 82.879430 82.879430
2 34 84.852814 84.852814 84.852814
2 35 78.102497 78.102497 78.102497
2 36 42.000000 42.000000 42.000000
2 37 40.000000 40.000000 40.000000
2 38 40.112342 40.112342 40.112342
2 39 37.336309 37.336309 37.336309
2 40 35.355339 35.355339 35.355339
2 41 35.000000 35.000000 35.000000
2 42 34.481879 34.481879 34.481879
2 43 30.413813 30.413813 30.413813
2 44 30.000000 30.000000 30.000000
2 45 30.149627 30.149627 30.149627
2 46 5.830952 5.830952 5.830952
2 47 8.602325 8.602325 8.602325
2 48 46.141088 46.141088 46.141088
2 49 81.786307 81.786307 81.786307
2 50 74.953319 74.953319 74.953319
2 51 68.622154 68.622154 68.622154
2 52 71.589105 71.589105 71.589105
2 53 55.000000 55.000000 55.000000
2 54 35.355339 35.355339 35.355339
2 55 39.051248 39.051248 39.051248
2 56 25.495098 25.495098 25.495098
2 57 55.901699 55.901699 55.901699
2 58 60.207973 60.207973 60.207973
2 59 75.663730 75.663730 75.663730
2 60 66.708320 66.708320 66.708320
2 61 26.925824 26.925824 26.925824
2 62 28.284271 28.284271 28.284271
2 63 64.031242 64.031242 64.031242
2 64 76.321688 76.321688 76.321688
2 65 58.523500 58.523500 58.523500
2 66 46.097722 46.097722 46.097722
2 67 50.596443 50.596443 50.596443
2 68 58.051701 58.051701 58.051701
2 69 29.154759 29.154759 29.154759
2 70 33.541020 33.541020 33.541020
2 71 18.867962 18.867962 18.867962
2 72 50.000000 50.000000 50.000000
2 73 42.941821 42.941821 42.941821
2 74 33.970576 33.970576 33.970576
2 75 65.192024 65.192024 65.192024
2 76 82.462113 82.462113 82.462113
2 77 80.956779 80.956779 80.956779
2 78 82.024387 82.024387 82.024387
2 79 33.615473 33.615473 33.615473
2 80 25.495098 25.495098 25.495098
2 81 43.908997 43.908997 43.908997
2 82 36.124784 36.124784 36.124784
2 83 42.047592 42.047592 42.047592
2 84 55.317267 55.317267 55.317267
2 85 64.498062 64.498062 64.498062
2 86 72.718636 72.718636 72.718636
2 87 61.131007 61.131007 61.131007
2 88 62.369865 62.369865 62.369865
2 89 27.018512 27.018512 27.018512
2 90 90.354856 90.354856 90.354856
2 91 39.849718 39.849718 39.849718
2 92 49.244289 49.244289 49.244289
2 93 50.477718 50.477718 50.477718
2 94 48.836462 48.836462 48.836462
2 95 48.918299 48.918299 48.918299
2 96 57.140179 57.140179 57.140179
2 97 43.139309 43.139309 43.139309
2 98 70.213959 70.213959 70.213959
2 99 33.015148 33.015148 33.015148
2 100 50.009999 50.009999 50.009999
2 101 18.973666 18.973666 18.973666
3 1 30.805844 30.805844 30.805844
3 2 10.440307 10.440307 10.440307
3 4 10.000000 10.000000 10.000000
3 5 5.385165 5.385165 5.385165
3 6 10.198039 10.198039 10.198039
3 7 4.000000 4.000000 4.000000
3 8 7.000000 7.000000 7.000000
3 9 8.602325 8.602325 8.602325
3 10 41.761226 41.761226 41.761226
3 11 37.000000 37.000000 37.000000
3 12 37.696154 37.696154 37.696154
3 13 33.105891 33.105891 33.105891
3 14 43.462628 43.462628 43.462628
3 15 34.481879 34.481879 34.481879
3 16 40.311289 40.311289 40.311289
3 17 41.340053 41.340053 41.340053
3 18 37.202150 37.202150 37.202150
3 19 73.375745 73.375745 73.375745
3 20 68.007353 68.007353 68.007353
3 21 63.245553 63.245553 63.245553

3 22 72.277244 72.277244 72.277244
3 23 62.641839 62.641839 62.641839
3 24 71.805292 71.805292 71.805292
3 25 62.096699 62.096699 62.096699
3 26 71.196910 71.196910 71.196910
3 27 85.755466 85.755466 85.755466
3 28 83.240615 83.240615 83.240615
3 29 83.216585 83.216585 83.216585
3 30 78.892332 78.892332 78.892332
3 31 79.881162 79.881162 79.881162
3 32 77.175126 77.175126 77.175126
3 33 79.056942 79.056942 79.056942
3 34 80.430094 80.430094 80.430094
3 35 74.625733 74.625733 74.625733
3 36 46.097722 46.097722 46.097722
3 37 44.147480 44.147480 44.147480
3 38 43.566042 43.566042 43.566042
3 39 40.311289 40.311289 40.311289
3 40 38.327536 38.327536 38.327536
3 41 39.293765 39.293765 39.293765
3 42 36.000000 36.000000 36.000000
3 43 33.376639 33.376639 33.376639
3 44 34.481879 34.481879 34.481879
3 45 33.734256 33.734256 33.734256
3 46 7.280110 7.280110 7.280110
3 47 6.403124 6.403124 6.403124
3 48 36.055513 36.055513 36.055513
3 49 72.801099 72.801099 72.801099
3 50 66.098411 66.098411 66.098411
3 51 64.031242 64.031242 64.031242
3 52 64.140471 64.140471 64.140471
3 53 45.099889 45.099889 45.099889
3 54 25.079872 25.079872 25.079872
3 55 36.249138 36.249138 36.249138
3 56 17.000000 17.000000 17.000000
3 57 48.826222 48.826222 48.826222
3 58 50.635956 50.635956 50.635956
3 59 65.375837 65.375837 65.375837
3 60 56.293872 56.293872 56.293872
3 61 16.552945 16.552945 16.552945
3 62 25.079872 25.079872 25.079872
3 63 58.728187 58.728187 58.728187
3 64 69.814039 69.814039 69.814039
3 65 50.537115 50.537115 50.537115
3 66 37.336309 37.336309 37.336309
3 67 42.485292 42.485292 42.485292
3 68 53.413481 53.413481 53.413481
3 69 23.430749 23.430749 23.430749
3 70 24.698178 24.698178 24.698178
3 71 14.317821 14.317821 14.317821
3 72 47.423623 47.423623 47.423623
3 73 42.201896 42.201896 42.201896
3 74 25.000000 25.000000 25.000000
3 75 55.036352 55.036352 55.036352
3 76 72.034714 72.034714 72.034714
3 77 73.573093 73.573093 73.573093
3 78 72.006944 72.006944 72.006944
3 79 23.600847 23.600847 23.600847
3 80 17.464249 17.464249 17.464249
3 81 37.536649 37.536649 37.536649
3 82 31.906112 31.906112 31.906112
3 83 32.388269 32.388269 32.388269
3 84 46.486557 46.486557 46.486557
3 85 57.801384 57.801384 57.801384
3 86 66.219333 66.219333 66.219333
3 87 51.009803 51.009803 51.009803
3 88 51.971146 51.971146 51.971146
3 89 17.117243 17.117243 17.117243
3 90 83.216585 83.216585 83.216585
3 91 31.764760 31.764760 31.764760
3 92 42.638011 42.638011 42.638011
3 93 44.553339 44.553339 44.553339
3 94 45.276926 45.276926 45.276926
3 95 44.204072 44.204072 44.204072
3 96 50.990195 50.990195 50.990195
3 97 39.115214 39.115214 39.115214
3 98 59.774577 59.774577 59.774577
3 99 23.345235 23.345235 23.345235
3 100 40.199502 40.199502 40.199502
3 101 12.041595 12.041595 12.041595
4 1 39.357337 39.357337 39.357337
4 2 3.000000 3.000000 3.000000
4 3 10.000000 10.000000 10.000000
4 5 5.385165 5.385165 5.385165
4 6 2.000000 2.000000 2.000000
4 7 10.770330 10.770330 10.770330
4 8 12.206556 12.206556 12.206556
4 9 8.602325 8.602325 8.602325
4 10 51.419841 51.419841 51.419841
4 11 46.572524 46.572524 46.572524
4 12 47.127487 47.127487 47.127487
4 13 42.379240 42.379240 42.379240
4 14 52.810984 52.810984 52.810984
4 15 43.462628 43.462628 43.462628
4 16 49.244289 49.244289 49.244289
4 17 50.089919 50.089919 50.089919

4 17 50.089919 50.089919 50.089919
4 18 45.650849 45.650849 45.650849
4 19 82.969874 82.969874 82.969874
4 20 77.620873 77.620873 77.620873
4 21 72.801099 72.801099 72.801099
4 22 82.000000 82.000000 82.000000
4 23 72.277244 72.277244 72.277244
4 24 81.584312 81.584312 81.584312
4 25 71.805292 71.805292 71.805292
4 26 81.049368 81.049368 81.049368
4 27 91.400219 91.400219 91.400219
4 28 88.481637 88.481637 88.481637
4 29 89.022469 89.022469 89.022469
4 30 84.403791 84.403791 84.403791
4 31 85.912746 85.912746 85.912746
4 32 82.800966 82.800966 82.800966
4 33 85.146932 85.146932 85.146932
4 34 87.000000 87.000000 87.000000
4 35 80.430094 80.430094 80.430094
4 36 45.000000 45.000000 45.000000
4 37 43.000000 43.000000 43.000000
4 38 43.104524 43.104524 43.104524
4 39 40.311289 40.311289 40.311289
4 40 38.327536 38.327536 38.327536
4 41 38.000000 38.000000 38.000000
4 42 37.363083 37.363083 37.363083
4 43 33.376639 33.376639 33.376639
4 44 33.000000 33.000000 33.000000
4 45 33.136083 33.136083 33.136083
4 46 3.605551 3.605551 3.605551
4 47 6.403124 6.403124 6.403124
4 48 44.721360 44.721360 44.721360
4 49 82.462113 82.462113 82.462113
4 50 75.690158 75.690158 75.690158
4 51 70.710678 70.710678 70.710678
4 52 72.897188 72.897188 72.897188
4 53 55.081757 55.081757 55.081757
4 54 35.057096 35.057096 35.057096
4 55 41.400483 41.400483 41.400483
4 56 26.248809 26.248809 26.248809
4 57 57.306195 57.306195 57.306195
4 58 60.530984 60.530984 60.530984
4 59 75.325958 75.325958 75.325958
4 60 66.098411 66.098411 66.098411
4 61 25.961510 25.961510 25.961510
4 62 30.479501 30.479501 30.479501
4 63 65.946948 65.946948 65.946948
4 64 77.935871 77.935871 77.935871
4 65 59.615434 59.615434 59.615434
4 66 46.840154 46.840154 46.840154
4 67 51.623638 51.623638 51.623638
4 68 60.108236 60.108236 60.108236
4 69 30.805844 30.805844 30.805844
4 70 34.205263 34.205263 34.205263
4 71 20.615528 20.615528 20.615528
4 72 52.430907 52.430907 52.430907
4 73 45.617979 45.617979 45.617979
4 74 32.015621 32.015621 32.015621
4 75 65.030762 65.030762 65.030762
4 76 81.786307 81.786307 81.786307
4 77 82.298238 82.298238 82.298238
4 78 82.006097 82.006097 82.006097
4 79 32.202484 32.202484 32.202484
4 80 23.345235 23.345235 23.345235
4 81 45.486262 45.486262 45.486262
4 82 38.183766 38.183766 38.183766
4 83 42.296572 42.296572 42.296572
4 84 56.044625 56.044625 56.044625
4 85 66.037868 66.037868 66.037868
4 86 74.330344 74.330344 74.330344
4 87 61.008196 61.008196 61.008196
4 88 61.814238 61.814238 61.814238
4 89 27.073973 27.073973 27.073973
4 90 91.787799 91.787799 91.787799
4 91 40.853396 40.853396 40.853396
4 92 50.774009 50.774009 50.774009
4 93 52.201533 52.201533 52.201533
4 94 51.088159 51.088159 51.088159
4 95 50.931326 50.931326 50.931326
4 96 58.821765 58.821765 58.821765
4 97 45.276926 45.276926 45.276926
4 98 69.375788 69.375788 69.375788
4 99 33.241540 33.241540 33.241540
4 100 50.159745 50.159745 50.159745
4 101 20.124612 20.124612 20.124612
5 1 36.055513 36.055513 36.055513
5 2 7.071068 7.071068 7.071068
5 3 5.385165 5.385165 5.385165
5 4 5.385165 5.385165 5.385165
5 6 5.000000 5.000000 5.000000
5 7 5.385165 5.385165 5.385165
5 8 7.071068 7.071068 7.071068
5 9 5.000000 5.000000 5.000000
5 10 46.097722 46.097722 46.097722
5 11 41.231056 41.231056 41.231056
5 12 41.761226 41.761226 41.761226

5 12 41.761226 41.761226 41.761226
5 13 37.000000 37.000000 37.000000
5 14 47.434165 47.434165 47.434165
5 15 38.078866 38.078866 38.078866
5 16 43.863424 43.863424 43.863424
5 17 44.721360 44.721360 44.721360
5 18 40.311289 40.311289 40.311289
5 19 78.746428 78.746428 78.746428
5 20 73.375745 73.375745 73.375745
5 21 68.622154 68.622154 68.622154
5 22 77.620873 77.620873 77.620873
5 23 68.007353 68.007353 68.007353
5 24 77.129761 77.129761 77.129761
5 25 67.446275 67.446275 67.446275
5 26 76.485293 76.485293 76.485293
5 27 90.138782 90.138782 90.138782
5 28 87.464278 87.464278 87.464278
5 29 87.658428 87.658428 87.658428
5 30 83.216585 83.216585 83.216585
5 31 84.403791 84.403791 84.403791
5 32 81.541401 81.541401 81.541401
5 33 83.600239 83.600239 83.600239
5 34 85.146932 85.146932 85.146932
5 35 79.056942 79.056942 79.056942
5 36 47.265209 47.265209 47.265209
5 37 45.276926 45.276926 45.276926
5 38 45.044423 45.044423 45.044423
5 39 42.000000 42.000000 42.000000
5 40 40.000000 40.000000 40.000000
5 41 40.311289 40.311289 40.311289
5 42 38.327536 38.327536 38.327536
5 43 35.000000 35.000000 35.000000
5 44 35.355339 35.355339 35.355339
5 45 35.057096 35.057096 35.057096
5 46 2.000000 2.000000 2.000000
5 47 2.000000 2.000000 2.000000
5 48 39.357337 39.357337 39.357337
5 49 78.160092 78.160092 78.160092
5 50 71.470274 71.470274 71.470274
5 51 68.767725 68.767725 68.767725
5 52 69.462220 69.462220 69.462220
5 53 50.249378 50.249378 50.249378
5 54 30.000000 30.000000 30.000000
5 55 40.311289 40.311289 40.311289
5 56 22.360680 22.360680 22.360680
5 57 54.083269 54.083269 54.083269
5 58 55.901699 55.901699 55.901699
5 59 70.178344 70.178344 70.178344
5 60 60.827625 60.827625 60.827625
5 61 20.615528 20.615528 20.615528
5 62 29.154759 29.154759 29.154759
5 63 63.639610 63.639610 63.639610
5 64 75.000000 75.000000 75.000000
5 65 55.901699 55.901699 55.901699
5 66 42.720019 42.720019 42.720019
5 67 47.853944 47.853944 47.853944
5 68 58.137767 58.137767 58.137767
5 69 28.284271 28.284271 28.284271
5 70 30.083218 30.083218 30.083218
5 71 18.601075 18.601075 18.601075
5 72 51.478151 51.478151 51.478151
5 73 45.541190 45.541190 45.541190
5 74 26.907248 26.907248 26.907248
5 75 60.000000 60.000000 60.000000
5 76 76.485293 76.485293 76.485293
5 77 78.892332 78.892332 78.892332
5 78 77.058419 77.058419 77.058419
5 79 26.832816 26.832816 26.832816
5 80 18.439089 18.439089 18.439089
5 81 42.638011 42.638011 42.638011
5 82 36.400549 36.400549 36.400549
5 83 37.656341 37.656341 37.656341
5 84 51.865210 51.865210 51.865210
5 85 63.007936 63.007936 63.007936
5 86 71.400280 71.400280 71.400280
5 87 56.008928 56.008928 56.008928
5 88 56.568542 56.568542 56.568542
5 89 22.360680 22.360680 22.360680
5 90 88.509886 88.509886 88.509886
5 91 37.121422 37.121422 37.121422
5 92 47.801674 47.801674 47.801674
5 93 49.578221 49.578221 49.578221
5 94 49.648766 49.648766 49.648766
5 95 48.918299 48.918299 48.918299
5 96 56.080300 56.080300 56.080300
5 97 43.600459 43.600459 43.600459
5 98 64.031242 64.031242 64.031242
5 99 28.635642 28.635642 28.635642
5 100 45.398238 45.398238 45.398238
5 101 17.029386 17.029386 17.029386
6 1 40.311289 40.311289 40.311289
6 2 5.000000 5.000000 5.000000
6 3 10.198039 10.198039 10.198039
6 4 2.000000 2.000000 2.000000
6 5 5.000000 5.000000 5.000000
6 7 10.198039 10.198039 10.198039

6 8 11.180340 11.180340 11.180340
6 9 7.071068 7.071068 7.071068
6 10 50.990195 50.990195 50.990195
6 11 46.097722 46.097722 46.097722
6 12 46.572524 46.572524 46.572524
6 13 41.761226 41.761226 41.761226
6 14 52.201533 52.201533 52.201533
6 15 42.720019 42.720019 42.720019
6 16 48.466483 48.466483 48.466483
6 17 49.244289 49.244289 49.244289
6 18 44.721360 44.721360 44.721360
6 19 83.522452 83.522452 83.522452
6 20 78.160092 78.160092 78.160092
6 21 73.375745 73.375745 73.375745
6 22 82.462113 82.462113 82.462113
6 23 72.801099 72.801099 72.801099
6 24 82.000000 82.000000 82.000000
6 25 72.277244 72.277244 72.277244
6 26 81.394103 81.394103 81.394103
6 27 93.005376 93.005376 93.005376
6 28 90.138782 90.138782 90.138782
6 29 90.603532 90.603532 90.603532
6 30 86.023253 86.023253 86.023253
6 31 87.458562 87.458562 87.458562
6 32 84.403791 84.403791 84.403791
6 33 86.683332 86.683332 86.683332
6 34 88.459030 88.459030 88.459030
6 35 82.006097 82.006097 82.006097
6 36 47.000000 47.000000 47.000000
6 37 45.000000 45.000000 45.000000
6 38 45.099889 45.099889 45.099889
6 39 42.296572 42.296572 42.296572
6 40 40.311289 40.311289 40.311289
6 41 40.000000 40.000000 40.000000
6 42 39.293765 39.293765 39.293765
6 43 35.355339 35.355339 35.355339
6 44 35.000000 35.000000 35.000000
6 45 35.128336 35.128336 35.128336
6 46 3.000000 3.000000 3.000000
6 47 5.385165 5.385165 5.385165
6 48 43.863424 43.863424 43.863424
6 49 82.969874 82.969874 82.969874
6 50 76.243032 76.243032 76.243032
6 51 72.138755 72.138755 72.138755
6 52 73.824115 73.824115 73.824115
6 53 55.226805 55.226805 55.226805
6 54 35.000000 35.000000 35.000000
6 55 43.011626 43.011626 43.011626
6 56 26.925824 26.925824 26.925824
6 57 58.309519 58.309519 58.309519
6 58 60.827625 60.827625 60.827625
6 59 75.166482 75.166482 75.166482
6 60 65.764732 65.764732 65.764732
6 61 25.495098 25.495098 25.495098
6 62 32.015621 32.015621 32.015621
6 63 67.268120 67.268120 67.268120
6 64 79.056942 79.056942 79.056942
6 65 60.415230 60.415230 60.415230
6 66 47.434165 47.434165 47.434165
6 67 52.392748 52.392748 52.392748
6 68 61.522354 61.522354 61.522354
6 69 32.015621 32.015621 32.015621
6 70 34.785054 34.785054 34.785054
6 71 21.931712 21.931712 21.931712
6 72 54.083269 54.083269 54.083269
6 73 47.423623 47.423623 47.423623
6 74 30.805844 30.805844 30.805844
6 75 65.000000 65.000000 65.000000
6 76 81.394103 81.394103 81.394103
6 77 83.240615 83.240615 83.240615
6 78 82.054860 82.054860 82.054860
6 79 31.384710 31.384710 31.384710
6 80 22.022716 22.022716 22.022716
6 81 46.615448 46.615448 46.615448
6 82 39.623226 39.623226 39.623226
6 83 42.579338 42.579338 42.579338
6 84 56.612719 56.612719 56.612719
6 85 67.119297 67.119297 67.119297
6 86 75.451971 75.451971 75.451971
6 87 61.008196 61.008196 61.008196
6 88 61.522354 61.522354 61.522354
6 89 27.294688 27.294688 27.294688
6 90 92.784697 92.784697 92.784697
6 91 41.629317 41.629317 41.629317
6 92 51.865210 51.865210 51.865210
6 93 53.413481 53.413481 53.413481
6 94 52.630789 52.630789 52.630789
6 95 52.325902 52.325902 52.325902
6 96 60.000000 60.000000 60.000000
6 97 46.754679 46.754679 46.754679
6 98 68.883960 68.883960 68.883960
6 99 33.541020 33.541020 33.541020
6 100 50.358713 50.358713 50.358713
6 101 21.095023 21.095023 21.095023
7 1 33.301652 33.301652 33.301652

7 2 12.206556 12.206556 12.206556
7 3 4.000000 4.000000 4.000000
7 4 10.770330 10.770330 10.770330
7 5 5.385165 5.385165 5.385165
7 6 10.198039 10.198039 10.198039
7 8 3.000000 3.000000 3.000000
7 9 5.830952 5.830952 5.830952
7 10 40.792156 40.792156 40.792156
7 11 35.902646 35.902646 35.902646
7 12 36.400549 36.400549 36.400549
7 13 31.622777 31.622777 31.622777
7 14 42.059482 42.059482 42.059482
7 15 32.695565 32.695565 32.695565
7 16 38.483763 38.483763 38.483763
7 17 39.357337 39.357337 39.357337
7 18 34.985711 34.985711 34.985711
7 19 74.672619 74.672619 74.672619
7 20 69.289249 69.289249 69.289249
7 21 64.621978 64.621978 64.621978
7 22 73.375745 73.375745 73.375745
7 23 63.906181 63.906181 63.906181
7 24 72.801099 72.801099 72.801099
7 25 63.245553 63.245553 63.245553
7 26 72.034714 72.034714 72.034714
7 27 89.185201 89.185201 89.185201
7 28 86.769810 86.769810 86.769810
7 29 86.608314 86.608314 86.608314
7 30 82.365041 82.365041 82.365041
7 31 83.216585 83.216585 83.216585
7 32 80.622577 80.622577 80.622577
7 33 82.377181 82.377181 82.377181
7 34 83.600239 83.600239 83.600239
7 35 78.032045 78.032045 78.032045
7 36 50.009999 50.009999 50.009999
7 37 48.052055 48.052055 48.052055
7 38 47.518417 47.518417 47.518417
7 39 44.283180 44.283180 44.283180
7 40 42.296572 42.296572 42.296572
7 41 43.174066 43.174066 43.174066
7 42 40.000000 40.000000 40.000000
7 43 37.336309 37.336309 37.336309
7 44 38.327536 38.327536 38.327536
7 45 37.656341 37.656341 37.656341
7 46 7.280110 7.280110 7.280110
7 47 5.000000 5.000000 5.000000
7 48 34.000000 34.000000 34.000000
7 49 74.000000 74.000000 74.000000
7 50 67.416615 67.416615 67.416615
7 51 67.201190 67.201190 67.201190
7 52 66.287254 66.287254 66.287254
7 53 45.541190 45.541190 45.541190
7 54 25.079872 25.079872 25.079872
7 55 39.924930 39.924930 39.924930
7 56 19.209373 19.209373 19.209373
7 57 51.224994 51.224994 51.224994
7 58 51.419841 51.419841 51.419841
7 59 65.069194 65.069194 65.069194
7 60 55.578773 55.578773 55.578773
7 61 15.297059 15.297059 15.297059
7 62 28.792360 28.792360 28.792360
7 63 61.717096 61.717096 61.717096
7 64 72.346389 72.346389 72.346389
7 65 52.478567 52.478567 52.478567
7 66 38.910153 38.910153 38.910153
7 67 44.418465 44.418465 44.418465
7 68 56.612719 56.612719 56.612719
7 69 26.627054 26.627054 26.627054
7 70 26.419690 26.419690 26.419690
7 71 18.027756 18.027756 18.027756
7 72 51.078371 51.078371 51.078371
7 73 46.097722 46.097722 46.097722
7 74 21.931712 21.931712 21.931712
7 75 55.036352 55.036352 55.036352
7 76 71.196910 71.196910 71.196910
7 77 75.716577 75.716577 75.716577
7 78 72.173402 72.173402 72.173402
7 79 21.470911 21.470911 21.470911
7 80 13.892444 13.892444 13.892444
7 81 40.311289 40.311289 40.311289
7 82 35.355339 35.355339 35.355339
7 83 33.241540 33.241540 33.241540
7 84 47.927028 47.927028 47.927028
7 85 60.307545 60.307545 60.307545
7 86 68.767725 68.767725 68.767725
7 87 51.088159 51.088159 51.088159
7 88 51.351728 51.351728 51.351728
7 89 18.027756 18.027756 18.027756
7 90 85.445889 85.445889 85.445889
7 91 33.837849 33.837849 33.837849
7 92 45.276926 45.276926 45.276926
7 93 47.423623 47.423623 47.423623
7 94 48.764741 48.764741 48.764741
7 95 47.434165 47.434165 47.434165
7 96 53.740115 53.740115 53.740115
7 97 42.544095 42.544095 42.544095

7 98 58.694122 58.694122 58.694122
7 99 24.351591 24.351591 24.351591
7 100 40.792156 40.792156 40.792156
7 101 15.264338 15.264338 15.264338
8 1 35.355339 35.355339 35.355339
8 2 14.142136 14.142136 14.142136
8 3 7.000000 7.000000 7.000000
8 4 12.206556 12.206556 12.206556
8 5 7.071068 7.071068 7.071068
8 6 11.180340 11.180340 11.180340
8 7 3.000000 3.000000 3.000000
8 9 5.000000 5.000000 5.000000
8 10 40.311289 40.311289 40.311289
8 11 35.355339 35.355339 35.355339
8 12 35.693137 35.693137 35.693137
8 13 30.805844 30.805844 30.805844
8 14 41.231056 41.231056 41.231056
8 15 31.622777 31.622777 31.622777
8 16 37.336309 37.336309 37.336309
8 17 38.078866 38.078866 38.078866
8 18 33.541020 33.541020 33.541020
8 19 75.769387 75.769387 75.769387
8 20 70.384657 70.384657 70.384657
8 21 65.795137 65.795137 65.795137
8 22 74.330344 74.330344 74.330344
8 23 65.000000 65.000000 65.000000
8 24 73.681748 73.681748 73.681748
8 25 64.257295 64.257295 64.257295
8 26 72.801099 72.801099 72.801099
8 27 91.787799 91.787799 91.787799
8 28 89.442719 89.442719 89.442719
8 29 89.185201 89.185201 89.185201
8 30 85.000000 85.000000 85.000000
8 31 85.755466 85.755466 85.755466
8 32 83.240615 83.240615 83.240615
8 33 84.905830 84.905830 84.905830
8 34 86.023253 86.023253 86.023253
8 35 80.622577 80.622577 80.622577
8 36 52.952809 52.952809 52.952809
8 37 50.990195 50.990195 50.990195
8 38 50.487622 50.487622 50.487622
8 39 47.265209 47.265209 47.265209
8 40 45.276926 45.276926 45.276926
8 41 46.097722 46.097722 46.097722
8 42 43.000000 43.000000 43.000000
8 43 40.311289 40.311289 40.311289
8 44 41.231056 41.231056 41.231056
8 45 40.607881 40.607881 40.607881
8 46 8.602325 8.602325 8.602325
8 47 5.830952 5.830952 5.830952
8 48 32.695565 32.695565 32.695565
8 49 75.026662 75.026662 75.026662
8 50 68.541958 68.541958 68.541958
8 51 69.634761 69.634761 69.634761
8 52 68.007353 68.007353 68.007353
8 53 46.097722 46.097722 46.097722
8 54 25.495098 25.495098 25.495098
8 55 42.720019 42.720019 42.720019
8 56 21.213203 21.213203 21.213203
8 57 53.150729 53.150729 53.150729
8 58 52.201533 52.201533 52.201533
8 59 65.000000 65.000000 65.000000
8 60 55.226805 55.226805 55.226805
8 61 15.000000 15.000000 15.000000
8 62 31.622777 31.622777 31.622777
8 63 64.031242 64.031242 64.031242
8 64 74.330344 74.330344 74.330344
8 65 54.083269 54.083269 54.083269
8 66 40.311289 40.311289 40.311289
8 67 46.043458 46.043458 46.043458
8 68 59.076222 59.076222 59.076222
8 69 29.154759 29.154759 29.154759
8 70 28.017851 28.017851 28.017851
8 71 20.880613 20.880613 20.880613
8 72 53.851648 53.851648 53.851648
8 73 49.030603 49.030603 49.030603
8 74 19.849433 19.849433 19.849433
8 75 55.226805 55.226805 55.226805
8 76 70.710678 70.710678 70.710678
8 77 77.420927 77.420927 77.420927
8 78 72.443081 72.443081 72.443081
8 79 20.248457 20.248457 20.248457
8 80 11.401754 11.401754 11.401754
8 81 42.520583 42.520583 42.520583
8 82 38.013156 38.013156 38.013156
8 83 34.176015 34.176015 34.176015
8 84 49.193496 49.193496 49.193496
8 85 62.289646 62.289646 62.289646
8 86 70.767224 70.767224 70.767224
8 87 51.351728 51.351728 51.351728
8 88 51.088159 51.088159 51.088159
8 89 19.235384 19.235384 19.235384
8 90 87.200917 87.200917 87.200917
8 91 35.608988 35.608988 35.608988
8 92 47.381431 47.381431 47.381431

8 93 49.678969 49.678969 49.678969
8 94 51.429563 51.429563 51.429563
8 95 49.929951 49.929951 49.929951
8 96 55.901699 55.901699 55.901699
8 97 45.177428 45.177428 45.177428
8 98 58.051701 58.051701 58.051701
8 99 25.495098 25.495098 25.495098
8 100 41.484937 41.484937 41.484937
8 101 17.888544 17.888544 17.888544
9 1 39.051248 39.051248 39.051248
9 2 11.180340 11.180340 11.180340
9 3 8.602325 8.602325 8.602325
9 4 8.602325 8.602325 8.602325
9 5 5.000000 5.000000 5.000000
9 6 7.071068 7.071068 7.071068
9 7 5.830952 5.830952 5.830952
9 8 5.000000 5.000000 5.000000
9 10 45.276926 45.276926 45.276926
9 11 40.311289 40.311289 40.311289
9 12 40.607881 40.607881 40.607881
9 13 35.693137 35.693137 35.693137
9 14 46.097722 46.097722 46.097722
9 15 36.400549 36.400549 36.400549
9 16 42.059482 42.059482 42.059482
9 17 42.720019 42.720019 42.720019
9 18 38.078866 38.078866 38.078866
9 19 80.411442 80.411442 80.411442
9 20 75.026662 75.026662 75.026662
9 21 70.384657 70.384657 70.384657
9 22 79.056942 79.056942 79.056942
9 23 69.641941 69.641941 69.641941
9 24 78.447435 78.447435 78.447435
9 25 68.949257 68.949257 68.949257
9 26 77.620873 77.620873 77.620873
9 27 94.339811 94.339811 94.339811
9 28 91.787799 91.787799 91.787799
9 29 91.809586 91.809586 91.809586
9 30 87.464278 87.464278 87.464278
9 31 88.481637 88.481637 88.481637
9 32 85.755466 85.755466 85.755466
9 33 87.658428 87.658428 87.658428
9 34 89.022469 89.022469 89.022469
9 35 83.216585 83.216585 83.216585
9 36 52.239832 52.239832 52.239832
9 37 50.249378 50.249378 50.249378
9 38 50.039984 50.039984 50.039984
9 39 47.000000 47.000000 47.000000
9 40 45.000000 45.000000 45.000000
9 41 45.276926 45.276926 45.276926
9 42 43.289722 43.289722 43.289722
9 43 40.000000 40.000000 40.000000
9 44 40.311289 40.311289 40.311289
9 45 40.049969 40.049969 40.049969
9 46 5.385165 5.385165 5.385165
9 47 3.000000 3.000000 3.000000
9 48 37.336309 37.336309 37.336309
9 49 79.711982 79.711982 79.711982
9 50 73.164199 73.164199 73.164199
9 51 72.622311 72.622311 72.622311
9 52 72.111026 72.111026 72.111026
9 53 50.990195 50.990195 50.990195
9 54 30.413813 30.413813 30.413813
9 55 44.721360 44.721360 44.721360
9 56 25.000000 25.000000 25.000000
9 57 57.008771 57.008771 57.008771
9 58 57.008771 57.008771 57.008771
9 59 70.000000 70.000000 70.000000
9 60 60.207973 60.207973 60.207973
9 61 20.000000 20.000000 20.000000
9 62 33.541020 33.541020 33.541020
9 63 67.268120 67.268120 67.268120
9 64 78.102497 78.102497 78.102497
9 65 58.309519 58.309519 58.309519
9 66 44.721360 44.721360 44.721360
9 67 50.249378 50.249378 50.249378
9 68 62.008064 62.008064 62.008064
9 69 32.015621 32.015621 32.015621
9 70 32.249031 32.249031 32.249031
9 71 22.825424 22.825424 22.825424
9 72 55.901699 55.901699 55.901699
9 73 50.289164 50.289164 50.289164
9 74 23.853721 23.853721 23.853721
9 75 60.207973 60.207973 60.207973
9 76 75.663730 75.663730 75.663730
9 77 81.541401 81.541401 81.541401
9 78 77.414469 77.414469 77.414469
9 79 25.000000 25.000000 25.000000
9 80 15.000000 15.000000 15.000000
9 81 45.967380 45.967380 45.967380
9 82 40.496913 40.496913 40.496913
9 83 38.897301 38.897301 38.897301
9 84 53.712196 53.712196 53.712196
9 85 66.068147 66.068147 66.068147
9 86 74.518454 74.518454 74.518454
9 87 56.320511 56.320511 56.320511

9 88 56.080300 56.080300 56.080300
9 89 23.769729 23.769729 23.769729
9 90 91.263355 91.263355 91.263355
9 91 39.661064 39.661064 39.661064
9 92 50.990195 50.990195 50.990195
9 93 53.037722 53.037722 53.037722
9 94 53.851648 53.851648 53.851648
9 95 52.801515 52.801515 52.801515
9 96 59.413803 59.413803 59.413803
9 97 47.707442 47.707442 47.707442
9 98 62.968246 62.968246 62.968246
9 99 30.083218 30.083218 30.083218
9 100 46.324939 46.324939 46.324939
9 101 20.615528 20.615528 20.615528
10 1 33.541020 33.541020 33.541020
10 2 52.201533 52.201533 52.201533
10 3 41.761226 41.761226 41.761226
10 4 51.419841 51.419841 51.419841
10 5 46.097722 46.097722 46.097722
10 6 50.990195 50.990195 50.990195
10 7 40.792156 40.792156 40.792156
10 8 40.311289 40.311289 40.311289
10 9 45.276926 45.276926 45.276926
10 11 5.000000 5.000000 5.000000
10 12 5.385165 5.385165 5.385165
10 13 10.198039 10.198039 10.198039
10 14 5.000000 5.000000 5.000000
10 15 11.180340 11.180340 11.180340
10 16 9.433981 9.433981 9.433981
10 17 11.180340 11.180340 11.180340
10 18 14.142136 14.142136 14.142136
10 19 45.343136 45.343136 45.343136
10 20 40.607881 40.607881 40.607881
10 21 37.735925 37.735925 37.735925
10 22 42.426407 42.426407 42.426407
10 23 36.055513 36.055513 36.055513
10 24 41.036569 41.036569 41.036569
10 25 34.409301 34.409301 34.409301
10 26 39.051248 39.051248 39.051248
10 27 85.146932 85.146932 85.146932
10 28 85.000000 85.000000 85.000000
10 29 82.152298 82.152298 82.152298
10 30 80.000000 80.000000 80.000000
10 31 78.160092 78.160092 78.160092
10 32 78.000000 78.000000 78.000000
10 33 77.162167 77.162167 77.162167
10 34 75.663730 75.663730 75.663730
10 35 75.000000 75.000000 75.000000
10 36 75.822160 75.822160 75.822160
10 37 74.330344 74.330344 74.330344
10 38 72.346389 72.346389 72.346389
10 39 68.767725 68.767725 68.767725
10 40 67.268120 67.268120 67.268120
10 41 70.710678 70.710678 70.710678
10 42 62.481997 62.481997 62.481997
10 43 63.639610 63.639610 63.639610
10 44 67.268120 67.268120 67.268120
10 45 65.069194 65.069194 65.069194
10 46 48.052055 48.052055 48.052055
10 47 45.705580 45.705580 45.705580
10 48 12.806248 12.806248 12.806248
10 49 43.863424 43.863424 43.863424
10 50 39.408121 39.408121 39.408121
10 51 62.000000 62.000000 62.000000
10 52 47.434165 47.434165 47.434165
10 53 15.811388 15.811388 15.811388
10 54 18.027756 18.027756 18.027756
10 55 51.478151 51.478151 51.478151
10 56 32.015621 32.015621 32.015621
10 57 40.000000 40.000000 40.000000
10 58 22.360680 22.360680 22.360680
10 59 25.495098 25.495098 25.495098
10 60 15.000000 15.000000 15.000000
10 61 25.495098 25.495098 25.495098
10 62 46.097722 46.097722 46.097722
10 63 55.000000 55.000000 55.000000
10 64 57.008771 57.008771 57.008771
10 65 35.355339 35.355339 35.355339
10 66 25.495098 25.495098 25.495098
10 67 31.064449 31.064449 31.064449
10 68 54.451814 54.451814 54.451814
10 69 39.051248 39.051248 39.051248
10 70 27.018512 27.018512 27.018512
10 71 42.201896 42.201896 42.201896
10 72 58.523500 58.523500 58.523500
10 73 60.901560 60.901560 60.901560
10 74 26.248809 26.248809 26.248809
10 75 18.027756 18.027756 18.027756
10 76 30.413813 30.413813 30.413813
10 77 55.036352 55.036352 55.036352
10 78 34.539832 34.539832 34.539832
10 79 21.095023 21.095023 21.095023
10 80 33.241540 33.241540 33.241540
10 81 38.897301 38.897301 38.897301
10 82 45.276926 45.276926 45.276926

10 83 18.788294 18.788294 18.788294
10 84 27.294688 27.294688 27.294688
10 85 47.381431 47.381431 47.381431
10 86 54.341513 54.341513 54.341513
10 87 15.556349 15.556349 15.556349
10 88 11.180340 11.180340 11.180340
10 89 26.925824 26.925824 26.925824
10 90 64.412732 64.412732 64.412732
10 91 29.546573 29.546573 29.546573
10 92 39.623226 39.623226 39.623226
10 93 43.737855 43.737855 43.737855
10 94 53.758720 53.758720 53.758720
10 95 48.764741 48.764741 48.764741
10 96 46.043458 46.043458 46.043458
10 97 48.846699 48.846699 48.846699
10 98 18.027756 18.027756 18.027756
10 99 23.345235 23.345235 23.345235
10 100 16.000000 16.000000 16.000000
10 101 38.275318 38.275318 38.275318
11 1 31.622777 31.622777 31.622777
11 2 47.434165 47.434165 47.434165
11 3 37.000000 37.000000 37.000000
11 4 46.572524 46.572524 46.572524
11 5 41.231056 41.231056 41.231056
11 6 46.097722 46.097722 46.097722
11 7 35.902646 35.902646 35.902646
11 8 35.355339 35.355339 35.355339
11 9 40.311289 40.311289 40.311289
11 10 5.000000 5.000000 5.000000
11 12 2.000000 2.000000 2.000000
11 13 5.385165 5.385165 5.385165
11 14 7.071068 7.071068 7.071068
11 15 7.071068 7.071068 7.071068
11 16 8.000000 8.000000 8.000000
11 17 10.000000 10.000000 10.000000
11 18 11.180340 11.180340 11.180340
11 19 48.795492 48.795492 48.795492
11 20 43.863424 43.863424 43.863424
11 21 40.607881 40.607881 40.607881
11 22 46.097722 46.097722 46.097722
11 23 39.051248 39.051248 39.051248
11 24 44.821870 44.821870 44.821870
11 25 37.536649 37.536649 37.536649
11 26 43.011626 43.011626 43.011626
11 27 85.586214 85.586214 85.586214
11 28 85.146932 85.146932 85.146932
11 29 82.607506 82.607506 82.607506
11 30 80.156098 80.156098 80.156098
11 31 78.638413 78.638413 78.638413
11 32 78.160092 78.160092 78.160092
11 33 77.646635 77.646635 77.646635
11 34 76.485293 76.485293 76.485293
11 35 75.166482 75.166482 75.166482
11 36 72.622311 72.622311 72.622311
11 37 71.063352 71.063352 71.063352
11 38 69.202601 69.202601 69.202601
11 39 65.604878 65.604878 65.604878
11 40 64.031242 64.031242 64.031242
11 41 67.268120 67.268120 67.268120
11 42 59.405387 59.405387 59.405387
11 43 60.207973 60.207973 60.207973
11 44 63.639610 63.639610 63.639610
11 45 61.554854 61.554854 61.554854
11 46 43.174066 43.174066 43.174066
11 47 40.792156 40.792156 40.792156
11 48 9.433981 9.433981 9.433981
11 49 47.423623 47.423623 47.423623
11 50 42.520583 42.520583 42.520583
11 51 62.201286 62.201286 62.201286
11 52 49.244289 49.244289 49.244289
11 53 18.027756 18.027756 18.027756
11 54 14.142136 14.142136 14.142136
11 55 49.244289 49.244289 49.244289
11 56 28.284271 28.284271 28.284271
11 57 40.311289 40.311289 40.311289
11 58 25.000000 25.000000 25.000000
11 59 30.413813 30.413813 30.413813
11 60 20.000000 20.000000 20.000000
11 61 20.615528 20.615528 20.615528
11 62 43.011626 43.011626 43.011626
11 63 55.226805 55.226805 55.226805
11 64 58.523500 58.523500 58.523500
11 65 36.400549 36.400549 36.400549
11 66 25.000000 25.000000 25.000000
11 67 31.144823 31.144823 31.144823
11 68 54.037024 54.037024 54.037024
11 69 36.055513 36.055513 36.055513
11 70 24.186773 24.186773 24.186773
11 71 38.288379 38.288379 38.288379
11 72 57.008771 57.008771 57.008771
11 73 58.600341 58.600341 58.600341
11 74 21.540659 21.540659 21.540659
11 75 22.360680 22.360680 22.360680
11 76 35.355339 35.355339 35.355339
11 77 57.306195 57.306195 57.306195

11 78 39.217343 39.217343 39.217343
11 79 16.124515 16.124515 16.124515
11 80 28.284271 28.284271 28.284271
11 81 37.656341 37.656341 37.656341
11 82 42.953463 42.953463 42.953463
11 83 17.262677 17.262677 17.262677
11 84 28.460499 28.460499 28.460499
11 85 48.270074 48.270074 48.270074
11 86 55.659680 55.659680 55.659680
11 87 19.416488 19.416488 19.416488
11 88 16.124515 16.124515 16.124515
11 89 22.803509 22.803509 22.803509
11 90 66.887966 66.887966 66.887966
11 91 27.892651 27.892651 27.892651
11 92 39.051248 39.051248 39.051248
11 93 43.104524 43.104524 43.104524
11 94 52.392748 52.392748 52.392748
11 95 47.675990 47.675990 47.675990
11 96 46.097722 46.097722 46.097722
11 97 47.127487 47.127487 47.127487
11 98 22.803509 22.803509 22.803509
11 99 20.000000 20.000000 20.000000
11 100 16.763055 16.763055 16.763055
11 101 34.205263 34.205263 34.205263
12 1 33.526109 33.526109 33.526109
12 2 48.104054 48.104054 48.104054
12 3 37.696154 37.696154 37.696154
12 4 47.127487 47.127487 47.127487
12 5 41.761226 41.761226 41.761226
12 6 46.572524 46.572524 46.572524
12 7 36.400549 36.400549 36.400549
12 8 35.693137 35.693137 35.693137
12 9 40.607881 40.607881 40.607881
12 10 5.385165 5.385165 5.385165
12 11 2.000000 2.000000 2.000000
12 13 5.000000 5.000000 5.000000
12 14 5.830952 5.830952 5.830952
12 15 5.830952 5.830952 5.830952
12 16 6.000000 6.000000 6.000000
12 17 8.000000 8.000000 8.000000
12 18 9.433981 9.433981 9.433981
12 19 50.209561 50.209561 50.209561
12 20 45.343136 45.343136 45.343136
12 21 42.201896 42.201896 42.201896
12 22 47.423623 47.423623 47.423623
12 23 40.607881 40.607881 40.607881
12 24 46.097722 46.097722 46.097722
12 25 39.051248 39.051248 39.051248
12 26 44.204072 44.204072 44.204072
12 27 87.572827 87.572827 87.572827
12 28 87.143560 87.143560 87.143560
12 29 84.593144 84.593144 84.593144
12 30 82.152298 82.152298 82.152298
12 31 80.622577 80.622577 80.622577
12 32 80.156098 80.156098 80.156098
12 33 79.630396 79.630396 79.630396
12 34 78.447435 78.447435 78.447435
12 35 77.162167 77.162167 77.162167
12 36 74.202426 74.202426 74.202426
12 37 72.622311 72.622311 72.622311
12 38 70.802542 70.802542 70.802542
12 39 67.201190 67.201190 67.201190
12 40 65.604878 65.604878 65.604878
12 41 68.767725 68.767725 68.767725
12 42 61.032778 61.032778 61.032778
12 43 61.717096 61.717096 61.717096
12 44 65.069194 65.069194 65.069194
12 45 63.031738 63.031738 63.031738
12 46 43.680659 43.680659 43.680659
12 47 41.231056 41.231056 41.231056
12 48 7.810250 7.810250 7.810250
12 49 48.795492 48.795492 48.795492
12 50 44.045431 44.045431 44.045431
12 51 64.195015 64.195015 64.195015
12 52 51.078371 51.078371 51.078371
12 53 19.723083 19.723083 19.723083
12 54 15.620499 15.620499 15.620499
12 55 51.078371 51.078371 51.078371
12 56 29.732137 29.732137 29.732137
12 57 42.296572 42.296572 42.296572
12 58 26.627054 26.627054 26.627054
12 59 30.805844 30.805844 30.805844
12 60 20.099751 20.099751 20.099751
12 61 21.189620 21.189620 21.189620
12 62 44.654227 44.654227 44.654227
12 63 57.218878 57.218878 57.218878
12 64 60.406953 60.406953 60.406953
12 65 38.327536 38.327536 38.327536
12 66 27.000000 27.000000 27.000000
12 67 33.136083 33.136083 33.136083
12 68 56.035703 56.035703 56.035703
12 69 37.735925 37.735925 37.735925
12 70 25.942244 25.942244 25.942244
12 71 39.623226 39.623226 39.623226
12 72 58.940648 58.940648 58.940648

12 73 60.415230 60.415230 60.415230
12 74 20.880613 20.880613 20.880613
12 75 23.323808 23.323808 23.323808
12 76 35.128336 35.128336 35.128336
12 77 59.059292 59.059292 59.059292
12 78 39.924930 39.924930 39.924930
12 79 16.000000 16.000000 16.000000
12 80 28.071338 28.071338 28.071338
12 81 39.623226 39.623226 39.623226
12 82 44.777226 44.777226 44.777226
12 83 19.235384 19.235384 19.235384
12 84 30.364453 30.364453 30.364453
12 85 50.219518 50.219518 50.219518
12 86 57.567352 57.567352 57.567352
12 87 20.615528 20.615528 20.615528
12 88 16.492423 16.492423 16.492423
12 89 24.083189 24.083189 24.083189
12 90 68.600292 68.600292 68.600292
12 91 29.832868 29.832868 29.832868
12 92 41.048752 41.048752 41.048752
12 93 45.099889 45.099889 45.099889
12 94 54.341513 54.341513 54.341513
12 95 49.648766 49.648766 49.648766
12 96 48.093659 48.093659 48.093659
12 97 49.040799 49.040799 49.040799
12 98 22.360680 22.360680 22.360680
12 99 21.633308 21.633308 21.633308
12 100 18.681542 18.681542 18.681542
12 101 35.468296 35.468296 35.468296
13 1 32.388269 32.388269 32.388269
13 2 43.462628 43.462628 43.462628
13 3 33.105891 33.105891 33.105891
13 4 42.379240 42.379240 42.379240
13 5 37.000000 37.000000 37.000000
13 6 41.761226 41.761226 41.761226
13 7 31.622777 31.622777 31.622777
13 8 30.805844 30.805844 30.805844
13 9 35.693137 35.693137 35.693137
13 10 10.198039 10.198039 10.198039
13 11 5.385165 5.385165 5.385165
13 12 5.000000 5.000000 5.000000
13 14 10.440307 10.440307 10.440307
13 15 3.000000 3.000000 3.000000
13 16 7.810250 7.810250 7.810250
13 17 9.433981 9.433981 9.433981
13 18 8.000000 8.000000 8.000000
13 19 53.814496 53.814496 53.814496
13 20 48.795492 48.795492 48.795492
13 21 45.343136 45.343136 45.343136
13 22 51.224994 51.224994 51.224994
13 23 43.863424 43.863424 43.863424
13 24 50.000000 50.000000 50.000000
13 25 42.426407 42.426407 42.426407
13 26 48.259714 48.259714 48.259714
13 27 88.283634 88.283634 88.283634
13 28 87.572827 87.572827 87.572827
13 29 85.328776 85.328776 85.328776
13 30 82.607506 82.607506 82.607506
13 31 81.394103 81.394103 81.394103
13 32 80.622577 80.622577 80.622577
13 33 80.411442 80.411442 80.411442
13 34 79.555012 79.555012 79.555012
13 35 77.646635 77.646635 77.646635
13 36 71.281134 71.281134 71.281134
13 37 69.634761 69.634761 69.634761
13 38 67.955868 67.955868 67.955868
13 39 64.350602 64.350602 64.350602
13 40 62.681736 62.681736 62.681736
13 41 65.604878 65.604878 65.604878
13 42 58.309519 58.309519 58.309519
13 43 58.600341 58.600341 58.600341
13 44 61.717096 61.717096 61.717096
13 45 59.816386 59.816386 59.816386
13 46 38.897301 38.897301 38.897301
13 47 36.400549 36.400549 36.400549
13 48 6.000000 6.000000 6.000000
13 49 52.497619 52.497619 52.497619
13 50 47.381431 47.381431 47.381431
13 51 64.776539 64.776539 64.776539
13 52 53.235327 53.235327 53.235327
13 53 22.671568 22.671568 22.671568
13 54 13.000000 13.000000 13.000000
13 55 49.335586 49.335586 49.335586
13 56 26.627054 26.627054 26.627054
13 57 43.174066 43.174066 43.174066
13 58 29.732137 29.732137 29.732137
13 59 35.693137 35.693137 35.693137
13 60 25.079872 25.079872 25.079872
13 61 16.552945 16.552945 16.552945
13 62 42.059482 42.059482 42.059482
13 63 57.870545 57.870545 57.870545
13 64 62.241465 62.241465 62.241465
13 65 39.924930 39.924930 39.924930
13 66 27.459060 27.459060 27.459060
13 67 33.955854 33.955854 33.955854

13 68 56.080300 56.080300 56.080300
13 69 35.341194 35.341194 35.341194
13 70 24.041631 24.041631 24.041631
13 71 36.124784 36.124784 36.124784
13 72 57.870545 57.870545 57.870545
13 73 58.523500 58.523500 58.523500
13 74 16.155494 16.155494 16.155494
13 75 27.730849 27.730849 27.730849
13 76 40.112342 40.112342 40.112342
13 77 61.587336 61.587336 61.587336
13 78 44.598206 44.598206 44.598206
13 79 11.000000 11.000000 11.000000
13 80 23.086793 23.086793 23.086793
13 81 39.051248 39.051248 39.051248
13 82 43.011626 43.011626 43.011626
13 83 19.104973 19.104973 19.104973
13 84 32.202484 32.202484 32.202484
13 85 51.546096 51.546096 51.546096
13 86 59.236813 59.236813 59.236813
13 87 24.698178 24.698178 24.698178
13 88 21.377558 21.377558 21.377558
13 89 20.615528 20.615528 20.615528
13 90 71.281134 71.281134 71.281134
13 91 29.068884 29.068884 29.068884
13 92 41.109610 41.109610 41.109610
13 93 45.044423 45.044423 45.044423
13 94 53.460266 53.460266 53.460266
13 95 49.091751 49.091751 49.091751
13 96 48.662100 48.662100 48.662100
13 97 47.853944 47.853944 47.853944
13 98 27.294688 27.294688 27.294688
13 99 19.313208 19.313208 19.313208
13 100 20.591260 20.591260 20.591260
13 101 31.827661 31.827661 31.827661
14 1 38.078866 38.078866 38.078866
14 2 53.851648 53.851648 53.851648
14 3 43.462628 43.462628 43.462628
14 4 52.810984 52.810984 52.810984
14 5 47.434165 47.434165 47.434165
14 6 52.201533 52.201533 52.201533
14 7 42.059482 42.059482 42.059482
14 8 41.231056 41.231056 41.231056
14 9 46.097722 46.097722 46.097722
14 10 5.000000 5.000000 5.000000
14 11 7.071068 7.071068 7.071068
14 12 5.830952 5.830952 5.830952
14 13 10.440307 10.440307 10.440307
14 15 10.000000 10.000000 10.000000
14 16 5.830952 5.830952 5.830952
14 17 7.071068 7.071068 7.071068
14 18 11.180340 11.180340 11.180340
14 19 49.203658 49.203658 49.203658
14 20 44.654227 44.654227 44.654227
14 21 42.059482 42.059482 42.059482
14 22 46.097722 46.097722 46.097722
14 23 40.311289 40.311289 40.311289
14 24 44.598206 44.598206 44.598206
14 25 38.587563 38.587563 38.587563
14 26 42.426407 42.426407 42.426407
14 27 90.138782 90.138782 90.138782
14 28 90.000000 90.000000 90.000000
14 29 87.143560 87.143560 87.143560
14 30 85.000000 85.000000 85.000000
14 31 83.150466 83.150466 83.150466
14 32 83.000000 83.000000 83.000000
14 33 82.152298 82.152298 82.152298
14 34 80.622577 80.622577 80.622577
14 35 80.000000 80.000000 80.000000
14 36 79.649231 79.649231 79.649231
14 37 78.102497 78.102497 78.102497
14 38 76.216796 76.216796 76.216796
14 39 72.622311 72.622311 72.622311
14 40 71.063352 71.063352 71.063352
14 41 74.330344 74.330344 74.330344
14 42 66.400301 66.400301 66.400301
14 43 67.268120 67.268120 67.268120
14 44 70.710678 70.710678 70.710678
14 45 68.622154 68.622154 68.622154
14 46 49.335586 49.335586 49.335586
14 47 46.840154 46.840154 46.840154
14 48 10.440307 10.440307 10.440307
14 49 47.634021 47.634021 47.634021
14 50 43.566042 43.566042 43.566042
14 51 67.000000 67.000000 67.000000
14 52 52.201533 52.201533 52.201533
14 53 20.615528 20.615528 20.615528
14 54 21.213203 21.213203 21.213203
14 55 55.901699 55.901699 55.901699
14 56 35.355339 35.355339 35.355339
14 57 45.000000 45.000000 45.000000
14 58 26.925824 26.925824 26.925824
14 59 26.925824 26.925824 26.925824
14 60 15.811388 15.811388 15.811388
14 61 26.925824 26.925824 26.925824
14 62 50.000000 50.000000 50.000000

14 63 60.000000 60.000000 60.000000
14 64 61.846584 61.846584 61.846584
14 65 40.311289 40.311289 40.311289
14 66 30.413813 30.413813 30.413813
14 67 36.055513 36.055513 36.055513
14 68 59.413803 59.413803 59.413803
14 69 43.011626 43.011626 43.011626
14 70 31.064449 31.064449 31.064449
14 71 45.343136 45.343136 45.343136
14 72 63.245553 63.245553 63.245553
14 73 65.299311 65.299311 65.299311
14 74 25.179357 25.179357 25.179357
14 75 21.213203 21.213203 21.213203
14 76 30.000000 30.000000 30.000000
14 77 59.615434 59.615434 59.615434
14 78 36.715120 36.715120 36.715120
14 79 21.213203 21.213203 21.213203
14 80 33.015148 33.015148 33.015148
14 81 43.680659 43.680659 43.680659
14 82 49.648766 49.648766 49.648766
14 83 23.409400 23.409400 23.409400
14 84 32.249031 32.249031 32.249031
14 85 52.345009 52.345009 52.345009
14 86 59.228372 59.228372 59.228372
14 87 19.416488 19.416488 19.416488
14 88 13.038405 13.038405 13.038405
14 89 29.832868 29.832868 29.832868
14 90 68.876701 68.876701 68.876701
14 91 34.176015 34.176015 34.176015
14 92 44.553339 44.553339 44.553339
14 93 48.662100 48.662100 48.662100
14 94 58.523500 58.523500 58.523500
14 95 53.600373 53.600373 53.600373
14 96 51.039201 51.039201 51.039201
14 97 53.488316 53.488316 53.488316
14 98 17.029386 17.029386 17.029386
14 99 27.018512 27.018512 27.018512
14 100 21.000000 21.000000 21.000000
14 101 41.231056 41.231056 41.231056
15 1 35.355339 35.355339 35.355339
15 2 44.721360 44.721360 44.721360
15 3 34.481879 34.481879 34.481879
15 4 43.462628 43.462628 43.462628
15 5 38.078866 38.078866 38.078866
15 6 42.720019 42.720019 42.720019
15 7 32.695565 32.695565 32.695565
15 8 31.622777 31.622777 31.622777
15 9 36.400549 36.400549 36.400549
15 10 11.180340 11.180340 11.180340
15 11 7.071068 7.071068 7.071068
15 12 5.830952 5.830952 5.830952
15 13 3.000000 3.000000 3.000000
15 14 10.000000 10.000000 10.000000
15 16 5.830952 5.830952 5.830952
15 17 7.071068 7.071068 7.071068
15 18 5.000000 5.000000 5.000000
15 19 55.865911 55.865911 55.865911
15 20 50.931326 50.931326 50.931326
15 21 47.634021 47.634021 47.634021
15 22 53.150729 53.150729 53.150729
15 23 46.097722 46.097722 46.097722
15 24 51.855569 51.855569 51.855569
15 25 44.598206 44.598206 44.598206
15 26 50.000000 50.000000 50.000000
15 27 91.241438 91.241438 91.241438
15 28 90.553851 90.553851 90.553851
15 29 88.283634 88.283634 88.283634
15 30 85.586214 85.586214 85.586214
15 31 84.344532 84.344532 84.344532
15 32 83.600239 83.600239 83.600239
15 33 83.360662 83.360662 83.360662
15 34 82.462113 82.462113 82.462113
15 35 80.622577 80.622577 80.622577
15 36 73.783467 73.783467 73.783467
15 37 72.111026 72.111026 72.111026
15 38 70.491134 70.491134 70.491134
15 39 66.887966 66.887966 66.887966
15 40 65.192024 65.192024 65.192024
15 41 68.007353 68.007353 68.007353
15 42 60.901560 60.901560 60.901560
15 43 61.032778 61.032778 61.032778
15 44 64.031242 64.031242 64.031242
15 45 62.201286 62.201286 62.201286
15 46 39.924930 39.924930 39.924930
15 47 37.336309 37.336309 37.336309
15 48 3.000000 3.000000 3.000000
15 49 54.488531 54.488531 54.488531
15 50 49.578221 49.578221 49.578221
15 51 67.742158 67.742158 67.742158
15 52 55.901699 55.901699 55.901699
15 53 25.000000 25.000000 25.000000
15 54 15.811388 15.811388 15.811388
15 55 52.201533 52.201533 52.201533
15 56 29.154759 29.154759 29.154759
15 57 46.097722 46.097722 46.097722

15 58 32.015621 32.015621 32.015621
15 59 36.400549 36.400549 36.400549
15 60 25.495098 25.495098 25.495098
15 61 18.027756 18.027756 18.027756
15 62 44.721360 44.721360 44.721360
15 63 60.827625 60.827625 60.827625
15 64 65.000000 65.000000 65.000000
15 65 42.720019 42.720019 42.720019
15 66 30.413813 30.413813 30.413813
15 67 36.878178 36.878178 36.878178
15 68 59.076222 59.076222 59.076222
15 69 38.078866 38.078866 38.078866
15 70 26.925824 26.925824 26.925824
15 71 38.418745 38.418745 38.418745
15 72 60.827625 60.827625 60.827625
15 73 61.351447 61.351447 61.351447
15 74 15.297059 15.297059 15.297059
15 75 29.154759 29.154759 29.154759
15 76 40.000000 40.000000 40.000000
15 77 64.140471 64.140471 64.140471
15 78 45.694639 45.694639 45.694639
15 79 11.401754 11.401754 11.401754
15 80 23.021729 23.021729 23.021729
15 81 42.047592 42.047592 42.047592
15 82 45.880279 45.880279 45.880279
15 83 22.090722 22.090722 22.090722
15 84 34.928498 34.928498 34.928498
15 85 54.405882 54.405882 54.405882
15 86 62.032250 62.032250 62.032250
15 87 26.400758 26.400758 26.400758
15 88 22.135944 22.135944 22.135944
15 89 23.021729 23.021729 23.021729
15 90 73.783467 73.783467 73.783467
15 91 32.062439 32.062439 32.062439
15 92 44.102154 44.102154 44.102154
15 93 48.041649 48.041649 48.041649
15 94 56.435804 56.435804 56.435804
15 95 52.086467 52.086467 52.086467
15 96 51.623638 51.623638 51.623638
15 97 50.803543 50.803543 50.803543
15 98 27.018512 27.018512 27.018512
15 99 22.135944 22.135944 22.135944
15 100 23.259407 23.259407 23.259407
15 101 34.058773 34.058773 34.058773
16 1 39.293765 39.293765 39.293765
16 2 50.537115 50.537115 50.537115
16 3 40.311289 40.311289 40.311289
16 4 49.244289 49.244289 49.244289
16 5 43.863424 43.863424 43.863424
16 6 48.466483 48.466483 48.466483
16 7 38.483763 38.483763 38.483763
16 8 37.336309 37.336309 37.336309
16 9 42.059482 42.059482 42.059482
16 10 9.433981 9.433981 9.433981
16 11 8.000000 8.000000 8.000000
16 12 6.000000 6.000000 6.000000
16 13 7.810250 7.810250 7.810250
16 14 5.830952 5.830952 5.830952
16 15 5.830952 5.830952 5.830952
16 17 2.000000 2.000000 2.000000
16 18 5.385165 5.385165 5.385165
16 19 54.671748 54.671748 54.671748
16 20 50.000000 50.000000 50.000000
16 21 47.169906 47.169906 47.169906
16 22 51.662365 51.662365 51.662365
16 23 45.486262 45.486262 45.486262
16 24 50.209561 50.209561 50.209561
16 25 43.829214 43.829214 43.829214
16 26 48.104054 48.104054 48.104054
16 27 93.536089 93.536089 93.536089
16 28 93.134312 93.134312 93.134312
16 29 90.553851 90.553851 90.553851
16 30 88.141931 88.141931 88.141931
16 31 86.579443 86.579443 86.579443
16 32 86.145226 86.145226 86.145226
16 33 85.586214 85.586214 85.586214
16 34 84.344532 84.344532 84.344532
16 35 83.150466 83.150466 83.150466
16 36 79.056942 79.056942 79.056942
16 37 77.420927 77.420927 77.420927
16 38 75.716577 75.716577 75.716577
16 39 72.111026 72.111026 72.111026
16 40 70.455660 70.455660 70.455660
16 41 73.409809 73.409809 73.409809
16 42 66.037868 66.037868 66.037868
16 43 66.400301 66.400301 66.400301
16 44 69.526973 69.526973 69.526973
16 45 67.623960 67.623960 67.623960
16 46 45.694639 45.694639 45.694639
16 47 43.081318 43.081318 43.081318
16 48 5.000000 5.000000 5.000000
16 49 53.150729 53.150729 53.150729
16 50 48.826222 48.826222 48.826222
16 51 70.178344 70.178344 70.178344
16 52 56.648036 56.648036 56.648036

16 53 25.079872 25.079872 25.079872
16 54 20.591260 20.591260 20.591260
16 55 56.648036 56.648036 56.648036
16 56 34.409301 34.409301 34.409301
16 57 48.259714 48.259714 48.259714
16 58 31.764760 31.764760 31.764760
16 59 32.695565 32.695565 32.695565
16 60 21.540659 21.540659 21.540659
16 61 23.853721 23.853721 23.853721
16 62 49.739320 49.739320 49.739320
16 63 63.198101 63.198101 63.198101
16 64 66.098411 66.098411 66.098411
16 65 44.147480 44.147480 44.147480
16 66 33.000000 33.000000 33.000000
16 67 39.115214 39.115214 39.115214
16 68 62.032250 62.032250 62.032250
16 69 42.941821 42.941821 42.941821
16 70 31.384710 31.384710 31.384710
16 71 43.931765 43.931765 43.931765
16 72 64.761099 64.761099 64.761099
16 73 65.924199 65.924199 65.924199
16 74 20.000000 20.000000 20.000000
16 75 26.907248 26.907248 26.907248
16 76 35.128336 35.128336 35.128336
16 77 64.404969 64.404969 64.404969
16 78 42.544095 42.544095 42.544095
16 79 17.088007 17.088007 17.088007
16 80 28.284271 28.284271 28.284271
16 81 45.541190 45.541190 45.541190
16 82 50.328918 50.328918 50.328918
16 83 25.179357 25.179357 25.179357
16 84 36.138622 36.138622 36.138622
16 85 56.089215 56.089215 56.089215
16 86 63.324561 63.324561 63.324561
16 87 24.839485 24.839485 24.839485
16 88 18.867962 18.867962 18.867962
16 89 28.425341 28.425341 28.425341
16 90 73.824115 73.824115 73.824115
16 91 35.693137 35.693137 35.693137
16 92 47.042534 47.042534 47.042534
16 93 51.088159 51.088159 51.088159
16 94 60.207973 60.207973 60.207973
16 95 55.578773 55.578773 55.578773
16 96 54.083269 54.083269 54.083269
16 97 54.817880 54.817880 54.817880
16 98 22.090722 22.090722 22.090722
16 99 26.832816 26.832816 26.832816
16 100 24.515301 24.515301 24.515301
16 101 39.623226 39.623226 39.623226
17 1 41.231056 41.231056 41.231056
17 2 51.478151 51.478151 51.478151
17 3 41.340053 41.340053 41.340053
17 4 50.089919 50.089919 50.089919
17 5 44.721360 44.721360 44.721360
17 6 49.244289 49.244289 49.244289
17 7 39.357337 39.357337 39.357337
17 8 38.078866 38.078866 38.078866
17 9 42.720019 42.720019 42.720019
17 10 11.180340 11.180340 11.180340
17 11 10.000000 10.000000 10.000000
17 12 8.000000 8.000000 8.000000
17 13 9.433981 9.433981 9.433981
17 14 7.071068 7.071068 7.071068
17 15 7.071068 7.071068 7.071068
17 16 2.000000 2.000000 2.000000
17 18 5.000000 5.000000 5.000000
17 19 56.222771 56.222771 56.222771
17 20 51.613952 51.613952 51.613952
17 21 48.877398 48.877398 48.877398
17 22 53.150729 53.150729 53.150729
17 23 47.169906 47.169906 47.169906
17 24 51.662365 51.662365 51.662365
17 25 45.486262 45.486262 45.486262
17 26 49.497475 49.497475 49.497475
17 27 95.524866 95.524866 95.524866
17 28 95.131488 95.131488 95.131488
17 29 92.541882 92.541882 92.541882
17 30 90.138782 90.138782 90.138782
17 31 88.566359 88.566359 88.566359
17 32 88.141931 88.141931 88.141931
17 33 87.572827 87.572827 87.572827
17 34 86.313383 86.313383 86.313383
17 35 85.146932 85.146932 85.146932
17 36 80.709355 80.709355 80.709355
17 37 79.056942 79.056942 79.056942
17 38 77.388630 77.388630 77.388630
17 39 73.783467 73.783467 73.783467
17 40 72.111026 72.111026 72.111026
17 41 75.000000 75.000000 75.000000
17 42 67.742158 67.742158 67.742158
17 43 68.007353 68.007353 68.007353
17 44 71.063352 71.063352 71.063352
17 45 69.202601 69.202601 69.202601
17 46 46.518813 46.518813 46.518813
17 47 43.863424 43.863424 43.863424

17 48 5.385165 5.385165 5.385165
17 49 54.671748 54.671748 54.671748
17 50 50.477718 50.477718 50.477718
17 51 72.173402 72.173402 72.173402
17 52 58.523500 58.523500 58.523500
17 53 26.925824 26.925824 26.925824
17 54 22.360680 22.360680 22.360680
17 55 58.523500 58.523500 58.523500
17 56 36.055513 36.055513 36.055513
17 57 50.249378 50.249378 50.249378
17 58 33.541020 33.541020 33.541020
17 59 33.541020 33.541020 33.541020
17 60 22.360680 22.360680 22.360680
17 61 25.000000 25.000000 25.000000
17 62 51.478151 51.478151 51.478151
17 63 65.192024 65.192024 65.192024
17 64 68.007353 68.007353 68.007353
17 65 46.097722 46.097722 46.097722
17 66 35.000000 35.000000 35.000000
17 67 41.109610 41.109610 41.109610
17 68 64.031242 64.031242 64.031242
17 69 44.721360 44.721360 44.721360
17 70 33.241540 33.241540 33.241540
17 71 45.453273 45.453273 45.453273
17 72 66.708320 66.708320 66.708320
17 73 67.779053 67.779053 67.779053
17 74 20.099751 20.099751 20.099751
17 75 28.284271 28.284271 28.284271
17 76 35.355339 35.355339 35.355339
17 77 66.211781 66.211781 66.211781
17 78 43.566042 43.566042 43.566042
17 79 17.888544 17.888544 17.888544
17 80 28.635642 28.635642 28.635642
17 81 47.518417 47.518417 47.518417
17 82 52.201533 52.201533 52.201533
17 83 27.166155 27.166155 27.166155
17 84 38.078866 38.078866 38.078866
17 85 58.051701 58.051701 58.051701
17 86 65.253352 65.253352 65.253352
17 87 26.400758 26.400758 26.400758
17 88 20.000000 20.000000 20.000000
17 89 30.000000 30.000000 30.000000
17 90 75.591005 75.591005 75.591005
17 91 37.656341 37.656341 37.656341
17 92 49.040799 49.040799 49.040799
17 93 53.084838 53.084838 53.084838
17 94 62.169124 62.169124 62.169124
17 95 57.558666 57.558666 57.558666
17 96 56.080300 56.080300 56.080300
17 97 56.753854 56.753854 56.753854
17 98 22.360680 22.360680 22.360680
17 99 28.635642 28.635642 28.635642
17 100 26.476405 26.476405 26.476405
17 101 41.109610 41.109610 41.109610
18 1 40.311289 40.311289 40.311289
18 2 47.169906 47.169906 47.169906
18 3 37.202150 37.202150 37.202150
18 4 45.650849 45.650849 45.650849
18 5 40.311289 40.311289 40.311289
18 6 44.721360 44.721360 44.721360
18 7 34.985711 34.985711 34.985711
18 8 33.541020 33.541020 33.541020
18 9 38.078866 38.078866 38.078866
18 10 14.142136 14.142136 14.142136
18 11 11.180340 11.180340 11.180340
18 12 9.433981 9.433981 9.433981
18 13 8.000000 8.000000 8.000000
18 14 11.180340 11.180340 11.180340
18 15 5.000000 5.000000 5.000000
18 16 5.385165 5.385165 5.385165
18 17 5.000000 5.000000 5.000000
18 19 59.464275 59.464275 59.464275
18 20 54.671748 54.671748 54.671748
18 21 51.613952 51.613952 51.613952
18 22 56.568542 56.568542 56.568542
18 23 50.000000 50.000000 50.000000
18 24 55.172457 55.172457 55.172457
18 25 48.414874 48.414874 48.414874
18 26 53.150729 53.150729 53.150729
18 27 96.176920 96.176920 96.176920
18 28 95.524866 95.524866 95.524866
18 29 93.214806 93.214806 93.214806
18 30 90.553851 90.553851 90.553851
18 31 89.269256 89.269256 89.269256
18 32 88.566359 88.566359 88.566359
18 33 88.283634 88.283634 88.283634
18 34 87.321246 87.321246 87.321246
18 35 85.586214 85.586214 85.586214
18 36 78.032045 78.032045 78.032045
18 37 76.321688 76.321688 76.321688
18 38 74.793048 74.793048 74.793048
18 39 71.196910 71.196910 71.196910
18 40 69.462220 69.462220 69.462220
18 41 72.111026 72.111026 72.111026
18 42 65.299311 65.299311 65.299311
18 43 65.192024 65.192024 65.192024

18 43 65.192024 65.192024 65.192024
18 44 68.007353 68.007353 68.007353
18 45 66.287254 66.287254 66.287254
18 46 42.059482 42.059482 42.059482
18 47 39.357337 39.357337 39.357337
18 48 2.000000 2.000000 2.000000
18 49 58.000000 58.000000 58.000000
18 50 53.413481 53.413481 53.413481
18 51 72.691127 72.691127 72.691127
18 52 60.415230 60.415230 60.415230
18 53 29.154759 29.154759 29.154759
18 54 20.615528 20.615528 20.615528
18 55 57.008771 57.008771 57.008771
18 56 33.541020 33.541020 33.541020
18 57 50.990195 50.990195 50.990195
18 58 36.055513 36.055513 36.055513
18 59 38.078866 38.078866 38.078866
18 60 26.925824 26.925824 26.925824
18 61 21.213203 21.213203 21.213203
18 62 49.244289 49.244289 49.244289
18 63 65.764732 65.764732 65.764732
18 64 69.641941 69.641941 69.641941
18 65 47.434165 47.434165 47.434165
18 66 35.355339 35.355339 35.355339
18 67 41.773197 41.773197 41.773197
18 68 64.070274 64.070274 64.070274
18 69 42.720019 42.720019 42.720019
18 70 31.780497 31.780497 31.780497
18 71 42.438190 42.438190 42.438190
18 72 65.764732 65.764732 65.764732
18 73 66.098411 66.098411 66.098411
18 74 15.132746 15.132746 15.132746
18 75 32.015621 32.015621 32.015621
18 76 40.311289 40.311289 40.311289
18 77 68.476273 68.476273 68.476273
18 78 47.885280 47.885280 47.885280
18 79 13.601471 13.601471 13.601471
18 80 23.769729 23.769729 23.769729
18 81 47.042534 47.042534 47.042534
18 82 50.695167 50.695167 50.695167
18 83 27.073973 27.073973 27.073973
18 84 39.560081 39.560081 39.560081
18 85 59.203040 59.203040 59.203040
18 86 66.730802 66.730802 66.730802
18 87 29.698485 29.698485 29.698485
18 88 24.186773 24.186773 24.186773
18 89 27.294688 27.294688 27.294688
18 90 78.032045 78.032045 78.032045
18 91 37.054015 37.054015 37.054015
18 92 49.091751 49.091751 49.091751
18 93 53.037722 53.037722 53.037722
18 94 61.400326 61.400326 61.400326
18 95 57.078893 57.078893 57.078893
18 96 56.568542 56.568542 56.568542
18 97 55.731499 55.731499 55.731499
18 98 27.294688 27.294688 27.294688
18 99 26.925824 26.925824 26.925824
18 100 27.856777 27.856777 27.856777
18 101 38.013156 38.013156 38.013156
19 1 45.177428 45.177428 45.177428
19 2 82.225300 82.225300 82.225300
19 3 73.375745 73.375745 73.375745
19 4 82.969874 82.969874 82.969874
19 5 78.746428 78.746428 78.746428
19 6 83.522452 83.522452 83.522452
19 7 74.672619 74.672619 74.672619
19 8 75.769387 75.769387 75.769387
19 9 80.411442 80.411442 80.411442
19 10 45.343136 45.343136 45.343136
19 11 48.795492 48.795492 48.795492
19 12 50.209561 50.209561 50.209561
19 13 53.814496 53.814496 53.814496
19 14 49.203658 49.203658 49.203658
19 15 55.865911 55.865911 55.865911
19 16 54.671748 54.671748 54.671748
19 17 56.222771 56.222771 56.222771
19 18 59.464275 59.464275 59.464275
19 20 5.385165 5.385165 5.385165
19 21 10.198039 10.198039 10.198039
19 22 4.000000 4.000000 4.000000
19 23 10.770330 10.770330 10.770330
19 24 6.000000 6.000000 6.000000
19 25 11.661904 11.661904 11.661904
19 26 9.000000 9.000000 9.000000
19 27 56.797887 56.797887 56.797887
19 28 59.169249 59.169249 59.169249
19 29 54.120237 54.120237 54.120237
19 30 54.918121 54.918121 54.918121
19 31 50.606324 50.606324 50.606324
19 32 53.254108 53.254108 53.254108
19 33 49.739320 49.739320 49.739320
19 34 45.617979 45.617979 45.617979
19 35 50.803543 50.803543 50.803543
19 36 83.240615 83.240615 83.240615
19 37 82.710338 82.710338 82.710338
19 38 79.812280 79.812280 79.812280

19 38 79.812280 79.812280 79.812280
19 39 77.129761 77.129761 77.129761
19 40 76.687678 76.687678 76.687678
19 41 81.584312 81.584312 81.584312
19 42 71.386273 71.386273 71.386273
19 43 75.802375 75.802375 75.802375
19 44 80.752709 80.752709 80.752709
19 45 77.781746 77.781746 77.781746
19 46 80.653580 80.653580 80.653580
19 47 79.378838 79.378838 79.378838
19 48 58.000000 58.000000 58.000000
19 49 2.000000 2.000000 2.000000
19 50 7.280110 7.280110 7.280110
19 51 41.036569 41.036569 41.036569
19 52 18.601075 18.601075 18.601075
19 53 31.400637 31.400637 31.400637
19 54 51.000000 51.000000 51.000000
19 55 56.089215 56.089215 56.089215
19 56 56.753854 56.753854 56.753854
19 57 30.594117 30.594117 30.594117
19 58 24.413111 24.413111 24.413111
19 59 29.427878 29.427878 29.427878
19 60 37.161808 37.161808 37.161808
19 61 62.177166 62.177166 62.177166
19 62 60.008333 60.008333 60.008333
19 63 36.619667 36.619667 36.619667
19 64 25.806976 25.806976 25.806976
19 65 25.019992 25.019992 25.019992
19 66 36.138622 36.138622 36.138622
19 67 32.140317 32.140317 32.140317
19 68 42.059482 42.059482 42.059482
19 69 55.145263 55.145263 55.145263
19 70 48.764741 48.764741 48.764741
19 71 64.629715 64.629715 64.629715
19 72 54.230987 54.230987 54.230987
19 73 62.936476 62.936476 62.936476
19 74 69.202601 69.202601 69.202601
19 75 28.301943 28.301943 28.301943
19 76 39.000000 39.000000 39.000000
19 77 17.464249 17.464249 17.464249
19 78 21.095023 21.095023 21.095023
19 79 62.425956 62.425956 62.425956
19 80 73.573093 73.573093 73.573093
19 81 42.107007 42.107007 42.107007
19 82 53.235327 53.235327 53.235327
19 83 41.629317 41.629317 41.629317
19 84 26.925824 26.925824 26.925824
19 85 27.294688 27.294688 27.294688
19 86 26.172505 26.172505 26.172505
19 87 29.832868 29.832868 29.832868
19 88 37.215588 37.215588 37.215588
19 89 56.648036 56.648036 56.648036
19 90 23.000000 23.000000 23.000000
19 91 42.579338 42.579338 42.579338
19 92 37.336309 37.336309 37.336309
19 93 39.051248 39.051248 39.051248
19 94 49.979996 49.979996 49.979996
19 95 44.922155 44.922155 44.922155
19 96 34.176015 34.176015 34.176015
19 97 50.219518 50.219518 50.219518
19 98 42.059482 42.059482 42.059482
19 99 50.328918 50.328918 50.328918
19 100 34.985711 34.985711 34.985711
19 101 63.348244 63.348244 63.348244
20 1 40.049969 40.049969 40.049969
20 2 76.902536 76.902536 76.902536
20 3 68.007353 68.007353 68.007353
20 4 77.620873 77.620873 77.620873
20 5 73.375745 73.375745 73.375745
20 6 78.160092 78.160092 78.160092
20 7 69.289249 69.289249 69.289249
20 8 70.384657 70.384657 70.384657
20 9 75.026662 75.026662 75.026662
20 10 40.607881 40.607881 40.607881
20 11 43.863424 43.863424 43.863424
20 12 45.343136 45.343136 45.343136
20 13 48.795492 48.795492 48.795492
20 14 44.654227 44.654227 44.654227
20 15 50.931326 50.931326 50.931326
20 16 50.000000 50.000000 50.000000
20 17 51.613952 51.613952 51.613952
20 18 54.671748 54.671748 54.671748
20 19 5.385165 5.385165 5.385165
20 21 5.000000 5.000000 5.000000
20 22 5.385165 5.385165 5.385165
20 23 5.385165 5.385165 5.385165
20 24 6.403124 6.403124 6.403124
20 25 6.403124 6.403124 6.403124
20 26 8.602325 8.602325 8.602325
20 27 56.648036 56.648036 56.648036
20 28 58.600341 58.600341 58.600341
20 29 53.851648 53.851648 53.851648
20 30 54.120237 54.120237 54.120237
20 31 50.159745 50.159745 50.159745
20 32 52.354560 52.354560 52.354560
20 33 49.244289 49.244289 49.244289

20 34 45.541190 45.541190 45.541190
20 35 49.739320 49.739320 49.739320
20 36 79.056942 79.056942 79.056942
20 37 78.447435 78.447435 78.447435
20 38 75.584390 75.584390 75.584390
20 39 72.801099 72.801099 72.801099
20 40 72.277244 72.277244 72.277244
20 41 77.129761 77.129761 77.129761
20 42 66.940272 66.940272 66.940272
20 43 71.196910 71.196910 71.196910
20 44 76.118329 76.118329 76.118329
20 45 73.164199 73.164199 73.164199
20 46 75.286121 75.286121 75.286121
20 47 74.000000 74.000000 74.000000
20 48 53.150729 53.150729 53.150729
20 49 5.000000 5.000000 5.000000
20 50 2.000000 2.000000 2.000000
20 51 39.051248 39.051248 39.051248
20 52 16.401219 16.401219 16.401219
20 53 26.248809 26.248809 26.248809
20 54 45.650849 45.650849 45.650849
20 55 51.662365 51.662365 51.662365
20 56 51.419841 51.419841 51.419841
20 57 26.248809 26.248809 26.248809
20 58 19.209373 19.209373 19.209373
20 59 27.000000 27.000000 27.000000
20 60 33.526109 33.526109 33.526109
20 61 56.824291 56.824291 56.824291
20 62 55.081757 55.081757 55.081757
20 63 33.970576 33.970576 33.970576
20 64 25.079872 25.079872 25.079872
20 65 20.223748 20.223748 20.223748
20 66 30.805844 30.805844 30.805844
20 67 27.018512 27.018512 27.018512
20 68 38.832976 38.832976 38.832976
20 69 50.039984 50.039984 50.039984
20 70 43.416587 43.416587 43.416587
20 71 59.413803 59.413803 59.413803
20 72 50.537115 50.537115 50.537115
20 73 58.872744 58.872744 58.872744
20 74 64.031242 64.031242 64.031242
20 75 24.166092 24.166092 24.166092
20 76 37.336309 37.336309 37.336309
20 77 18.110770 18.110770 18.110770
20 78 20.248457 20.248457 20.248457
20 79 57.201399 57.201399 57.201399
20 80 68.264193 68.264193 68.264193
20 81 37.336309 37.336309 37.336309
20 82 48.507731 48.507731 48.507731
20 83 36.249138 36.249138 36.249138
20 84 21.587033 21.587033 21.587033
20 85 24.207437 24.207437 24.207437
20 86 24.698178 24.698178 24.698178
20 87 25.238859 25.238859 25.238859
20 88 33.105891 33.105891 33.105891
20 89 51.264022 51.264022 51.264022
20 90 25.495098 25.495098 25.495098
20 91 37.336309 37.336309 37.336309
20 92 32.756679 32.756679 32.756679
20 93 34.785054 34.785054 34.785054
20 94 46.097722 46.097722 46.097722
20 95 40.853396 40.853396 40.853396
20 96 30.413813 30.413813 30.413813
20 97 45.880279 45.880279 45.880279
20 98 38.832976 38.832976 38.832976
20 99 44.944410 44.944410 44.944410
20 100 29.681644 29.681644 29.681644
20 101 58.051701 58.051701 58.051701
21 1 35.057096 35.057096 35.057096
21 2 72.034714 72.034714 72.034714
21 3 63.245553 63.245553 63.245553
21 4 72.801099 72.801099 72.801099
21 5 68.622154 68.622154 68.622154
21 6 73.375745 73.375745 73.375745
21 7 64.621978 64.621978 64.621978
21 8 65.795137 65.795137 65.795137
21 9 70.384657 70.384657 70.384657
21 10 37.735925 37.735925 37.735925
21 11 40.607881 40.607881 40.607881
21 12 42.201896 42.201896 42.201896
21 13 45.343136 45.343136 45.343136
21 14 42.059482 42.059482 42.059482
21 15 47.634021 47.634021 47.634021
21 16 47.169906 47.169906 47.169906
21 17 48.877398 48.877398 48.877398
21 18 51.613952 51.613952 51.613952
21 19 10.198039 10.198039 10.198039
21 20 5.000000 5.000000 5.000000
21 22 10.198039 10.198039 10.198039
21 23 2.000000 2.000000 2.000000
21 24 10.770330 10.770330 10.770330
21 25 4.000000 4.000000 4.000000
21 26 12.206556 12.206556 12.206556
21 27 55.081757 55.081757 55.081757
21 28 56.648036 56.648036 56.648036

21 29 52.201533 52.201533 52.201533
21 30 52.000000 52.000000 52.000000
21 31 48.383882 48.383882 48.383882
21 32 50.159745 50.159745 50.159745
21 33 47.434165 47.434165 47.434165
21 34 44.147480 44.147480 44.147480
21 35 47.423623 47.423623 47.423623
21 36 74.330344 74.330344 74.330344
21 37 73.681748 73.681748 73.681748
21 38 70.837843 70.837843 70.837843
21 39 68.007353 68.007353 68.007353
21 40 67.446275 67.446275 67.446275
21 41 72.277244 72.277244 72.277244
21 42 62.096699 62.096699 62.096699
21 43 66.287254 66.287254 66.287254
21 44 71.196910 71.196910 71.196910
21 45 68.249542 68.249542 68.249542
21 46 70.519501 70.519501 70.519501
21 47 69.289249 69.289249 69.289249
21 48 50.000000 50.000000 50.000000
21 49 10.000000 10.000000 10.000000
21 50 3.000000 3.000000 3.000000
21 51 36.055513 36.055513 36.055513
21 52 13.928388 13.928388 13.928388
21 53 22.671568 22.671568 22.671568
21 54 41.340053 41.340053 41.340053
21 55 46.840154 46.840154 46.840154
21 56 46.572524 46.572524 46.572524
21 57 21.540659 21.540659 21.540659
21 58 15.620499 15.620499 15.620499
21 59 27.459060 27.459060 27.459060
21 60 32.388269 32.388269 32.388269
21 61 52.478567 52.478567 52.478567
21 62 50.089919 50.089919 50.089919
21 63 30.479501 30.479501 30.479501
21 64 23.537205 23.537205 23.537205
21 65 15.297059 15.297059 15.297059
21 66 25.961510 25.961510 25.961510
21 67 22.022716 22.022716 22.022716
21 68 34.828150 34.828150 34.828150
21 69 45.044423 45.044423 45.044423
21 70 38.600518 38.600518 38.600518
21 71 54.451814 54.451814 54.451814
21 72 46.141088 46.141088 46.141088
21 73 54.230987 54.230987 54.230987
21 74 60.207973 60.207973 60.207973
21 75 22.561028 22.561028 22.561028
21 76 38.327536 38.327536 38.327536
21 77 18.248288 18.248288 18.248288
21 78 22.472205 22.472205 22.472205
21 79 53.263496 53.263496 53.263496
21 80 64.070274 64.070274 64.070274
21 81 32.388269 32.388269 32.388269
21 82 43.566042 43.566042 43.566042
21 83 31.764760 31.764760 31.764760
21 84 16.763055 16.763055 16.763055
21 85 20.518285 20.518285 20.518285
21 86 22.472205 22.472205 22.472205
21 87 22.847319 22.847319 22.847319
21 88 31.320920 31.320920 31.320920
21 89 46.615448 46.615448 46.615448
21 90 26.925824 26.925824 26.925824
21 91 32.388269 32.388269 32.388269
21 92 27.892651 27.892651 27.892651
21 93 30.083218 30.083218 30.083218
21 94 41.593269 41.593269 41.593269
21 95 36.249138 36.249138 36.249138
21 96 26.076810 26.076810 26.076810
21 97 41.109610 41.109610 41.109610
21 98 38.118237 38.118237 38.118237
21 99 40.311289 40.311289 40.311289
21 100 25.612497 25.612497 25.612497
21 101 53.150729 53.150729 53.150729
22 1 45.000000 45.000000 45.000000
22 2 81.394103 81.394103 81.394103
22 3 72.277244 72.277244 72.277244
22 4 82.000000 82.000000 82.000000
22 5 77.620873 77.620873 77.620873
22 6 82.462113 82.462113 82.462113
22 7 73.375745 73.375745 73.375745
22 8 74.330344 74.330344 74.330344
22 9 79.056942 79.056942 79.056942
22 10 42.426407 42.426407 42.426407
22 11 46.097722 46.097722 46.097722
22 12 47.423623 47.423623 47.423623
22 13 51.224994 51.224994 51.224994
22 14 46.097722 46.097722 46.097722
22 15 53.150729 53.150729 53.150729
22 16 51.662365 51.662365 51.662365
22 17 53.150729 53.150729 53.150729
22 18 56.568542 56.568542 56.568542
22 19 4.000000 4.000000 4.000000
22 20 5.385165 5.385165 5.385165
22 21 10.198039 10.198039 10.198039
22 23 10.000000 10.000000 10.000000

22 24 2.000000 2.000000 2.000000
22 25 10.198039 10.198039 10.198039
22 26 5.000000 5.000000 5.000000
22 27 60.415230 60.415230 60.415230
22 28 62.649820 62.649820 62.649820
22 29 57.697487 57.697487 57.697487
22 30 58.309519 58.309519 58.309519
22 31 54.120237 54.120237 54.120237
22 32 56.603887 56.603887 56.603887
22 33 53.235327 53.235327 53.235327
22 34 49.244289 49.244289 49.244289
22 35 54.083269 54.083269 54.083269
22 36 84.433406 84.433406 84.433406
22 37 83.815273 83.815273 83.815273
22 38 80.956779 80.956779 80.956779
22 39 78.160092 78.160092 78.160092
22 40 77.620873 77.620873 77.620873
22 41 82.462113 82.462113 82.462113
22 42 72.277244 72.277244 72.277244
22 43 76.485293 76.485293 76.485293
22 44 81.394103 81.394103 81.394103
22 45 78.447435 78.447435 78.447435
22 46 79.555012 79.555012 79.555012
22 47 78.160092 78.160092 78.160092
22 48 55.172457 55.172457 55.172457
22 49 2.000000 2.000000 2.000000
22 50 7.280110 7.280110 7.280110
22 51 43.863424 43.863424 43.863424
22 52 21.213203 21.213203 21.213203
22 53 29.154759 29.154759 29.154759
22 54 49.244289 49.244289 49.244289
22 55 57.008771 57.008771 57.008771
22 56 55.901699 55.901699 55.901699
22 57 31.622777 31.622777 31.622777
22 58 22.360680 22.360680 22.360680
22 59 25.495098 25.495098 25.495098
22 60 33.541020 33.541020 33.541020
22 61 60.415230 60.415230 60.415230
22 62 60.207973 60.207973 60.207973
22 63 39.051248 39.051248 39.051248
22 64 29.154759 29.154759 29.154759
22 65 25.495098 25.495098 25.495098
22 66 35.355339 35.355339 35.355339
22 67 32.015621 32.015621 32.015621
22 68 44.102154 44.102154 44.102154
22 69 55.000000 55.000000 55.000000
22 70 47.853944 47.853944 47.853944
22 71 64.195015 64.195015 64.195015
22 72 55.901699 55.901699 55.901699
22 73 64.257295 64.257295 64.257295
22 74 66.850580 66.850580 66.850580
22 75 25.000000 25.000000 25.000000
22 76 35.000000 35.000000 35.000000
22 77 21.189620 21.189620 21.189620
22 78 17.117243 17.117243 17.117243
22 79 60.207973 60.207973 60.207973
22 80 71.589105 71.589105 71.589105
22 81 42.579338 42.579338 42.579338
22 82 53.758720 53.758720 53.758720
22 83 40.162171 40.162171 40.162171
22 84 26.172505 26.172505 26.172505
22 85 29.410882 29.410882 29.410882
22 86 29.206164 29.206164 29.206164
22 87 26.870058 26.870058 26.870058
22 88 33.837849 33.837849 33.837849
22 89 55.362442 55.362442 55.362442
22 90 27.000000 27.000000 27.000000
22 91 42.107007 42.107007 42.107007
22 92 38.078866 38.078866 38.078866
22 93 40.162171 40.162171 40.162171
22 94 51.478151 51.478151 51.478151
22 95 46.238512 46.238512 46.238512
22 96 35.777088 35.777088 35.777088
22 97 51.244512 51.244512 51.244512
22 98 38.275318 38.275318 38.275318
22 99 49.040799 49.040799 49.040799
22 100 33.105891 33.105891 33.105891
22 101 62.649820 62.649820 62.649820
23 1 35.000000 35.000000 35.000000
23 2 71.589105 71.589105 71.589105
23 3 62.641839 62.641839 62.641839
23 4 72.277244 72.277244 72.277244
23 5 68.007353 68.007353 68.007353
23 6 72.801099 72.801099 72.801099
23 7 63.906181 63.906181 63.906181
23 8 65.000000 65.000000 65.000000
23 9 69.641941 69.641941 69.641941
23 10 36.055513 36.055513 36.055513
23 11 39.051248 39.051248 39.051248
23 12 40.607881 40.607881 40.607881
23 13 43.863424 43.863424 43.863424
23 14 40.311289 40.311289 40.311289
23 15 46.097722 46.097722 46.097722
23 16 45.486262 45.486262 45.486262
23 17 47.169906 47.169906 47.169906

23 18 50.000000 50.000000 50.000000
23 19 10.770330 10.770330 10.770330
23 20 5.385165 5.385165 5.385165
23 21 2.000000 2.000000 2.000000
23 22 10.000000 10.000000 10.000000
23 24 10.198039 10.198039 10.198039
23 25 2.000000 2.000000 2.000000
23 26 11.180340 11.180340 11.180340
23 27 57.008771 57.008771 57.008771
23 28 58.523500 58.523500 58.523500
23 29 54.120237 54.120237 54.120237
23 30 53.851648 53.851648 53.851648
23 31 50.289164 50.289164 50.289164
23 32 52.000000 52.000000 52.000000
23 33 49.335586 49.335586 49.335586
23 34 46.097722 46.097722 46.097722
23 35 49.244289 49.244289 49.244289
23 36 75.026662 75.026662 75.026662
23 37 74.330344 74.330344 74.330344
23 38 71.512237 71.512237 71.512237
23 39 68.622154 68.622154 68.622154
23 40 68.007353 68.007353 68.007353
23 41 72.801099 72.801099 72.801099
23 42 62.641839 62.641839 62.641839
23 43 66.708320 66.708320 66.708320
23 44 71.589105 71.589105 71.589105
23 45 68.658576 68.658576 68.658576
23 46 69.921384 69.921384 69.921384
23 47 68.622154 68.622154 68.622154
23 48 48.414874 48.414874 48.414874
23 49 10.198039 10.198039 10.198039
23 50 3.605551 3.605551 3.605551
23 51 37.735925 37.735925 37.735925
23 52 15.811388 15.811388 15.811388
23 53 21.213203 21.213203 21.213203
23 54 40.311289 40.311289 40.311289
23 55 47.434165 47.434165 47.434165
23 56 46.097722 46.097722 46.097722
23 57 22.360680 22.360680 22.360680
23 58 14.142136 14.142136 14.142136
23 59 25.495098 25.495098 25.495098
23 60 30.413813 30.413813 30.413813
23 61 51.478151 51.478151 51.478151
23 62 50.249378 50.249378 50.249378
23 63 32.015621 32.015621 32.015621
23 64 25.495098 25.495098 25.495098
23 65 15.811388 15.811388 15.811388
23 66 25.495098 25.495098 25.495098
23 67 22.022716 22.022716 22.022716
23 68 36.124784 36.124784 36.124784
23 69 45.000000 45.000000 45.000000
23 70 38.078866 38.078866 38.078866
23 71 54.230987 54.230987 54.230987
23 72 47.169906 47.169906 47.169906
23 73 55.036352 55.036352 55.036352
23 74 58.898217 58.898217 58.898217
23 75 20.615528 20.615528 20.615528
23 76 36.400549 36.400549 36.400549
23 77 20.223748 20.223748 20.223748
23 78 20.808652 20.808652 20.808652
23 79 52.009614 52.009614 52.009614
23 80 62.968246 62.968246 62.968246
23 81 32.756679 32.756679 32.756679
23 82 43.931765 43.931765 43.931765
23 83 30.870698 30.870698 30.870698
23 84 16.278821 16.278821 16.278821
23 85 22.022716 22.022716 22.022716
23 86 24.351591 24.351591 24.351591
23 87 21.023796 21.023796 21.023796
23 88 29.410882 29.410882 29.410882
23 89 45.880279 45.880279 45.880279
23 90 28.792360 28.792360 28.792360
23 91 32.140317 32.140317 32.140317
23 92 28.460499 28.460499 28.460499
23 93 30.870698 30.870698 30.870698
23 94 42.544095 42.544095 42.544095
23 95 37.121422 37.121422 37.121422
23 96 27.202941 27.202941 27.202941
23 97 41.785165 41.785165 41.785165
23 98 36.124784 36.124784 36.124784
23 99 39.560081 39.560081 39.560081
23 100 24.413111 24.413111 24.413111
23 101 52.773099 52.773099 52.773099
24 1 45.044423 45.044423 45.044423
24 2 81.049368 81.049368 81.049368
24 3 71.805292 71.805292 71.805292
24 4 81.584312 81.584312 81.584312
24 5 77.129761 77.129761 77.129761
24 6 82.000000 82.000000 82.000000
24 7 72.801099 72.801099 72.801099
24 8 73.681748 73.681748 73.681748
24 9 78.447435 78.447435 78.447435
24 10 41.036569 41.036569 41.036569
24 11 44.821870 44.821870 44.821870
24 12 46.097722 46.097722 46.097722

24 13 50.000000 50.000000 50.000000
24 14 44.598206 44.598206 44.598206
24 15 51.855569 51.855569 51.855569
24 16 50.209561 50.209561 50.209561
24 17 51.662365 51.662365 51.662365
24 18 55.172457 55.172457 55.172457
24 19 6.000000 6.000000 6.000000
24 20 6.403124 6.403124 6.403124
24 21 10.770330 10.770330 10.770330
24 22 2.000000 2.000000 2.000000
24 23 10.198039 10.198039 10.198039
24 25 10.000000 10.000000 10.000000
24 26 3.000000 3.000000 3.000000
24 27 62.241465 62.241465 62.241465
24 28 64.412732 64.412732 64.412732
24 29 59.506302 59.506302 59.506302
24 30 60.033324 60.033324 60.033324
24 31 55.901699 55.901699 55.901699
24 32 58.309519 58.309519 58.309519
24 33 55.009090 55.009090 55.009090
24 34 51.078371 51.078371 51.078371
24 35 55.758407 55.758407 55.758407
24 36 85.094066 85.094066 85.094066
24 37 84.433406 84.433406 84.433406
24 38 81.596569 81.596569 81.596569
24 39 78.746428 78.746428 78.746428
24 40 78.160092 78.160092 78.160092
24 41 82.969874 82.969874 82.969874
24 42 72.801099 72.801099 72.801099
24 43 76.902536 76.902536 76.902536
24 44 81.786307 81.786307 81.786307
24 45 78.854296 78.854296 78.854296
24 46 79.075913 79.075913 79.075913
24 47 77.620873 77.620873 77.620873
24 48 53.814496 53.814496 53.814496
24 49 4.000000 4.000000 4.000000
24 50 8.062258 8.062258 8.062258
24 51 45.343136 45.343136 45.343136
24 52 22.671568 22.671568 22.671568
24 53 28.178006 28.178006 28.178006
24 54 48.466483 48.466483 48.466483
24 55 57.567352 57.567352 57.567352
24 56 55.578773 55.578773 55.578773
24 57 32.310989 32.310989 32.310989
24 58 21.540659 21.540659 21.540659
24 59 23.537205 23.537205 23.537205
24 60 31.764760 31.764760 31.764760
24 61 59.615434 59.615434 59.615434
24 62 60.406953 60.406953 60.406953
24 63 40.360872 40.360872 40.360872
24 64 30.886890 30.886890 30.886890
24 65 25.961510 25.961510 25.961510
24 66 35.128336 35.128336 35.128336
24 67 32.140317 32.140317 32.140317
24 68 45.221676 45.221676 45.221676
24 69 55.036352 55.036352 55.036352
24 70 47.518417 47.518417 47.518417
24 71 64.070274 64.070274 64.070274
24 72 56.824291 56.824291 56.824291
24 73 65.000000 65.000000 65.000000
24 74 65.734314 65.734314 65.734314
24 75 23.430749 23.430749 23.430749
24 76 33.000000 33.000000 33.000000
24 77 23.086793 23.086793 23.086793
24 78 15.132746 15.132746 15.132746
24 79 59.169249 59.169249 59.169249
24 80 70.661163 70.661163 70.661163
24 81 42.953463 42.953463 42.953463
24 82 54.129474 54.129474 54.129474
24 83 39.560081 39.560081 39.560081
24 84 26.019224 26.019224 26.019224
24 85 30.610456 30.610456 30.610456
24 86 30.805844 30.805844 30.805844
24 87 25.495098 25.495098 25.495098
24 88 32.202484 32.202484 32.202484
24 89 54.817880 54.817880 54.817880
24 90 29.000000 29.000000 29.000000
24 91 42.011903 42.011903 42.011903
24 92 38.600518 38.600518 38.600518
24 93 40.853396 40.853396 40.853396
24 94 52.325902 52.325902 52.325902
24 95 47.010637 47.010637 47.010637
24 96 36.715120 36.715120 36.715120
24 97 51.865210 51.865210 51.865210
24 98 36.400549 36.400549 36.400549
24 99 48.507731 48.507731 48.507731
24 100 32.310989 32.310989 32.310989
24 101 62.393910 62.393910 62.393910
25 1 35.057096 35.057096 35.057096
25 2 71.196910 71.196910 71.196910
25 3 62.096699 62.096699 62.096699
25 4 71.805292 71.805292 71.805292
25 5 67.446275 67.446275 67.446275
25 6 72.277244 72.277244 72.277244
25 7 63.245553 63.245553 63.245553

25 8 64.257295 64.257295 64.257295
25 9 68.949257 68.949257 68.949257
25 10 34.409301 34.409301 34.409301
25 11 37.536649 37.536649 37.536649
25 12 39.051248 39.051248 39.051248
25 13 42.426407 42.426407 42.426407
25 14 38.587563 38.587563 38.587563
25 15 44.598206 44.598206 44.598206
25 16 43.829214 43.829214 43.829214
25 17 45.486262 45.486262 45.486262
25 18 48.414874 48.414874 48.414874
25 19 11.661904 11.661904 11.661904
25 20 6.403124 6.403124 6.403124
25 21 4.000000 4.000000 4.000000
25 22 10.198039 10.198039 10.198039
25 23 2.000000 2.000000 2.000000
25 24 10.000000 10.000000 10.000000
25 26 10.440307 10.440307 10.440307
25 27 58.940648 58.940648 58.940648
25 28 60.406953 60.406953 60.406953
25 29 56.044625 56.044625 56.044625
25 30 55.713553 55.713553 55.713553
25 31 52.201533 52.201533 52.201533
25 32 53.851648 53.851648 53.851648
25 33 51.244512 51.244512 51.244512
25 34 48.052055 48.052055 48.052055
25 35 51.078371 51.078371 51.078371
25 36 75.769387 75.769387 75.769387
25 37 75.026662 75.026662 75.026662
25 38 72.235725 72.235725 72.235725
25 39 69.289249 69.289249 69.289249
25 40 68.622154 68.622154 68.622154
25 41 73.375745 73.375745 73.375745
25 42 63.245553 63.245553 63.245553
25 43 67.186308 67.186308 67.186308
25 44 72.034714 72.034714 72.034714
25 45 69.123079 69.123079 69.123079
25 46 69.375788 69.375788 69.375788
25 47 68.007353 68.007353 68.007353
25 48 46.861498 46.861498 46.861498
25 49 10.770330 10.770330 10.770330
25 50 5.000000 5.000000 5.000000
25 51 39.446166 39.446166 39.446166
25 52 17.720045 17.720045 17.720045
25 53 19.849433 19.849433 19.849433
25 54 39.357337 39.357337 39.357337
25 55 48.104054 48.104054 48.104054
25 56 45.705580 45.705580 45.705580
25 57 23.323808 23.323808 23.323808
25 58 12.806248 12.806248 12.806248
25 59 23.537205 23.537205 23.537205
25 60 28.442925 28.442925 28.442925
25 61 50.537115 50.537115 50.537115
25 62 50.487622 50.487622 50.487622
25 63 33.600595 33.600595 33.600595
25 64 27.459060 27.459060 27.459060
25 65 16.552945 16.552945 16.552945
25 66 25.179357 25.179357 25.179357
25 67 22.203603 22.203603 22.203603
25 68 37.483330 37.483330 37.483330
25 69 45.044423 45.044423 45.044423
25 70 37.656341 37.656341 37.656341
25 71 54.083269 54.083269 54.083269
25 72 48.259714 48.259714 48.259714
25 73 55.901699 55.901699 55.901699
25 74 57.628118 57.628118 57.628118
25 75 18.681542 18.681542 18.681542
25 76 34.481879 34.481879 34.481879
25 77 22.203603 22.203603 22.203603
25 78 19.209373 19.209373 19.209373
25 79 50.803543 50.803543 50.803543
25 80 61.911227 61.911227 61.911227
25 81 33.241540 33.241540 33.241540
25 82 44.384682 44.384682 44.384682
25 83 30.083218 30.083218 30.083218
25 84 16.031220 16.031220 16.031220
25 85 23.600847 23.600847 23.600847
25 86 26.248809 26.248809 26.248809
25 87 19.235384 19.235384 19.235384
25 88 27.513633 27.513633 27.513633
25 89 45.221676 45.221676 45.221676
25 90 30.675723 30.675723 30.675723
25 91 32.015621 32.015621 32.015621
25 92 29.154759 29.154759 29.154759
25 93 31.764760 31.764760 31.764760
25 94 43.566042 43.566042 43.566042
25 95 38.078866 38.078866 38.078866
25 96 28.425341 28.425341 28.425341
25 97 42.544095 42.544095 42.544095
25 98 34.132096 34.132096 34.132096
25 99 38.897301 38.897301 38.897301
25 100 23.323808 23.323808 23.323808
25 101 52.469038 52.469038 52.469038
26 1 45.276926 45.276926 45.276926
26 2 80.622577 80.622577 80.622577

26 3 71.196910 71.196910 71.196910
26 4 81.049368 81.049368 81.049368
26 5 76.485293 76.485293 76.485293
26 6 81.394103 81.394103 81.394103
26 7 72.034714 72.034714 72.034714
26 8 72.801099 72.801099 72.801099
26 9 77.620873 77.620873 77.620873
26 10 39.051248 39.051248 39.051248
26 11 43.011626 43.011626 43.011626
26 12 44.204072 44.204072 44.204072
26 13 48.259714 48.259714 48.259714
26 14 42.426407 42.426407 42.426407
26 15 50.000000 50.000000 50.000000
26 16 48.104054 48.104054 48.104054
26 17 49.497475 49.497475 49.497475
26 18 53.150729 53.150729 53.150729
26 19 9.000000 9.000000 9.000000
26 20 8.602325 8.602325 8.602325
26 21 12.206556 12.206556 12.206556
26 22 5.000000 5.000000 5.000000
26 23 11.180340 11.180340 11.180340
26 24 3.000000 3.000000 3.000000
26 25 10.440307 10.440307 10.440307
26 27 65.000000 65.000000 65.000000
26 28 67.082039 67.082039 67.082039
26 29 62.241465 62.241465 62.241465
26 30 62.649820 62.649820 62.649820
26 31 58.600341 58.600341 58.600341
26 32 60.901560 60.901560 60.901560
26 33 57.697487 57.697487 57.697487
26 34 53.851648 53.851648 53.851648
26 35 58.309519 58.309519 58.309519
26 36 86.162637 86.162637 86.162637
26 37 85.440037 85.440037 85.440037
26 38 82.637764 82.637764 82.637764
26 39 79.711982 79.711982 79.711982
26 40 79.056942 79.056942 79.056942
26 41 83.815273 83.815273 83.815273
26 42 73.681748 73.681748 73.681748
26 43 77.620873 77.620873 77.620873
26 44 82.462113 82.462113 82.462113
26 45 79.555012 79.555012 79.555012
26 46 78.447435 78.447435 78.447435
26 47 76.902536 76.902536 76.902536
26 48 51.855569 51.855569 51.855569
26 49 7.000000 7.000000 7.000000
26 50 9.899495 9.899495 9.899495
26 51 47.634021 47.634021 47.634021
26 52 25.000000 25.000000 25.000000
26 53 26.925824 26.925824 26.925824
26 54 47.434165 47.434165 47.434165
26 55 58.523500 58.523500 58.523500
26 56 55.226805 55.226805 55.226805
26 57 33.541020 33.541020 33.541020
26 58 20.615528 20.615528 20.615528
26 59 20.615528 20.615528 20.615528
26 60 29.154759 29.154759 29.154759
26 61 58.523500 58.523500 58.523500
26 62 60.827625 60.827625 60.827625
26 63 42.426407 42.426407 42.426407
26 64 33.541020 33.541020 33.541020
26 65 26.925824 26.925824 26.925824
26 66 35.000000 35.000000 35.000000
26 67 32.557641 32.557641 32.557641
26 68 47.010637 47.010637 47.010637
26 69 55.226805 55.226805 55.226805
26 70 47.169906 47.169906 47.169906
26 71 64.000000 64.000000 64.000000
26 72 58.309519 58.309519 58.309519
26 73 66.211781 66.211781 66.211781
26 74 64.140471 64.140471 64.140471
26 75 21.213203 21.213203 21.213203
26 76 30.000000 30.000000 30.000000
26 77 25.961510 25.961510 25.961510
26 78 12.165525 12.165525 12.165525
26 79 57.706152 57.706152 57.706152
26 80 69.354164 69.354164 69.354164
26 81 43.680659 43.680659 43.680659
26 82 54.817880 54.817880 54.817880
26 83 38.832976 38.832976 38.832976
26 84 26.076810 26.076810 26.076810
26 85 32.557641 32.557641 32.557641
26 86 33.286634 33.286634 33.286634
26 87 23.600847 23.600847 23.600847
26 88 29.832868 29.832868 29.832868
26 89 54.129474 54.129474 54.129474
26 90 32.000000 32.000000 32.000000
26 91 42.047592 42.047592 42.047592
26 92 39.560081 39.560081 39.560081
26 93 42.047592 42.047592 42.047592
26 94 53.712196 53.712196 53.712196
26 95 48.301139 48.301139 48.301139
26 96 38.275318 38.275318 38.275318
26 97 52.924474 52.924474 52.924474
26 98 33.615473 33.615473 33.615473

26 99 47.853944 47.853944 47.853944
26 100 31.320920 31.320920 31.320920
26 101 62.128898 62.128898 62.128898
27 1 58.523500 58.523500 58.523500
27 2 89.022469 89.022469 89.022469
27 3 85.755466 85.755466 85.755466
27 4 91.400219 91.400219 91.400219
27 5 90.138782 90.138782 90.138782
27 6 93.005376 93.005376 93.005376
27 7 89.185201 89.185201 89.185201
27 8 91.787799 91.787799 91.787799
27 9 94.339811 94.339811 94.339811
27 10 85.146932 85.146932 85.146932
27 11 85.586214 85.586214 85.586214
27 12 87.572827 87.572827 87.572827
27 13 88.283634 88.283634 88.283634
27 14 90.138782 90.138782 90.138782
27 15 91.241438 91.241438 91.241438
27 16 93.536089 93.536089 93.536089
27 17 95.524866 95.524866 95.524866
27 18 96.176920 96.176920 96.176920
27 19 56.797887 56.797887 56.797887
27 20 56.648036 56.648036 56.648036
27 21 55.081757 55.081757 55.081757
27 22 60.415230 60.415230 60.415230
27 23 57.008771 57.008771 57.008771
27 24 62.241465 62.241465 62.241465
27 25 58.940648 58.940648 58.940648
27 26 65.000000 65.000000 65.000000
27 28 5.000000 5.000000 5.000000
27 29 3.000000 3.000000 3.000000
27 30 7.071068 7.071068 7.071068
27 31 7.000000 7.000000 7.000000
27 32 8.602325 8.602325 8.602325
27 33 8.000000 8.000000 8.000000
27 34 11.180340 11.180340 11.180340
27 35 11.180340 11.180340 11.180340
27 36 61.717096 61.717096 61.717096
27 37 62.649820 62.649820 62.649820
27 38 60.033324 60.033324 60.033324
27 39 59.908263 59.908263 59.908263
27 40 61.032778 61.032778 61.032778
27 41 65.192024 65.192024 65.192024
27 42 58.258047 58.258047 58.258047
27 43 64.031242 64.031242 64.031242
27 44 68.007353 68.007353 68.007353
27 45 65.604878 65.604878 65.604878
27 46 91.263355 91.263355 91.263355
27 47 91.809586 91.809586 91.809586
27 48 94.201911 94.201911 94.201911
27 49 58.600341 58.600341 58.600341
27 50 55.973208 55.973208 55.973208
27 51 23.537205 23.537205 23.537205
27 52 41.231056 41.231056 41.231056
27 53 70.000000 70.000000 70.000000
27 54 77.620873 77.620873 77.620873
27 55 50.000000 50.000000 50.000000
27 56 71.589105 71.589105 71.589105
27 57 45.276926 45.276926 45.276926
27 58 65.192024 65.192024 65.192024
27 59 82.462113 82.462113 82.462113
27 60 85.586214 85.586214 85.586214
27 61 85.440037 85.440037 85.440037
27 62 61.032778 61.032778 61.032778
27 63 30.413813 30.413813 30.413813
27 64 31.622777 31.622777 31.622777
27 65 50.000000 50.000000 50.000000
27 66 60.827625 60.827625 60.827625
27 67 54.451814 54.451814 54.451814
27 68 33.241540 33.241540 33.241540
27 69 62.649820 62.649820 62.649820
27 70 67.675697 67.675697 67.675697
27 71 71.561163 71.561163 71.561163
27 72 39.051248 39.051248 39.051248
27 73 47.423623 47.423623 47.423623
27 74 97.718985 97.718985 97.718985
27 75 75.663730 75.663730 75.663730
27 76 93.407708 93.407708 93.407708
27 77 39.357337 39.357337 39.357337
27 78 76.896034 76.896034 76.896034
27 79 90.801982 90.801982 90.801982
27 80 96.772930 96.772930 96.772930
27 81 50.921508 50.921508 50.921508
27 82 53.851648 53.851648 53.851648
27 83 69.231496 69.231496 69.231496
27 84 58.008620 58.008620 58.008620
27 85 38.013156 38.013156 38.013156
27 86 32.756679 32.756679 32.756679
27 87 74.242845 74.242845 74.242845
27 88 83.216585 83.216585 83.216585
27 89 76.321688 76.321688 76.321688
27 90 37.536649 37.536649 37.536649
27 91 60.440053 60.440053 60.440053
27 92 47.539457 47.539457 47.539457
27 93 43.965896 43.965896 43.965896

27 94 40.496913 40.496913 40.496913
27 95 42.047592 42.047592 42.047592
27 96 39.623226 39.623226 39.623226
27 97 46.647615 46.647615 46.647615
27 98 91.787799 91.787799 91.787799
27 99 72.422372 72.422372 72.422372
27 100 69.180922 69.180922 69.180922
27 101 73.925638 73.925638 73.925638
28 1 57.008771 57.008771 57.008771
28 2 86.023253 86.023253 86.023253
28 3 83.240615 83.240615 83.240615
28 4 88.481637 88.481637 88.481637
28 5 87.464278 87.464278 87.464278
28 6 90.138782 90.138782 90.138782
28 7 86.769810 86.769810 86.769810
28 8 89.442719 89.442719 89.442719
28 9 91.787799 91.787799 91.787799
28 10 85.000000 85.000000 85.000000
28 11 85.146932 85.146932 85.146932
28 12 87.143560 87.143560 87.143560
28 13 87.572827 87.572827 87.572827
28 14 90.000000 90.000000 90.000000
28 15 90.553851 90.553851 90.553851
28 16 93.134312 93.134312 93.134312
28 17 95.131488 95.131488 95.131488
28 18 95.524866 95.524866 95.524866
28 19 59.169249 59.169249 59.169249
28 20 58.600341 58.600341 58.600341
28 21 56.648036 56.648036 56.648036
28 22 62.649820 62.649820 62.649820
28 23 58.523500 58.523500 58.523500
28 24 64.412732 64.412732 64.412732
28 25 60.406953 60.406953 60.406953
28 26 67.082039 67.082039 67.082039
28 27 5.000000 5.000000 5.000000
28 29 5.830952 5.830952 5.830952
28 30 5.000000 5.000000 5.000000
28 31 8.602325 8.602325 8.602325
28 32 7.000000 7.000000 7.000000
28 33 9.433981 9.433981 9.433981
28 34 14.142136 14.142136 14.142136
28 35 10.000000 10.000000 10.000000
28 36 57.306195 57.306195 57.306195
28 37 58.309519 58.309519 58.309519
28 38 55.758407 55.758407 55.758407
28 39 55.803226 55.803226 55.803226
28 40 57.008771 57.008771 57.008771
28 41 61.032778 61.032778 61.032778
28 42 54.488531 54.488531 54.488531
28 43 60.207973 60.207973 60.207973
28 44 64.031242 64.031242 64.031242
28 45 61.717096 61.717096 61.717096
28 46 88.509886 88.509886 88.509886
28 47 89.185201 89.185201 89.185201
28 48 93.536089 93.536089 93.536089
28 49 60.901560 60.901560 60.901560
28 50 57.775427 57.775427 57.775427
28 51 23.000000 23.000000 23.000000
28 52 42.720019 42.720019 42.720019
28 53 70.178344 70.178344 70.178344
28 54 76.485293 76.485293 76.485293
28 55 47.169906 47.169906 47.169906
28 56 69.641941 69.641941 69.641941
28 57 45.000000 45.000000 45.000000
28 58 65.764732 65.764732 65.764732
28 59 83.815273 83.815273 83.815273
28 60 86.313383 86.313383 86.313383
28 61 83.815273 83.815273 83.815273
28 62 58.309519 58.309519 58.309519
28 63 30.000000 30.000000 30.000000
28 64 33.541020 33.541020 33.541020
28 65 50.249378 50.249378 50.249378
28 66 60.207973 60.207973 60.207973
28 67 54.037024 54.037024 54.037024
28 68 31.780497 31.780497 31.780497
28 69 60.415230 60.415230 60.415230
28 70 66.219333 66.219333 66.219333
28 71 68.963759 68.963759 68.963759
28 72 36.055513 36.055513 36.055513
28 73 43.863424 43.863424 43.863424
28 74 96.301610 96.301610 96.301610
28 75 76.485293 76.485293 76.485293
28 76 94.868330 94.868330 94.868330
28 77 41.880783 41.880783 41.880783
28 78 78.790862 78.790862 78.790862
28 79 89.498603 89.498603 89.498603
28 80 94.921020 94.921020 94.921020
28 81 49.477268 49.477268 49.477268
28 82 51.429563 51.429563 51.429563
28 83 68.468971 68.468971 68.468971
28 84 58.137767 58.137767 58.137767
28 85 38.470768 38.470768 38.470768
28 86 34.176015 34.176015 34.176015
28 87 74.813100 74.813100 74.813100
28 88 83.725743 83.725743 83.725743

28 89 74.632433 74.632433 74.632433
28 90 41.036569 41.036569 41.036569
28 91 59.228372 59.228372 59.228372
28 92 46.529560 46.529560 46.529560
28 93 42.755117 42.755117 42.755117
28 94 38.013156 38.013156 38.013156
28 95 40.162171 40.162171 40.162171
28 96 39.051248 39.051248 39.051248
28 97 44.283180 44.283180 44.283180
28 98 92.574294 92.574294 92.574294
28 99 71.063352 71.063352 71.063352
28 100 69.000000 69.000000 69.000000
28 101 71.554175 71.554175 71.554175
29 1 55.713553 55.713553 55.713553
29 2 86.683332 86.683332 86.683332
29 3 83.216585 83.216585 83.216585
29 4 89.022469 89.022469 89.022469
29 5 87.658428 87.658428 87.658428
29 6 90.603532 90.603532 90.603532
29 7 86.608314 86.608314 86.608314
29 8 89.185201 89.185201 89.185201
29 9 91.809586 91.809586 91.809586
29 10 82.152298 82.152298 82.152298
29 11 82.607506 82.607506 82.607506
29 12 84.593144 84.593144 84.593144
29 13 85.328776 85.328776 85.328776
29 14 87.143560 87.143560 87.143560
29 15 88.283634 88.283634 88.283634
29 16 90.553851 90.553851 90.553851
29 17 92.541882 92.541882 92.541882
29 18 93.214806 93.214806 93.214806
29 19 54.120237 54.120237 54.120237
29 20 53.851648 53.851648 53.851648
29 21 52.201533 52.201533 52.201533
29 22 57.697487 57.697487 57.697487
29 23 54.120237 54.120237 54.120237
29 24 59.506302 59.506302 59.506302
29 25 56.044625 56.044625 56.044625
29 26 62.241465 62.241465 62.241465
29 27 3.000000 3.000000 3.000000
29 28 5.830952 5.830952 5.830952
29 30 5.385165 5.385165 5.385165
29 31 4.000000 4.000000 4.000000
29 32 6.403124 6.403124 6.403124
29 33 5.000000 5.000000 5.000000
29 34 8.602325 8.602325 8.602325
29 35 8.602325 8.602325 8.602325
29 36 60.415230 60.415230 60.415230
29 37 61.269895 61.269895 61.269895
29 38 58.591808 58.591808 58.591808
29 39 58.309519 58.309519 58.309519
29 40 59.363288 59.363288 59.363288
29 41 63.631753 63.631753 63.631753
29 42 56.400355 56.400355 56.400355
29 43 62.201286 62.201286 62.201286
29 44 66.287254 66.287254 66.287254
29 45 63.820060 63.820060 63.820060
29 46 88.814413 88.814413 88.814413
29 47 89.308454 89.308454 89.308454
29 48 91.241438 91.241438 91.241438
29 49 55.901699 55.901699 55.901699
29 50 53.141321 53.141321 53.141321
29 51 20.615528 20.615528 20.615528
29 52 38.327536 38.327536 38.327536
29 53 67.000000 67.000000 67.000000
29 54 74.726167 74.726167 74.726167
29 55 47.634021 47.634021 47.634021
29 56 68.876701 68.876701 68.876701
29 57 42.296572 42.296572 42.296572
29 58 62.201286 62.201286 62.201286
29 59 79.555012 79.555012 79.555012
29 60 82.607506 82.607506 82.607506
29 61 82.637764 82.637764 82.637764
29 62 58.600341 58.600341 58.600341
29 63 27.459060 27.459060 27.459060
29 64 28.792360 28.792360 28.792360
29 65 47.000000 47.000000 47.000000
29 66 57.870545 57.870545 57.870545
29 67 51.478151 51.478151 51.478151
29 68 30.463092 30.463092 30.463092
29 69 60.033324 60.033324 60.033324
29 70 64.845971 64.845971 64.845971
29 71 69.065187 69.065187 69.065187
29 72 36.796739 36.796739 36.796739
29 73 45.453273 45.453273 45.453273
29 74 94.868330 94.868330 94.868330
29 75 72.691127 72.691127 72.691127
29 76 90.520716 90.520716 90.520716
29 77 36.715120 36.715120 36.715120
29 78 74.094534 74.094534 74.094534
29 79 87.931792 87.931792 87.931792
29 80 94.021274 94.021274 94.021274
29 81 48.104054 48.104054 48.104054
29 82 51.312766 51.312766 51.312766
29 83 66.287254 66.287254 66.287254

29 84 55.009090 55.009090 55.009090
29 85 35.014283 35.014283 35.014283
29 86 29.832868 29.832868 29.832868
29 87 71.253070 71.253070 71.253070
29 88 80.224684 80.224684 80.224684
29 89 73.539105 73.539105 73.539105
29 90 35.355339 35.355339 35.355339
29 91 57.567352 57.567352 57.567352
29 92 44.643029 44.643029 44.643029
29 93 41.109610 41.109610 41.109610
29 94 38.013156 38.013156 38.013156
29 95 39.357337 39.357337 39.357337
29 96 36.674242 36.674242 36.674242
29 97 44.102154 44.102154 44.102154
29 98 88.814413 88.814413 88.814413
29 99 69.570109 69.570109 69.570109
29 100 66.189123 66.189123 66.189123
29 101 71.344236 71.344236 71.344236
30 1 52.201533 52.201533 52.201533
30 2 82.006097 82.006097 82.006097
30 3 78.892332 78.892332 78.892332
30 4 84.403791 84.403791 84.403791
30 5 83.216585 83.216585 83.216585
30 6 86.023253 86.023253 86.023253
30 7 82.365041 82.365041 82.365041
30 8 85.000000 85.000000 85.000000
30 9 87.464278 87.464278 87.464278
30 10 80.000000 80.000000 80.000000
30 11 80.156098 80.156098 80.156098
30 12 82.152298 82.152298 82.152298
30 13 82.607506 82.607506 82.607506
30 14 85.000000 85.000000 85.000000
30 15 85.586214 85.586214 85.586214
30 16 88.141931 88.141931 88.141931
30 17 90.138782 90.138782 90.138782
30 18 90.553851 90.553851 90.553851
30 19 54.918121 54.918121 54.918121
30 20 54.120237 54.120237 54.120237
30 21 52.000000 52.000000 52.000000
30 22 58.309519 58.309519 58.309519
30 23 53.851648 53.851648 53.851648
30 24 60.033324 60.033324 60.033324
30 25 55.713553 55.713553 55.713553
30 26 62.649820 62.649820 62.649820
30 27 7.071068 7.071068 7.071068
30 28 5.000000 5.000000 5.000000
30 29 5.385165 5.385165 5.385165
30 31 5.385165 5.385165 5.385165
30 32 2.000000 2.000000 2.000000
30 33 5.830952 5.830952 5.830952
30 34 11.180340 11.180340 11.180340
30 35 5.000000 5.000000 5.000000
30 36 55.036352 55.036352 55.036352
30 37 55.901699 55.901699 55.901699
30 38 53.235327 53.235327 53.235327
30 39 53.000000 53.000000 53.000000
30 40 54.083269 54.083269 54.083269
30 41 58.309519 58.309519 58.309519
30 42 51.224994 51.224994 51.224994
30 43 57.008771 57.008771 57.008771
30 44 61.032778 61.032778 61.032778
30 45 58.600341 58.600341 58.600341
30 46 84.314886 84.314886 84.314886
30 47 84.905830 84.905830 84.905830
30 48 88.566359 88.566359 88.566359
30 49 56.603887 56.603887 56.603887
30 50 53.225934 53.225934 53.225934
30 51 18.000000 18.000000 18.000000
30 52 38.078866 38.078866 38.078866
30 53 65.192024 65.192024 65.192024
30 54 71.589105 71.589105 71.589105
30 55 43.011626 43.011626 43.011626
30 56 65.000000 65.000000 65.000000
30 57 40.000000 40.000000 40.000000
30 58 60.827625 60.827625 60.827625
30 59 79.056942 79.056942 79.056942
30 60 81.394103 81.394103 81.394103
30 61 79.056942 79.056942 79.056942
30 62 54.083269 54.083269 54.083269
30 63 25.000000 25.000000 25.000000
30 64 29.154759 29.154759 29.154759
30 65 45.276926 45.276926 45.276926
30 66 55.226805 55.226805 55.226805
30 67 49.040799 49.040799 49.040799
30 68 26.925824 26.925824 26.925824
30 69 55.901699 55.901699 55.901699
30 70 61.400326 61.400326 61.400326
30 71 64.660653 64.660653 64.660653
30 72 32.015621 32.015621 32.015621
30 73 40.360872 40.360872 40.360872
30 74 91.482239 91.482239 91.482239
30 75 71.589105 71.589105 71.589105
30 76 90.138782 90.138782 90.138782
30 77 37.802116 37.802116 37.802116
30 78 74.249579 74.249579 74.249579

30 79 84.646323 84.646323 84.646323
30 80 90.249654 90.249654 90.249654
30 81 44.643029 44.643029 44.643029
30 82 47.010637 47.010637 47.010637
30 83 63.505905 63.505905 63.505905
30 84 53.150729 53.150729 53.150729
30 85 33.541020 33.541020 33.541020
30 86 29.546573 29.546573 29.546573
30 87 69.871310 69.871310 69.871310
30 88 78.771822 78.771822 78.771822
30 89 69.892775 69.892775 69.892775
30 90 37.802116 37.802116 37.802116
30 91 54.341513 54.341513 54.341513
30 92 41.593269 41.593269 41.593269
30 93 37.854986 37.854986 37.854986
30 94 33.615473 33.615473 33.615473
30 95 35.468296 35.468296 35.468296
30 96 34.058773 34.058773 34.058773
30 97 39.824616 39.824616 39.824616
30 98 87.664132 87.664132 87.664132
30 99 66.219333 66.219333 66.219333
30 100 64.000000 64.000000 64.000000
30 101 67.119297 67.119297 67.119297
31 1 52.000000 52.000000 52.000000
31 2 83.630138 83.630138 83.630138
31 3 79.881162 79.881162 79.881162
31 4 85.912746 85.912746 85.912746
31 5 84.403791 84.403791 84.403791
31 6 87.458562 87.458562 87.458562
31 7 83.216585 83.216585 83.216585
31 8 85.755466 85.755466 85.755466
31 9 88.481637 88.481637 88.481637
31 10 78.160092 78.160092 78.160092
31 11 78.638413 78.638413 78.638413
31 12 80.622577 80.622577 80.622577
31 13 81.394103 81.394103 81.394103
31 14 83.150466 83.150466 83.150466
31 15 84.344532 84.344532 84.344532
31 16 86.579443 86.579443 86.579443
31 17 88.566359 88.566359 88.566359
31 18 89.269256 89.269256 89.269256
31 19 50.606324 50.606324 50.606324
31 20 50.159745 50.159745 50.159745
31 21 48.383882 48.383882 48.383882
31 22 54.120237 54.120237 54.120237
31 23 50.289164 50.289164 50.289164
31 24 55.901699 55.901699 55.901699
31 25 52.201533 52.201533 52.201533
31 26 58.600341 58.600341 58.600341
31 27 7.000000 7.000000 7.000000
31 28 8.602325 8.602325 8.602325
31 29 4.000000 4.000000 4.000000
31 30 5.385165 5.385165 5.385165
31 32 5.000000 5.000000 5.000000
31 33 1.000000 1.000000 1.000000
31 34 5.830952 5.830952 5.830952
31 35 5.830952 5.830952 5.830952
31 36 58.872744 58.872744 58.872744
31 37 59.615434 59.615434 59.615434
31 38 56.859476 56.859476 56.859476
31 39 56.356011 56.356011 56.356011
31 40 57.306195 57.306195 57.306195
31 41 61.717096 61.717096 61.717096
31 42 54.083269 54.083269 54.083269
31 43 59.908263 59.908263 59.908263
31 44 64.140471 64.140471 64.140471
31 45 61.587336 61.587336 61.587336
31 46 85.603738 85.603738 85.603738
31 47 86.023253 86.023253 86.023253
31 48 87.298339 87.298339 87.298339
31 49 52.354560 52.354560 52.354560
31 50 49.396356 49.396356 49.396356
31 51 16.763055 16.763055 16.763055
31 52 34.481879 34.481879 34.481879
31 53 63.000000 63.000000 63.000000
31 54 70.880181 70.880181 70.880181
31 55 44.598206 44.598206 44.598206
31 56 65.299311 65.299311 65.299311
31 57 38.327536 38.327536 38.327536
31 58 58.215118 58.215118 58.215118
31 59 75.690158 75.690158 75.690158
31 60 78.638413 78.638413 78.638413
31 61 78.924014 78.924014 78.924014
31 62 55.443665 55.443665 55.443665
31 63 23.537205 23.537205 23.537205
31 64 25.079872 25.079872 25.079872
31 65 43.000000 43.000000 43.000000
31 66 53.935146 53.935146 53.935146
31 67 47.518417 47.518417 47.518417
31 68 26.832816 26.832816 26.832816
31 69 56.603887 56.603887 56.603887
31 70 61.098281 61.098281 61.098281
31 71 65.802736 65.802736 65.802736
31 72 33.970576 33.970576 33.970576
31 73 43.011626 43.011626 43.011626

31 74 91.082380 91.082380 91.082380
31 75 68.731361 68.731361 68.731361
31 76 86.683332 86.683332 86.683332
31 77 33.286634 33.286634 33.286634
31 78 70.384657 70.384657 70.384657
31 79 84.118963 84.118963 84.118963
31 80 90.376988 90.376988 90.376988
31 81 44.384682 44.384682 44.384682
31 82 48.010416 48.010416 48.010416
31 83 62.369865 62.369865 62.369865
31 84 51.009803 51.009803 51.009803
31 85 31.016125 31.016125 31.016125
31 86 25.961510 25.961510 25.961510
31 87 67.268120 67.268120 67.268120
31 88 76.236474 76.236474 76.236474
31 89 69.856997 69.856997 69.856997
31 90 32.649655 32.649655 32.649655
31 91 53.758720 53.758720 53.758720
31 92 40.804412 40.804412 40.804412
31 93 37.336309 37.336309 37.336309
31 94 34.828150 34.828150 34.828150
31 95 35.846897 35.846897 35.846897
31 96 32.756679 32.756679 32.756679
31 97 40.804412 40.804412 40.804412
31 98 84.852814 84.852814 84.852814
31 99 65.787537 65.787537 65.787537
31 100 62.201286 62.201286 62.201286
31 101 67.955868 67.955868 67.955868
32 1 50.289164 50.289164 50.289164
32 2 80.430094 80.430094 80.430094
32 3 77.175126 77.175126 77.175126
32 4 82.800966 82.800966 82.800966
32 5 81.541401 81.541401 81.541401
32 6 84.403791 84.403791 84.403791
32 7 80.622577 80.622577 80.622577
32 8 83.240615 83.240615 83.240615
32 9 85.755466 85.755466 85.755466
32 10 78.000000 78.000000 78.000000
32 11 78.160092 78.160092 78.160092
32 12 80.156098 80.156098 80.156098
32 13 80.622577 80.622577 80.622577
32 14 83.000000 83.000000 83.000000
32 15 83.600239 83.600239 83.600239
32 16 86.145226 86.145226 86.145226
32 17 88.141931 88.141931 88.141931
32 18 88.566359 88.566359 88.566359
32 19 53.254108 53.254108 53.254108
32 20 52.354560 52.354560 52.354560
32 21 50.159745 50.159745 50.159745
32 22 56.603887 56.603887 56.603887
32 23 52.000000 52.000000 52.000000
32 24 58.309519 58.309519 58.309519
32 25 53.851648 53.851648 53.851648
32 26 60.901560 60.901560 60.901560
32 27 8.602325 8.602325 8.602325
32 28 7.000000 7.000000 7.000000
32 29 6.403124 6.403124 6.403124
32 30 2.000000 2.000000 2.000000
32 31 5.000000 5.000000 5.000000
32 33 5.099020 5.099020 5.099020
32 34 10.440307 10.440307 10.440307
32 35 3.000000 3.000000 3.000000
32 36 54.230987 54.230987 54.230987
32 37 55.036352 55.036352 55.036352
32 38 52.325902 52.325902 52.325902
32 39 51.971146 51.971146 51.971146
32 40 53.000000 53.000000 53.000000
32 41 57.306195 57.306195 57.306195
32 42 50.000000 50.000000 50.000000
32 43 55.803226 55.803226 55.803226
32 44 59.908263 59.908263 59.908263
32 45 57.428216 57.428216 57.428216
32 46 82.661962 82.661962 82.661962
32 47 83.216585 83.216585 83.216585
32 48 86.579443 86.579443 86.579443
32 49 54.918121 54.918121 54.918121
32 50 51.429563 51.429563 51.429563
32 51 16.000000 16.000000 16.000000
32 52 36.249138 36.249138 36.249138
32 53 63.198101 63.198101 63.198101
32 54 69.634761 69.634761 69.634761
32 55 41.400483 41.400483 41.400483
32 56 63.158531 63.158531 63.158531
32 57 38.000000 38.000000 38.000000
32 58 58.855756 58.855756 58.855756
32 59 77.162167 77.162167 77.162167
32 60 79.429214 79.429214 79.429214
32 61 77.162167 77.162167 77.162167
32 62 52.430907 52.430907 52.430907
32 63 23.000000 23.000000 23.000000
32 64 27.459060 27.459060 27.459060
32 65 43.289722 43.289722 43.289722
32 66 53.235327 53.235327 53.235327
32 67 47.042534 47.042534 47.042534
32 68 25.000000 25.000000 25.000000
32 69 54.129237 54.129237 54.129237

32 69 54.120237 54.120237 54.120237
32 70 59.481089 59.481089 59.481089
32 71 62.968246 62.968246 62.968246
32 72 30.479501 30.479501 30.479501
32 73 39.051248 39.051248 39.051248
32 74 89.560036 89.560036 89.560036
32 75 69.634761 69.634761 69.634761
32 76 88.255311 88.255311 88.255311
32 77 36.235342 36.235342 36.235342
32 78 72.449983 72.449983 72.449983
32 79 82.710338 82.710338 82.710338
32 80 88.391176 88.391176 88.391176
32 81 42.720019 42.720019 42.720019
32 82 45.276926 45.276926 45.276926
32 83 61.522354 61.522354 61.522354
32 84 51.156622 51.156622 51.156622
32 85 31.575307 31.575307 31.575307
32 86 27.730849 27.730849 27.730849
32 87 67.896981 67.896981 67.896981
32 88 76.791927 76.791927 76.791927
32 89 68.007353 68.007353 68.007353
32 90 36.619667 36.619667 36.619667
32 91 52.392748 52.392748 52.392748
32 92 39.623226 39.623226 39.623226
32 93 35.902646 35.902646 35.902646
32 94 31.906112 31.906112 31.906112
32 95 33.615473 33.615473 33.615473
32 96 32.062439 32.062439 32.062439
32 97 38.078866 38.078866 38.078866
32 98 85.702975 85.702975 85.702975
32 99 64.288413 64.288413 64.288413
32 100 62.000000 62.000000 62.000000
32 101 65.368188 65.368188 65.368188
33 1 51.078371 51.078371 51.078371
33 2 82.879430 82.879430 82.879430
33 3 79.056942 79.056942 79.056942
33 4 85.146932 85.146932 85.146932
33 5 83.600239 83.600239 83.600239
33 6 86.683332 86.683332 86.683332
33 7 82.377181 82.377181 82.377181
33 8 84.905830 84.905830 84.905830
33 9 87.658428 87.658428 87.658428
33 10 77.162167 77.162167 77.162167
33 11 77.646635 77.646635 77.646635
33 12 79.630396 79.630396 79.630396
33 13 80.411442 80.411442 80.411442
33 14 82.152298 82.152298 82.152298
33 15 83.360662 83.360662 83.360662
33 16 85.586214 85.586214 85.586214
33 17 87.572827 87.572827 87.572827
33 18 88.283634 88.283634 88.283634
33 19 49.739320 49.739320 49.739320
33 20 49.244289 49.244289 49.244289
33 21 47.434165 47.434165 47.434165
33 22 53.235327 53.235327 53.235327
33 23 49.335586 49.335586 49.335586
33 24 55.009090 55.009090 55.009090
33 25 51.244512 51.244512 51.244512
33 26 57.697487 57.697487 57.697487
33 27 8.000000 8.000000 8.000000
33 28 9.433981 9.433981 9.433981
33 29 5.000000 5.000000 5.000000
33 30 5.830952 5.830952 5.830952
33 31 1.000000 1.000000 1.000000
33 32 5.099020 5.099020 5.099020
33 34 5.385165 5.385165 5.385165
33 35 5.385165 5.385165 5.385165
33 36 58.523500 58.523500 58.523500
33 37 59.236813 59.236813 59.236813
33 38 56.462377 56.462377 56.462377
33 39 55.901699 55.901699 55.901699
33 40 56.824291 56.824291 56.824291
33 41 61.269895 61.269895 61.269895
33 42 53.535035 53.535035 53.535035
33 43 59.363288 59.363288 59.363288
33 44 63.631753 63.631753 63.631753
33 45 61.057350 61.057350 61.057350
33 46 84.811556 84.811556 84.811556
33 47 85.211502 85.211502 85.211502
33 48 86.313383 86.313383 86.313383
33 49 51.478151 51.478151 51.478151
33 50 48.466483 48.466483 48.466483
33 51 15.811388 15.811388 15.811388
33 52 33.526109 33.526109 33.526109
33 53 62.000000 62.000000 62.000000
33 54 69.921384 69.921384 69.921384
33 55 43.863424 43.863424 43.863424
33 56 64.412732 64.412732 64.412732
33 57 37.336309 37.336309 37.336309
33 58 57.218878 57.218878 57.218878
33 59 74.726167 74.726167 74.726167
33 60 77.646635 77.646635 77.646635
33 61 78.000000 78.000000 78.000000
33 62 54.671748 54.671748 54.671748
33 63 22.561028 22.561028 22.561028
33 64 24.166092 24.166092 24.166092

33 64 24.186092 24.186092 24.186092
33 65 42.000000 42.000000 42.000000
33 66 52.952809 52.952809 52.952809
33 67 46.529560 46.529560 46.529560
33 68 25.942244 25.942244 25.942244
33 69 55.758407 55.758407 55.758407
33 70 60.166436 60.166436 60.166436
33 71 65.000000 65.000000 65.000000
33 72 33.301652 33.301652 33.301652
33 73 42.438190 42.438190 42.438190
33 74 90.138782 90.138782 90.138782
33 75 67.742158 67.742158 67.742158
33 76 85.726309 85.726309 85.726309
33 77 32.449961 32.449961 32.449961
33 78 69.462220 69.462220 69.462220
33 79 83.168504 83.168504 83.168504
33 80 89.470666 89.470666 89.470666
33 81 43.462628 43.462628 43.462628
33 82 47.201695 47.201695 47.201695
33 83 61.392182 61.392182 61.392182
33 84 50.009999 50.009999 50.009999
33 85 30.016662 30.016662 30.016662
33 86 25.000000 25.000000 25.000000
33 87 66.272166 66.272166 66.272166
33 88 75.239617 75.239617 75.239617
33 89 68.942005 68.942005 68.942005
33 90 32.015621 32.015621 32.015621
33 91 52.810984 52.810984 52.810984
33 92 39.849718 39.849718 39.849718
33 93 36.400549 36.400549 36.400549
33 94 34.058773 34.058773 34.058773
33 95 34.985711 34.985711 34.985711
33 96 31.780497 31.780497 31.780497
33 97 40.000000 40.000000 40.000000
33 98 83.862983 83.862983 83.862983
33 99 64.845971 64.845971 64.845971
33 100 61.204575 61.204575 61.204575
33 101 67.119297 67.119297 67.119297
34 1 51.478151 51.478151 51.478151
34 2 84.852814 84.852814 84.852814
34 3 80.430094 80.430094 80.430094
34 4 87.000000 87.000000 87.000000
34 5 85.146932 85.146932 85.146932
34 6 88.459030 88.459030 88.459030
34 7 83.600239 83.600239 83.600239
34 8 86.023253 86.023253 86.023253
34 9 89.022469 89.022469 89.022469
34 10 75.663730 75.663730 75.663730
34 11 76.485293 76.485293 76.485293
34 12 78.447435 78.447435 78.447435
34 13 79.555012 79.555012 79.555012
34 14 80.622577 80.622577 80.622577
34 15 82.462113 82.462113 82.462113
34 16 84.344532 84.344532 84.344532
34 17 86.313383 86.313383 86.313383
34 18 87.321246 87.321246 87.321246
34 19 45.617979 45.617979 45.617979
34 20 45.541190 45.541190 45.541190
34 21 44.147480 44.147480 44.147480
34 22 49.244289 49.244289 49.244289
34 23 46.097722 46.097722 46.097722
34 24 51.078371 51.078371 51.078371
34 25 48.052055 48.052055 48.052055
34 26 53.851648 53.851648 53.851648
34 27 11.180340 11.180340 11.180340
34 28 14.142136 14.142136 14.142136
34 29 8.602325 8.602325 8.602325
34 30 11.180340 11.180340 11.180340
34 31 5.830952 5.830952 5.830952
34 32 10.440307 10.440307 10.440307
34 33 5.385165 5.385165 5.385165
34 35 10.000000 10.000000 10.000000
34 36 62.641839 62.641839 62.641839
34 37 63.245553 63.245553 63.245553
34 38 60.406953 60.406953 60.406953
34 39 59.615434 59.615434 59.615434
34 40 60.415230 60.415230 60.415230
34 41 65.000000 65.000000 65.000000
34 42 56.824291 56.824291 56.824291
34 43 62.649820 62.649820 62.649820
34 44 67.082039 67.082039 67.082039
34 45 64.412732 64.412732 64.412732
34 46 86.452299 86.452299 86.452299
34 47 86.683332 86.683332 86.683332
34 48 85.375641 85.375641 85.375641
34 49 47.423623 47.423623 47.423623
34 50 44.922155 44.922155 44.922155
34 51 16.401219 16.401219 16.401219
34 52 30.413813 30.413813 30.413813
34 53 60.207973 60.207973 60.207973
34 54 69.641941 69.641941 69.641941
34 55 46.097722 46.097722 46.097722
34 56 65.192024 65.192024 65.192024
34 57 36.400549 36.400549 36.400549
34 58 55.000000 55.000000 55.000000
34 59 71.589105 71.589105 71.589105

34 60 75.166482 75.166482 75.166482
34 61 78.262379 78.262379 78.262379
34 62 56.568542 56.568542 56.568542
34 63 22.360680 22.360680 22.360680
34 64 20.615528 20.615528 20.615528
34 65 40.311289 40.311289 40.311289
34 66 52.201533 52.201533 52.201533
34 67 45.607017 45.607017 45.607017
34 68 27.018512 27.018512 27.018512
34 69 57.008771 57.008771 57.008771
34 70 60.373835 60.373835 60.373835
34 71 66.603303 66.603303 66.603303
34 72 36.055513 36.055513 36.055513
34 73 45.650849 45.650849 45.650849
34 74 90.077744 90.077744 90.077744
34 75 65.192024 65.192024 65.192024
34 76 82.462113 82.462113 82.462113
34 77 28.178006 28.178006 28.178006
34 78 65.787537 65.787537 65.787537
34 79 83.006024 83.006024 83.006024
34 80 89.944427 89.944427 89.944427
34 81 43.908997 43.908997 43.908997
34 82 48.836462 48.836462 48.836462
34 83 60.728906 60.728906 60.728906
34 84 48.373546 48.373546 48.373546
34 85 28.284271 28.284271 28.284271
34 86 22.090722 22.090722 22.090722
34 87 64.007812 64.007812 64.007812
34 88 73.006849 73.006849 73.006849
34 89 69.354164 69.354164 69.354164
34 90 26.907248 26.907248 26.907248
34 91 52.801515 52.801515 52.801515
34 92 39.812058 39.812058 39.812058
34 93 36.715120 36.715120 36.715120
34 94 36.124784 36.124784 36.124784
34 95 36.235342 36.235342 36.235342
34 96 31.384710 31.384710 31.384710
34 97 41.725292 41.725292 41.725292
34 98 81.301906 81.301906 81.301906
34 99 64.884513 64.884513 64.884513
34 100 59.841457 59.841457 59.841457
34 101 68.410526 68.410526 68.410526
35 1 47.434165 47.434165 47.434165
35 2 78.102497 78.102497 78.102497
35 3 74.625733 74.625733 74.625733
35 4 80.430094 80.430094 80.430094
35 5 79.056942 79.056942 79.056942
35 6 82.006097 82.006097 82.006097
35 7 78.032045 78.032045 78.032045
35 8 80.622577 80.622577 80.622577
35 9 83.216585 83.216585 83.216585
35 10 75.000000 75.000000 75.000000
35 11 75.166482 75.166482 75.166482
35 12 77.162167 77.162167 77.162167
35 13 77.646635 77.646635 77.646635
35 14 80.000000 80.000000 80.000000
35 15 80.622577 80.622577 80.622577
35 16 83.150466 83.150466 83.150466
35 17 85.146932 85.146932 85.146932
35 18 85.586214 85.586214 85.586214
35 19 50.803543 50.803543 50.803543
35 20 49.739320 49.739320 49.739320
35 21 47.423623 47.423623 47.423623
35 22 54.083269 54.083269 54.083269
35 23 49.244289 49.244289 49.244289
35 24 55.758407 55.758407 55.758407
35 25 51.078371 51.078371 51.078371
35 26 58.309519 58.309519 58.309519
35 27 11.180340 11.180340 11.180340
35 28 10.000000 10.000000 10.000000
35 29 8.602325 8.602325 8.602325
35 30 5.000000 5.000000 5.000000
35 31 5.830952 5.830952 5.830952
35 32 3.000000 3.000000 3.000000
35 33 5.385165 5.385165 5.385165
35 34 10.000000 10.000000 10.000000
35 36 53.141321 53.141321 53.141321
35 37 53.851648 53.851648 53.851648
35 38 51.078371 51.078371 51.078371
35 39 50.537115 50.537115 50.537115
35 40 51.478151 51.478151 51.478151
35 41 55.901699 55.901699 55.901699
35 42 48.259714 48.259714 48.259714
35 43 54.083269 54.083269 54.083269
35 44 58.309519 58.309519 58.309519
35 45 55.758407 55.758407 55.758407
35 46 80.212219 80.212219 80.212219
35 47 80.709355 80.709355 80.709355
35 48 83.600239 83.600239 83.600239
35 49 52.430907 52.430907 52.430907
35 50 48.764741 48.764741 48.764741
35 51 13.000000 13.000000 13.000000
35 52 33.541020 33.541020 33.541020
35 53 60.207973 60.207973 60.207973
35 54 66.708320 66.708320 66.708320

35 55 39.051248 39.051248 39.051248
35 56 60.415230 60.415230 60.415230
35 57 35.000000 35.000000 35.000000
35 58 55.901699 55.901699 55.901699
35 59 74.330344 74.330344 74.330344
35 60 76.485293 76.485293 76.485293
35 61 74.330344 74.330344 74.330344
35 62 50.000000 50.000000 50.000000
35 63 20.000000 20.000000 20.000000
35 64 25.000000 25.000000 25.000000
35 65 40.311289 40.311289 40.311289
35 66 50.249378 50.249378 50.249378
35 67 44.045431 44.045431 44.045431
35 68 22.135944 22.135944 22.135944
35 69 51.478151 51.478151 51.478151
35 70 56.612719 56.612719 56.612719
35 71 60.464866 60.464866 60.464866
35 72 28.284271 28.284271 28.284271
35 73 37.202150 37.202150 37.202150
35 74 86.683332 86.683332 86.683332
35 75 66.708320 66.708320 66.708320
35 76 85.440037 85.440037 85.440037
35 77 33.970576 33.970576 33.970576
35 78 69.771054 69.771054 69.771054
35 79 79.812280 79.812280 79.812280
35 80 85.615419 85.615419 85.615419
35 81 39.849718 39.849718 39.849718
35 82 42.720019 42.720019 42.720019
35 83 58.549125 58.549125 58.549125
35 84 48.166378 48.166378 48.166378
35 85 28.635642 28.635642 28.635642
35 86 25.059928 25.059928 25.059928
35 87 64.938432 64.938432 64.938432
35 88 73.824115 73.824115 73.824115
35 89 65.192024 65.192024 65.192024
35 90 34.985711 34.985711 34.985711
35 91 49.477268 49.477268 49.477268
35 92 36.674242 36.674242 36.674242
35 93 32.984845 32.984845 32.984845
35 94 29.410882 29.410882 29.410882
35 95 30.870698 30.870698 30.870698
35 96 29.068884 29.068884 29.068884
35 97 35.510562 35.510562 35.510562
35 98 82.764727 82.764727 82.764727
35 99 61.400326 61.400326 61.400326
35 100 59.000000 59.000000 59.000000
35 101 62.769419 62.769419 62.769419
36 1 44.204072 44.204072 44.204072
36 2 42.000000 42.000000 42.000000
36 3 46.097722 46.097722 46.097722
36 4 45.000000 45.000000 45.000000
36 5 47.265209 47.265209 47.265209
36 6 47.000000 47.000000 47.000000
36 7 50.009999 50.009999 50.009999
36 8 52.952809 52.952809 52.952809
36 9 52.239832 52.239832 52.239832
36 10 75.822160 75.822160 75.822160
36 11 72.622311 72.622311 72.622311
36 12 74.202426 74.202426 74.202426
36 13 71.281134 71.281134 71.281134
36 14 79.649231 79.649231 79.649231
36 15 73.783467 73.783467 73.783467
36 16 79.056942 79.056942 79.056942
36 17 80.709355 80.709355 80.709355
36 18 78.032045 78.032045 78.032045
36 19 83.240615 83.240615 83.240615
36 20 79.056942 79.056942 79.056942
36 21 74.330344 74.330344 74.330344
36 22 84.433406 84.433406 84.433406
36 23 75.026662 75.026662 75.026662
36 24 85.094066 85.094066 85.094066
36 25 75.769387 75.769387 75.769387
36 26 86.162637 86.162637 86.162637
36 27 61.717096 61.717096 61.717096
36 28 57.306195 57.306195 57.306195
36 29 60.415230 60.415230 60.415230
36 30 55.036352 55.036352 55.036352
36 31 58.872744 58.872744 58.872744
36 32 54.230987 54.230987 54.230987
36 33 58.523500 58.523500 58.523500
36 34 62.641839 62.641839 62.641839
36 35 53.141321 53.141321 53.141321
36 37 2.000000 2.000000 2.000000
36 38 3.605551 3.605551 3.605551
36 39 7.071068 7.071068 7.071068
36 40 8.602325 8.602325 8.602325
36 41 7.000000 7.000000 7.000000
36 42 13.453624 13.453624 13.453624
36 43 13.000000 13.000000 13.000000
36 44 12.000000 12.000000 12.000000
36 45 12.369317 12.369317 12.369317
36 46 47.095647 47.095647 47.095647
36 47 49.254441 49.254441 49.254441
36 48 76.321688 76.321688 76.321688
36 49 83.815273 83.815273 83.815273

36 50 77.162167 77.162167 77.162167
36 51 50.249378 50.249378 50.249378
36 52 66.098411 66.098411 66.098411
36 53 69.202601 69.202601 69.202601
36 54 58.600341 58.600341 58.600341
36 55 27.730849 27.730849 27.730849
36 56 44.654227 44.654227 44.654227
36 57 52.810984 52.810984 52.810984
36 58 70.491134 70.491134 70.491134
36 59 91.263355 91.263355 91.263355
36 60 86.452299 86.452299 86.452299
36 61 57.697487 57.697487 57.697487
36 62 29.732137 29.732137 29.732137
36 63 50.039984 50.039984 50.039984
36 64 65.030762 65.030762 65.030762
36 65 59.236813 59.236813 59.236813
36 66 55.217751 55.217751 55.217751
36 67 54.589376 54.589376 54.589376
36 68 43.104524 43.104524 43.104524
36 69 36.796739 36.796739 36.796739
36 70 48.836462 48.836462 48.836462
36 71 35.777088 35.777088 35.777088
36 72 30.066593 30.066593 30.066593
36 73 20.396078 20.396078 20.396078
36 74 69.641941 69.641941 69.641941
36 75 80.212219 80.212219 80.212219
36 76 101.212647 101.212647 101.212647
36 77 73.334848 73.334848 73.334848
36 78 93.059121 93.059121 93.059121
36 79 65.741920 65.741920 65.741920
36 80 63.324561 63.324561 63.324561
36 81 42.941821 42.941821 42.941821
36 82 32.449961 32.449961 32.449961
36 83 58.000000 58.000000 58.000000
36 84 61.773781 61.773781 61.773781
36 85 56.885851 56.885851 56.885851
36 86 62.128898 62.128898 62.128898
36 87 76.400262 76.400262 76.400262
36 88 82.134037 82.134037 82.134037
36 89 50.774009 50.774009 50.774009
36 90 80.000000 80.000000 80.000000
36 91 48.414874 48.414874 48.414874
36 92 46.615448 46.615448 46.615448
36 93 44.271887 44.271887 44.271887
36 94 33.541020 33.541020 33.541020
36 95 38.327536 38.327536 38.327536
36 96 49.244289 49.244289 49.244289
36 97 33.241540 33.241540 33.241540
36 98 91.967386 91.967386 91.967386
36 99 52.630789 52.630789 52.630789
36 100 64.660653 64.660653 64.660653
36 101 40.249224 40.249224 40.249224
37 1 43.011626 43.011626 43.011626
37 2 40.000000 40.000000 40.000000
37 3 44.147480 44.147480 44.147480
37 4 43.000000 43.000000 43.000000
37 5 45.276926 45.276926 45.276926
37 6 45.000000 45.000000 45.000000
37 7 48.052055 48.052055 48.052055
37 8 50.990195 50.990195 50.990195
37 9 50.249378 50.249378 50.249378
37 10 74.330344 74.330344 74.330344
37 11 71.063352 71.063352 71.063352
37 12 72.622311 72.622311 72.622311
37 13 69.634761 69.634761 69.634761
37 14 78.102497 78.102497 78.102497
37 15 72.111026 72.111026 72.111026
37 16 77.420927 77.420927 77.420927
37 17 79.056942 79.056942 79.056942
37 18 76.321688 76.321688 76.321688
37 19 82.710338 82.710338 82.710338
37 20 78.447435 78.447435 78.447435
37 21 73.681748 73.681748 73.681748
37 22 83.815273 83.815273 83.815273
37 23 74.330344 74.330344 74.330344
37 24 84.433406 84.433406 84.433406
37 25 75.026662 75.026662 75.026662
37 26 85.440037 85.440037 85.440037
37 27 62.649820 62.649820 62.649820
37 28 58.309519 58.309519 58.309519
37 29 61.269895 61.269895 61.269895
37 30 55.901699 55.901699 55.901699
37 31 59.615434 59.615434 59.615434
37 32 55.036352 55.036352 55.036352
37 33 59.236813 59.236813 59.236813
37 34 63.245553 63.245553 63.245553
37 35 53.851648 53.851648 53.851648
37 36 2.000000 2.000000 2.000000
37 38 3.000000 3.000000 3.000000
37 39 5.830952 5.830952 5.830952
37 40 7.071068 7.071068 7.071068
37 41 5.000000 5.000000 5.000000
37 42 12.206556 12.206556 12.206556
37 43 11.180340 11.180340 11.180340
37 44 10.000000 10.000000 10.000000

37 45 10.440307 10.440307 10.440307
37 46 45.099889 45.099889 45.099889
37 47 47.265209 47.265209 47.265209
37 48 74.625733 74.625733 74.625733
37 49 83.240615 83.240615 83.240615
37 50 76.537572 76.537572 76.537572
37 51 50.487622 50.487622 50.487622
37 52 65.764732 65.764732 65.764732
37 53 68.007353 68.007353 68.007353
37 54 57.008771 57.008771 57.008771
37 55 26.925824 26.925824 26.925824
37 56 43.011626 43.011626 43.011626
37 57 52.201533 52.201533 52.201533
37 58 69.462220 69.462220 69.462220
37 59 90.138782 90.138782 90.138782
37 60 85.146932 85.146932 85.146932
37 61 55.901699 55.901699 55.901699
37 62 28.284271 28.284271 28.284271
37 63 50.000000 50.000000 50.000000
37 64 65.000000 65.000000 65.000000
37 65 58.523500 58.523500 58.523500
37 66 54.083269 54.083269 54.083269
37 67 53.665631 53.665631 53.665631
37 68 43.011626 43.011626 43.011626
37 69 35.355339 35.355339 35.355339
37 70 47.381431 47.381431 47.381431
37 71 34.000000 34.000000 34.000000
37 72 30.000000 30.000000 30.000000
37 73 20.099751 20.099751 20.099751
37 74 67.779053 67.779053 67.779053
37 75 79.056942 79.056942 79.056942
37 76 100.000000 100.000000 100.000000
37 77 73.171033 73.171033 73.171033
37 78 92.130342 92.130342 92.130342
37 79 63.953108 63.953108 63.953108
37 80 61.400326 61.400326 61.400326
37 81 42.047592 42.047592 42.047592
37 82 31.384710 31.384710 31.384710
37 83 56.639209 56.639209 56.639209
37 84 60.827625 60.827625 60.827625
37 85 56.568542 56.568542 56.568542
37 86 62.032250 62.032250 62.032250
37 87 75.213031 75.213031 75.213031
37 88 80.808415 80.808415 80.808415
37 89 49.091751 49.091751 49.091751
37 90 80.024996 80.024996 80.024996
37 91 47.201695 47.201695 47.201695
37 92 45.880279 45.880279 45.880279
37 93 43.680659 43.680659 43.680659
37 94 33.241540 33.241540 33.241540
37 95 37.854986 37.854986 37.854986
37 96 48.836462 48.836462 48.836462
37 97 32.572995 32.572995 32.572995
37 98 90.609050 90.609050 90.609050
37 99 51.088159 51.088159 51.088159
37 100 63.411355 63.411355 63.411355
37 101 38.470768 38.470768 38.470768
38 1 40.607881 40.607881 40.607881
38 2 40.112342 40.112342 40.112342
38 3 43.566042 43.566042 43.566042
38 4 43.104524 43.104524 43.104524
38 5 45.044423 45.044423 45.044423
38 6 45.099889 45.099889 45.099889
38 7 47.518417 47.518417 47.518417
38 8 50.487622 50.487622 50.487622
38 9 50.039984 50.039984 50.039984
38 10 72.346389 72.346389 72.346389
38 11 69.202601 69.202601 69.202601
38 12 70.802542 70.802542 70.802542
38 13 67.955868 67.955868 67.955868
38 14 76.216796 76.216796 76.216796
38 15 70.491134 70.491134 70.491134
38 16 75.716577 75.716577 75.716577
38 17 77.388630 77.388630 77.388630
38 18 74.793048 74.793048 74.793048
38 19 79.812280 79.812280 79.812280
38 20 75.584390 75.584390 75.584390
38 21 70.837843 70.837843 70.837843
38 22 80.956779 80.956779 80.956779
38 23 71.512237 71.512237 71.512237
38 24 81.596569 81.596569 81.596569
38 25 72.235725 72.235725 72.235725
38 26 82.637764 82.637764 82.637764
38 27 60.033324 60.033324 60.033324
38 28 55.758407 55.758407 55.758407
38 29 58.591808 58.591808 58.591808
38 30 53.235327 53.235327 53.235327
38 31 56.859476 56.859476 56.859476
38 32 52.325902 52.325902 52.325902
38 33 56.462377 56.462377 56.462377
38 34 60.406953 60.406953 60.406953
38 35 51.078371 51.078371 51.078371
38 36 3.605551 3.605551 3.605551
38 37 3.000000 3.000000 3.000000
38 39 3.605551 3.605551 3.605551

38 40 5.385165 5.385165 5.385165
38 41 5.830952 5.830952 5.830952
38 42 9.899495 9.899495 9.899495
38 43 10.198039 10.198039 10.198039
38 44 10.440307 10.440307 10.440307
38 45 10.000000 10.000000 10.000000
38 46 45.000000 45.000000 45.000000
38 47 47.042534 47.042534 47.042534
38 48 73.061618 73.061618 73.061618
38 49 80.361682 80.361682 80.361682
38 50 73.681748 73.681748 73.681748
38 51 47.518417 47.518417 47.518417
38 52 62.801274 62.801274 62.801274
38 53 65.604878 65.604878 65.604878
38 54 55.217751 55.217751 55.217751
38 55 24.166092 24.166092 24.166092
38 56 41.340053 41.340053 41.340053
38 57 49.335586 49.335586 49.335586
38 58 66.887966 66.887966 66.887966
38 59 87.658428 87.658428 87.658428
38 60 82.879430 82.879430 82.879430
38 61 54.626001 54.626001 54.626001
38 62 26.248809 26.248809 26.248809
38 63 47.000000 47.000000 47.000000
38 64 62.000000 62.000000 62.000000
38 65 55.713553 55.713553 55.713553
38 66 51.613952 51.613952 51.613952
38 67 51.000000 51.000000 51.000000
38 68 40.012498 40.012498 40.012498
38 69 33.301652 33.301652 33.301652
38 70 45.343136 45.343136 45.343136
38 71 32.695565 32.695565 32.695565
38 72 27.000000 27.000000 27.000000
38 73 17.117243 17.117243 17.117243
38 74 66.730802 66.730802 66.730802
38 75 76.609399 76.609399 76.609399
38 76 97.616597 97.616597 97.616597
38 77 70.178344 70.178344 70.178344
38 78 89.470666 89.470666 89.470666
38 79 62.649820 62.649820 62.649820
38 80 60.638272 60.638272 60.638272
38 81 39.357337 39.357337 39.357337
38 82 28.844410 28.844410 28.844410
38 83 54.451814 54.451814 54.451814
38 84 58.180753 58.180753 58.180753
38 85 53.600373 53.600373 53.600373
38 86 59.033889 59.033889 59.033889
38 87 72.801099 72.801099 72.801099
38 88 78.568442 78.568442 78.568442
38 89 47.507894 47.507894 47.507894
38 90 77.025970 77.025970 77.025970
38 91 44.821870 44.821870 44.821870
38 92 43.081318 43.081318 43.081318
38 93 40.804412 40.804412 40.804412
38 94 30.265492 30.265492 30.265492
38 95 34.928498 34.928498 34.928498
38 96 45.891176 45.891176 45.891176
38 97 29.732137 29.732137 29.732137
38 98 88.413800 88.413800 88.413800
38 99 49.203658 49.203658 49.203658
38 100 61.073726 61.073726 61.073726
38 101 37.161808 37.161808 37.161808
39 1 37.202150 37.202150 37.202150
39 2 37.336309 37.336309 37.336309
39 3 40.311289 40.311289 40.311289
39 4 40.311289 40.311289 40.311289
39 5 42.000000 42.000000 42.000000
39 6 42.296572 42.296572 42.296572
39 7 44.283180 44.283180 44.283180
39 8 47.265209 47.265209 47.265209
39 9 47.000000 47.000000 47.000000
39 10 68.767725 68.767725 68.767725
39 11 65.604878 65.604878 65.604878
39 12 67.201190 67.201190 67.201190
39 13 64.350602 64.350602 64.350602
39 14 72.622311 72.622311 72.622311
39 15 66.887966 66.887966 66.887966
39 16 72.111026 72.111026 72.111026
39 17 73.783467 73.783467 73.783467
39 18 71.196910 71.196910 71.196910
39 19 77.129761 77.129761 77.129761
39 20 72.801099 72.801099 72.801099
39 21 68.007353 68.007353 68.007353
39 22 78.160092 78.160092 78.160092
39 23 68.622154 68.622154 68.622154
39 24 78.746428 78.746428 78.746428
39 25 69.289249 69.289249 69.289249
39 26 79.711982 79.711982 79.711982
39 27 59.908263 59.908263 59.908263
39 28 55.803226 55.803226 55.803226
39 29 58.309519 58.309519 58.309519
39 30 53.000000 53.000000 53.000000
39 31 56.356011 56.356011 56.356011
39 32 51.971146 51.971146 51.971146
39 33 55.901699 55.901699 55.901699

39 34 59.615434 59.615434 59.615434
39 35 50.537115 50.537115 50.537115
39 36 7.071068 7.071068 7.071068
39 37 5.830952 5.830952 5.830952
39 38 3.605551 3.605551 3.605551
39 40 2.000000 2.000000 2.000000
39 41 5.385165 5.385165 5.385165
39 42 6.403124 6.403124 6.403124
39 43 7.000000 7.000000 7.000000
39 44 8.602325 8.602325 8.602325
39 45 7.280110 7.280110 7.280110
39 46 42.047592 42.047592 42.047592
39 47 44.000000 44.000000 44.000000
39 48 69.462220 69.462220 69.462220
39 49 77.620873 77.620873 77.620873
39 50 70.880181 70.880181 70.880181
39 51 46.097722 46.097722 46.097722
39 52 60.406953 60.406953 60.406953
39 53 62.201286 62.201286 62.201286
39 54 51.613952 51.613952 51.613952
39 55 21.189620 21.189620 21.189620
39 56 37.735925 37.735925 37.735925
39 57 46.572524 46.572524 46.572524
39 58 63.631753 63.631753 63.631753
39 59 84.314886 84.314886 84.314886
39 60 79.397733 79.397733 79.397733
39 61 51.078371 51.078371 51.078371
39 62 22.671568 22.671568 22.671568
39 63 45.099889 45.099889 45.099889
39 64 60.074953 60.074953 60.074953
39 65 52.810984 52.810984 52.810984
39 66 48.259714 48.259714 48.259714
39 67 47.853944 47.853944 47.853944
39 68 38.052595 38.052595 38.052595
39 69 29.732137 29.732137 29.732137
39 70 41.773197 41.773197 41.773197
39 71 29.154759 29.154759 29.154759
39 72 25.179357 25.179357 25.179357
39 73 15.033296 15.033296 15.033296
39 74 63.245553 63.245553 63.245553
39 75 73.239334 73.239334 73.239334
39 76 94.201911 94.201911 94.201911
39 77 68.029405 68.029405 68.029405
39 78 86.313383 86.313383 86.313383
39 79 59.093147 59.093147 59.093147
39 80 57.271284 57.271284 57.271284
39 81 36.249138 36.249138 36.249138
39 82 25.553865 25.553865 25.553865
39 83 50.931326 50.931326 50.931326
39 84 55.009090 55.009090 55.009090
39 85 51.244512 51.244512 51.244512
39 86 57.008771 57.008771 57.008771
39 87 69.404611 69.404611 69.404611
39 88 75.073298 75.073298 75.073298
39 89 43.908997 43.908997 43.908997
39 90 75.166482 75.166482 75.166482
39 91 41.400483 41.400483 41.400483
39 92 40.162171 40.162171 40.162171
39 93 38.078866 38.078866 38.078866
39 94 28.017851 28.017851 28.017851
39 95 32.388269 32.388269 32.388269
39 96 43.416587 43.416587 43.416587
39 97 26.925824 26.925824 26.925824
39 98 84.899941 84.899941 84.899941
39 99 45.607017 45.607017 45.607017
39 100 57.628118 57.628118 57.628118
39 101 33.615473 33.615473 33.615473
40 1 36.055513 36.055513 36.055513
40 2 35.355339 35.355339 35.355339
40 3 38.327536 38.327536 38.327536
40 4 38.327536 38.327536 38.327536
40 5 40.000000 40.000000 40.000000
40 6 40.311289 40.311289 40.311289
40 7 42.296572 42.296572 42.296572
40 8 45.276926 45.276926 45.276926
40 9 45.000000 45.000000 45.000000
40 10 67.268120 67.268120 67.268120
40 11 64.031242 64.031242 64.031242
40 12 65.604878 65.604878 65.604878
40 13 62.681736 62.681736 62.681736
40 14 71.063352 71.063352 71.063352
40 15 65.192024 65.192024 65.192024
40 16 70.455660 70.455660 70.455660
40 17 72.111026 72.111026 72.111026
40 18 69.462220 69.462220 69.462220
40 19 76.687678 76.687678 76.687678
40 20 72.277244 72.277244 72.277244
40 21 67.446275 67.446275 67.446275
40 22 77.620873 77.620873 77.620873
40 23 68.007353 68.007353 68.007353
40 24 78.160092 78.160092 78.160092
40 25 68.622154 68.622154 68.622154
40 26 79.056942 79.056942 79.056942
40 27 61.032778 61.032778 61.032778
40 28 57.008771 57.008771 57.008771

40 29 59.363288 59.363288 59.363288
40 30 54.083269 54.083269 54.083269
40 31 57.306195 57.306195 57.306195
40 32 53.000000 53.000000 53.000000
40 33 56.824291 56.824291 56.824291
40 34 60.415230 60.415230 60.415230
40 35 51.478151 51.478151 51.478151
40 36 8.602325 8.602325 8.602325
40 37 7.071068 7.071068 7.071068
40 38 5.385165 5.385165 5.385165
40 39 2.000000 2.000000 2.000000
40 41 5.000000 5.000000 5.000000
40 42 5.385165 5.385165 5.385165
40 43 5.000000 5.000000 5.000000
40 44 7.071068 7.071068 7.071068
40 45 5.385165 5.385165 5.385165
40 46 40.049969 40.049969 40.049969
40 47 42.000000 42.000000 42.000000
40 48 67.742158 67.742158 67.742158
40 49 77.129761 77.129761 77.129761
40 50 70.342022 70.342022 70.342022
40 51 46.572524 46.572524 46.572524
40 52 60.207973 60.207973 60.207973
40 53 61.032778 61.032778 61.032778
40 54 50.000000 50.000000 50.000000
40 55 20.615528 20.615528 20.615528
40 56 36.055513 36.055513 36.055513
40 57 46.097722 46.097722 46.097722
40 58 62.649820 62.649820 62.649820
40 59 83.216585 83.216585 83.216585
40 60 78.102497 78.102497 78.102497
40 61 49.244289 49.244289 49.244289
40 62 21.213203 21.213203 21.213203
40 63 45.276926 45.276926 45.276926
40 64 60.207973 60.207973 60.207973
40 65 52.201533 52.201533 52.201533
40 66 47.169906 47.169906 47.169906
40 67 47.010637 47.010637 47.010637
40 68 38.209946 38.209946 38.209946
40 69 28.284271 28.284271 28.284271
40 70 40.311289 40.311289 40.311289
40 71 27.313001 27.313001 27.313001
40 72 25.495098 25.495098 25.495098
40 73 15.297059 15.297059 15.297059
40 74 61.351447 61.351447 61.351447
40 75 72.111026 72.111026 72.111026
40 76 93.005376 93.005376 93.005376
40 77 68.000000 68.000000 68.000000
40 78 85.428333 85.428333 85.428333
40 79 57.271284 57.271284 57.271284
40 80 55.317267 55.317267 55.317267
40 81 35.468296 35.468296 35.468296
40 82 24.596748 24.596748 24.596748
40 83 49.578221 49.578221 49.578221
40 84 54.129474 54.129474 54.129474
40 85 51.088159 51.088159 51.088159
40 86 57.078893 57.078893 57.078893
40 87 68.242216 68.242216 68.242216
40 88 73.756356 73.756356 73.756356
40 89 42.190046 42.190046 42.190046
40 90 75.325958 75.325958 75.325958
40 91 40.224371 40.224371 40.224371
40 92 39.560081 39.560081 39.560081
40 93 37.656341 37.656341 37.656341
40 94 28.017851 28.017851 28.017851
40 95 32.140317 32.140317 32.140317
40 96 43.185646 43.185646 43.185646
40 97 26.476405 26.476405 26.476405
40 98 83.546394 83.546394 83.546394
40 99 44.045431 44.045431 44.045431
40 100 56.400355 56.400355 56.400355
40 101 31.780497 31.780497 31.780497
41 1 40.311289 40.311289 40.311289
41 2 35.000000 35.000000 35.000000
41 3 39.293765 39.293765 39.293765
41 4 38.000000 38.000000 38.000000
41 5 40.311289 40.311289 40.311289
41 6 40.000000 40.000000 40.000000
41 7 43.174066 43.174066 43.174066
41 8 46.097722 46.097722 46.097722
41 9 45.276926 45.276926 45.276926
41 10 70.710678 70.710678 70.710678
41 11 67.268120 67.268120 67.268120
41 12 68.767725 68.767725 68.767725
41 13 65.604878 65.604878 65.604878
41 14 74.330344 74.330344 74.330344
41 15 68.007353 68.007353 68.007353
41 16 73.409809 73.409809 73.409809
41 17 75.000000 75.000000 75.000000
41 18 72.111026 72.111026 72.111026
41 19 81.584312 81.584312 81.584312
41 20 77.129761 77.129761 77.129761
41 21 72.277244 72.277244 72.277244
41 22 82.462113 82.462113 82.462113
41 23 72.801099 72.801099 72.801099

41 24 82.969874 82.969874 82.969874
41 25 73.375745 73.375745 73.375745
41 26 83.815273 83.815273 83.815273
41 27 65.192024 65.192024 65.192024
41 28 61.032778 61.032778 61.032778
41 29 63.631753 63.631753 63.631753
41 30 58.309519 58.309519 58.309519
41 31 61.717096 61.717096 61.717096
41 32 57.306195 57.306195 57.306195
41 33 61.269895 61.269895 61.269895
41 34 65.000000 65.000000 65.000000
41 35 55.901699 55.901699 55.901699
41 36 7.000000 7.000000 7.000000
41 37 5.000000 5.000000 5.000000
41 38 5.830952 5.830952 5.830952
41 39 5.385165 5.385165 5.385165
41 40 5.000000 5.000000 5.000000
41 42 10.198039 10.198039 10.198039
41 43 7.071068 7.071068 7.071068
41 44 5.000000 5.000000 5.000000
41 45 5.830952 5.830952 5.830952
41 46 40.112342 40.112342 40.112342
41 47 42.296572 42.296572 42.296572
41 48 70.455660 70.455660 70.455660
41 49 82.000000 82.000000 82.000000
41 50 75.186435 75.186435 75.186435
41 51 51.419841 51.419841 51.419841
41 52 65.192024 65.192024 65.192024
41 53 65.192024 65.192024 65.192024
41 54 53.150729 53.150729 53.150729
41 55 25.495098 25.495098 25.495098
41 56 39.051248 39.051248 39.051248
41 57 50.990195 50.990195 50.990195
41 58 67.082039 67.082039 67.082039
41 59 87.464278 87.464278 87.464278
41 60 82.006097 82.006097 82.006097
41 61 51.478151 51.478151 51.478151
41 62 25.000000 25.000000 25.000000
41 63 50.249378 50.249378 50.249378
41 64 65.192024 65.192024 65.192024
41 65 57.008771 57.008771 57.008771
41 66 51.478151 51.478151 51.478151
41 67 51.623638 51.623638 51.623638
41 68 43.185646 43.185646 43.185646
41 69 32.015621 32.015621 32.015621
41 70 43.931765 43.931765 43.931765
41 71 29.681644 29.681644 29.681644
41 72 30.413813 30.413813 30.413813
41 73 20.223748 20.223748 20.223748
41 74 63.158531 63.158531 63.158531
41 75 76.321688 76.321688 76.321688
41 76 97.082439 97.082439 97.082439
41 77 73.000000 73.000000 73.000000
41 78 89.961103 89.961103 89.961103
41 79 59.539903 59.539903 59.539903
41 80 56.612719 56.612719 56.612719
41 81 40.162171 40.162171 40.162171
41 82 29.154759 29.154759 29.154759
41 83 53.413481 53.413481 53.413481
41 84 58.694122 58.694122 58.694122
41 85 56.080300 56.080300 56.080300
41 86 62.072538 62.072538 62.072538
41 87 72.401657 72.401657 72.401657
41 88 77.620873 77.620873 77.620873
41 89 45.000000 45.000000 45.000000
41 90 80.305666 80.305666 80.305666
41 91 44.418465 44.418465 44.418465
41 92 44.384682 44.384682 44.384682
41 93 42.579338 42.579338 42.579338
41 94 33.015148 33.015148 33.015148
41 95 37.121422 37.121422 37.121422
41 96 48.166378 48.166378 48.166378
41 97 31.400637 31.400637 31.400637
41 98 87.321246 87.321246 87.321246
41 99 47.381431 47.381431 47.381431
41 100 60.464866 60.464866 60.464866
41 101 34.132096 34.132096 34.132096
42 1 30.805844 30.805844 30.805844
42 2 34.481879 34.481879 34.481879
42 3 36.000000 36.000000 36.000000
42 4 37.363083 37.363083 37.363083
42 5 38.327536 38.327536 38.327536
42 6 39.293765 39.293765 39.293765
42 7 40.000000 40.000000 40.000000
42 8 43.000000 43.000000 43.000000
42 9 43.289722 43.289722 43.289722
42 10 62.481997 62.481997 62.481997
42 11 59.405387 59.405387 59.405387
42 12 61.032778 61.032778 61.032778
42 13 58.309519 58.309519 58.309519
42 14 66.400301 66.400301 66.400301
42 15 60.901560 60.901560 60.901560
42 16 66.037868 66.037868 66.037868
42 17 67.742158 67.742158 67.742158
42 18 65.299311 65.299311 65.299311

42 19 71.386273 71.386273 71.386273
42 20 66.940272 66.940272 66.940272
42 21 62.096699 62.096699 62.096699
42 22 72.277244 72.277244 72.277244
42 23 62.641839 62.641839 62.641839
42 24 72.801099 72.801099 72.801099
42 25 63.245553 63.245553 63.245553
42 26 73.681748 73.681748 73.681748
42 27 58.258047 58.258047 58.258047
42 28 54.488531 54.488531 54.488531
42 29 56.400355 56.400355 56.400355
42 30 51.224994 51.224994 51.224994
42 31 54.083269 54.083269 54.083269
42 32 50.000000 50.000000 50.000000
42 33 53.535035 53.535035 53.535035
42 34 56.824291 56.824291 56.824291
42 35 48.259714 48.259714 48.259714
42 36 13.453624 13.453624 13.453624
42 37 12.206556 12.206556 12.206556
42 38 9.899495 9.899495 9.899495
42 39 6.403124 6.403124 6.403124
42 40 5.385165 5.385165 5.385165
42 41 10.198039 10.198039 10.198039
42 43 5.830952 5.830952 5.830952
42 44 10.440307 10.440307 10.440307
42 45 7.615773 7.615773 7.615773
42 46 38.639358 38.639358 38.639358
42 47 40.311289 40.311289 40.311289
42 48 63.529521 63.529521 63.529521
42 49 71.805292 71.805292 71.805292
42 50 65.000000 65.000000 65.000000
42 51 42.379240 42.379240 42.379240
42 52 55.081757 55.081757 55.081757
42 53 55.803226 55.803226 55.803226
42 54 45.486262 45.486262 45.486262
42 55 15.297059 15.297059 15.297059
42 56 31.764760 31.764760 31.764760
42 57 40.792156 40.792156 40.792156
42 58 57.306195 57.306195 57.306195
42 59 77.935871 77.935871 77.935871
42 60 73.000000 73.000000 73.000000
42 61 45.541190 45.541190 45.541190
42 62 16.401219 16.401219 16.401219
42 63 40.607881 40.607881 40.607881
42 64 55.443665 55.443665 55.443665
42 65 46.840154 46.840154 46.840154
42 66 41.880783 41.880783 41.880783
42 67 41.629317 41.629317 41.629317
42 68 33.541020 33.541020 33.541020
42 69 23.430749 23.430749 23.430749
42 70 35.468296 35.468296 35.468296
42 71 23.769729 23.769729 23.769729
42 72 21.189620 21.189620 21.189620
42 73 11.180340 11.180340 11.180340
42 74 57.974132 57.974132 57.974132
42 75 66.850580 66.850580 66.850580
42 76 87.800911 87.800911 87.800911
42 77 63.031738 63.031738 63.031738
42 78 80.056230 80.056230 80.056230
42 79 53.488316 53.488316 53.488316
42 80 52.469038 52.469038 52.469038
42 81 30.083218 30.083218 30.083218
42 82 19.235384 19.235384 19.235384
42 83 44.553339 44.553339 44.553339
42 84 48.754487 48.754487 48.754487
42 85 46.010868 46.010868 46.010868
42 86 52.239832 52.239832 52.239832
42 87 63.007936 63.007936 63.007936
42 88 68.680419 68.680419 68.680419
42 89 38.013156 38.013156 38.013156
42 90 70.576200 70.576200 70.576200
42 91 35.000000 35.000000 35.000000
42 92 34.205263 34.205263 34.205263
42 93 32.388269 32.388269 32.388269
42 94 23.194827 23.194827 23.194827
42 95 27.018512 27.018512 27.018512
42 96 38.052595 38.052595 38.052595
42 97 21.213203 21.213203 21.213203
42 98 78.517514 78.517514 78.517514
42 99 39.408121 39.408121 39.408121
42 100 51.224994 51.224994 51.224994
42 101 28.160256 28.160256 28.160256
43 1 33.541020 33.541020 33.541020
43 2 30.413813 30.413813 30.413813
43 3 33.376639 33.376639 33.376639
43 4 33.376639 33.376639 33.376639
43 5 35.000000 35.000000 35.000000
43 6 35.355339 35.355339 35.355339
43 7 37.336309 37.336309 37.336309
43 8 40.311289 40.311289 40.311289
43 9 40.000000 40.000000 40.000000
43 10 63.639610 63.639610 63.639610
43 11 60.207973 60.207973 60.207973
43 12 61.717096 61.717096 61.717096
43 13 58.600341 58.600341 58.600341

43 14 67.268120 67.268120 67.268120
43 15 61.032778 61.032778 61.032778
43 16 66.400301 66.400301 66.400301
43 17 68.007353 68.007353 68.007353
43 18 65.192024 65.192024 65.192024
43 19 75.802375 75.802375 75.802375
43 20 71.196910 71.196910 71.196910
43 21 66.287254 66.287254 66.287254
43 22 76.485293 76.485293 76.485293
43 23 66.708320 66.708320 66.708320
43 24 76.902536 76.902536 76.902536
43 25 67.186308 67.186308 67.186308
43 26 77.620873 77.620873 77.620873
43 27 64.031242 64.031242 64.031242
43 28 60.207973 60.207973 60.207973
43 29 62.201286 62.201286 62.201286
43 30 57.008771 57.008771 57.008771
43 31 59.908263 59.908263 59.908263
43 32 55.803226 55.803226 55.803226
43 33 59.363288 59.363288 59.363288
43 34 62.649820 62.649820 62.649820
43 35 54.083269 54.083269 54.083269
43 36 13.000000 13.000000 13.000000
43 37 11.180340 11.180340 11.180340
43 38 10.198039 10.198039 10.198039
43 39 7.000000 7.000000 7.000000
43 40 5.000000 5.000000 5.000000
43 41 7.071068 7.071068 7.071068
43 42 5.830952 5.830952 5.830952
43 44 5.000000 5.000000 5.000000
43 45 2.000000 2.000000 2.000000
43 46 35.057096 35.057096 35.057096
43 47 37.000000 37.000000 37.000000
43 48 63.513778 63.513778 63.513778
43 49 76.118329 76.118329 76.118329
43 50 69.231496 69.231496 69.231496
43 51 48.104054 48.104054 48.104054
43 52 60.000000 60.000000 60.000000
43 53 58.309519 58.309519 58.309519
43 54 46.097722 46.097722 46.097722
43 55 20.000000 20.000000 20.000000
43 56 32.015621 32.015621 32.015621
43 57 45.276926 45.276926 45.276926
43 58 60.415230 60.415230 60.415230
43 59 80.622577 80.622577 80.622577
43 60 75.000000 75.000000 75.000000
43 61 44.721360 44.721360 44.721360
43 62 18.027756 18.027756 18.027756
43 63 46.097722 46.097722 46.097722
43 64 60.827625 60.827625 60.827625
43 65 50.990195 50.990195 50.990195
43 66 44.721360 44.721360 44.721360
43 67 45.221676 45.221676 45.221676
43 68 39.051248 39.051248 39.051248
43 69 25.000000 25.000000 25.000000
43 70 36.878178 36.878178 36.878178
43 71 22.825424 22.825424 22.825424
43 72 26.925824 26.925824 26.925824
43 73 17.000000 17.000000 17.000000
43 74 56.648036 56.648036 56.648036
43 75 69.462220 69.462220 69.462220
43 76 90.138782 90.138782 90.138782
43 77 68.183576 68.183576 68.183576
43 78 83.384651 83.384651 83.384651
43 79 52.773099 52.773099 52.773099
43 80 50.447993 50.447993 50.447993
43 81 33.955854 33.955854 33.955854
43 82 22.803509 22.803509 22.803509
43 83 46.400431 46.400431 46.400431
43 84 52.201533 52.201533 52.201533
43 85 51.039201 51.039201 51.039201
43 86 57.558666 57.558666 57.558666
43 87 65.513357 65.513357 65.513357
43 88 70.604532 70.604532 70.604532
43 89 38.013156 38.013156 38.013156
43 90 75.953933 75.953933 75.953933
43 91 37.589892 37.589892 37.589892
43 92 38.470768 38.470768 38.470768
43 93 37.054015 37.054015 37.054015
43 94 28.635642 28.635642 28.635642
43 95 32.062439 32.062439 32.062439
43 96 43.011626 43.011626 43.011626
43 97 26.000000 26.000000 26.000000
43 98 80.280757 80.280757 80.280757
43 99 40.311289 40.311289 40.311289
43 100 53.535035 53.535035 53.535035
43 101 27.294688 27.294688 27.294688
44 1 38.078866 38.078866 38.078866
44 2 30.000000 30.000000 30.000000
44 3 34.481879 34.481879 34.481879
44 4 33.000000 33.000000 33.000000
44 5 35.355339 35.355339 35.355339
44 6 35.000000 35.000000 35.000000
44 7 38.327536 38.327536 38.327536
44 8 41.231056 41.231056 41.231056

44 9 40.311289 40.311289 40.311289
44 10 67.268120 67.268120 67.268120
44 11 63.639610 63.639610 63.639610
44 12 65.069194 65.069194 65.069194
44 13 61.717096 61.717096 61.717096
44 14 70.710678 70.710678 70.710678
44 15 64.031242 64.031242 64.031242
44 16 69.526973 69.526973 69.526973
44 17 71.063352 71.063352 71.063352
44 18 68.007353 68.007353 68.007353
44 19 80.752709 80.752709 80.752709
44 20 76.118329 76.118329 76.118329
44 21 71.196910 71.196910 71.196910
44 22 81.394103 81.394103 81.394103
44 23 71.589105 71.589105 71.589105
44 24 81.786307 81.786307 81.786307
44 25 72.034714 72.034714 72.034714
44 26 82.462113 82.462113 82.462113
44 27 68.007353 68.007353 68.007353
44 28 64.031242 64.031242 64.031242
44 29 66.287254 66.287254 66.287254
44 30 61.032778 61.032778 61.032778
44 31 64.140471 64.140471 64.140471
44 32 59.908263 59.908263 59.908263
44 33 63.631753 63.631753 63.631753
44 34 67.082039 67.082039 67.082039
44 35 58.309519 58.309519 58.309519
44 36 12.000000 12.000000 12.000000
44 37 10.000000 10.000000 10.000000
44 38 10.440307 10.440307 10.440307
44 39 8.602325 8.602325 8.602325
44 40 7.071068 7.071068 7.071068
44 41 5.000000 5.000000 5.000000
44 42 10.440307 10.440307 10.440307
44 43 5.000000 5.000000 5.000000
44 45 3.000000 3.000000 3.000000
44 46 35.128336 35.128336 35.128336
44 47 37.336309 37.336309 37.336309
44 48 66.400301 66.400301 66.400301
44 49 81.049368 81.049368 81.049368
44 50 74.148500 74.148500 74.148500
44 51 52.810984 52.810984 52.810984
44 52 65.000000 65.000000 65.000000
44 53 62.649820 62.649820 62.649820
44 54 49.497475 49.497475 49.497475
44 55 25.000000 25.000000 25.000000
44 56 35.355339 35.355339 35.355339
44 57 50.249378 50.249378 50.249378
44 58 65.000000 65.000000 65.000000
44 59 85.000000 85.000000 85.000000
44 60 79.056942 79.056942 79.056942
44 61 47.169906 47.169906 47.169906
44 62 22.360680 22.360680 22.360680
44 63 50.990195 50.990195 50.990195
44 64 65.764732 65.764732 65.764732
44 65 55.901699 55.901699 55.901699
44 66 49.244289 49.244289 49.244289
44 67 50.000000 50.000000 50.000000
44 68 43.931765 43.931765 43.931765
44 69 29.154759 29.154759 29.154759
44 70 40.804412 40.804412 40.804412
44 71 25.612497 25.612497 25.612497
44 72 31.622777 31.622777 31.622777
44 73 21.540659 21.540659 21.540659
44 74 58.600341 58.600341 58.600341
44 75 73.824115 73.824115 73.824115
44 76 94.339811 94.339811 94.339811
44 77 73.171033 73.171033 73.171033
44 78 88.022724 88.022724 88.022724
44 79 55.226805 55.226805 55.226805
44 80 51.865210 51.865210 51.865210
44 81 38.832976 38.832976 38.832976
44 82 27.658633 27.658633 27.658633
44 83 50.477718 50.477718 50.477718
44 84 56.920998 56.920998 56.920998
44 85 56.035703 56.035703 56.035703
44 86 62.513998 62.513998 62.513998
44 87 69.835521 69.835521 69.835521
44 88 74.632433 74.632433 74.632433
44 89 41.109610 41.109610 41.109610
44 90 80.894994 80.894994 80.894994
44 91 42.047592 42.047592 42.047592
44 92 43.416587 43.416587 43.416587
44 93 42.047592 42.047592 42.047592
44 94 33.541020 33.541020 33.541020
44 95 37.054015 37.054015 37.054015
44 96 48.010416 48.010416 48.010416
44 97 31.000000 31.000000 31.000000
44 98 84.202138 84.202138 84.202138
44 99 43.931765 43.931765 43.931765
44 100 57.801384 57.801384 57.801384
44 101 30.000000 30.000000 30.000000
45 1 35.341194 35.341194 35.341194
45 2 30.149627 30.149627 30.149627
45 3 33.734256 33.734256 33.734256

45 4 33.136083 33.136083 33.136083
45 5 35.057096 35.057096 35.057096
45 6 35.128336 35.128336 35.128336
45 7 37.656341 37.656341 37.656341
45 8 40.607881 40.607881 40.607881
45 9 40.049969 40.049969 40.049969
45 10 65.069194 65.069194 65.069194
45 11 61.554854 61.554854 61.554854
45 12 63.031738 63.031738 63.031738
45 13 59.816386 59.816386 59.816386
45 14 68.622154 68.622154 68.622154
45 15 62.201286 62.201286 62.201286
45 16 67.623960 67.623960 67.623960
45 17 69.202601 69.202601 69.202601
45 18 66.287254 66.287254 66.287254
45 19 77.781746 77.781746 77.781746
45 20 73.164199 73.164199 73.164199
45 21 68.249542 68.249542 68.249542
45 22 78.447435 78.447435 78.447435
45 23 68.658576 68.658576 68.658576
45 24 78.854296 78.854296 78.854296
45 25 69.123079 69.123079 69.123079
45 26 79.555012 79.555012 79.555012
45 27 65.604878 65.604878 65.604878
45 28 61.717096 61.717096 61.717096
45 29 63.820060 63.820060 63.820060
45 30 58.600341 58.600341 58.600341
45 31 61.587336 61.587336 61.587336
45 32 57.428216 57.428216 57.428216
45 33 61.057350 61.057350 61.057350
45 34 64.412732 64.412732 64.412732
45 35 55.758407 55.758407 55.758407
45 36 12.369317 12.369317 12.369317
45 37 10.440307 10.440307 10.440307
45 38 10.000000 10.000000 10.000000
45 39 7.280110 7.280110 7.280110
45 40 5.385165 5.385165 5.385165
45 41 5.830952 5.830952 5.830952
45 42 7.615773 7.615773 7.615773
45 43 2.000000 2.000000 2.000000
45 44 3.000000 3.000000 3.000000
45 46 35.000000 35.000000 35.000000
45 47 37.054015 37.054015 37.054015
45 48 64.637450 64.637450 64.637450
45 49 78.089692 78.089692 78.089692
45 50 71.196910 71.196910 71.196910
45 51 49.979996 49.979996 49.979996
45 52 62.000000 62.000000 62.000000
45 53 60.033324 60.033324 60.033324
45 54 47.423623 47.423623 47.423623
45 55 22.000000 22.000000 22.000000
45 56 33.301652 33.301652 33.301652
45 57 47.265209 47.265209 47.265209
45 58 62.241465 62.241465 62.241465
45 59 82.365041 82.365041 82.365041
45 60 76.609399 76.609399 76.609399
45 61 45.650849 45.650849 45.650849
45 62 19.723083 19.723083 19.723083
45 63 48.052055 48.052055 48.052055
45 64 62.801274 62.801274 62.801274
45 65 52.952809 52.952809 52.952809
45 66 46.518813 46.518813 46.518813
45 67 47.127487 47.127487 47.127487
45 68 41.000000 41.000000 41.000000
45 69 26.627054 26.627054 26.627054
45 70 38.418745 38.418745 38.418745
45 71 23.853721 23.853721 23.853721
45 72 28.792360 28.792360 28.792360
45 73 18.788294 18.788294 18.788294
45 74 57.384667 57.384667 57.384667
45 75 71.196910 71.196910 71.196910
45 76 91.809586 91.809586 91.809586
45 77 70.178344 70.178344 70.178344
45 78 85.234969 85.234969 85.234969
45 79 53.712196 53.712196 53.712196
45 80 50.960769 50.960769 50.960769
45 81 35.902646 35.902646 35.902646
45 82 24.738634 24.738634 24.738634
45 83 48.010416 48.010416 48.010416
45 84 54.083269 54.083269 54.083269
45 85 53.037722 53.037722 53.037722
45 86 59.539903 59.539903 59.539903
45 87 67.230945 67.230945 67.230945
45 88 72.201108 72.201108 72.201108
45 89 39.204592 39.204592 39.204592
45 90 77.929455 77.929455 77.929455
45 91 39.357337 39.357337 39.357337
45 92 40.447497 40.447497 40.447497
45 93 39.051248 39.051248 39.051248
45 94 30.594117 30.594117 30.594117
45 95 34.058773 34.058773 34.058773
45 96 45.011110 45.011110 45.011110
45 97 28.000000 28.000000 28.000000
45 98 81.835200 81.835200 81.835200
45 99 41.725292 41.725292 41.725292

45 100 55.226805 55.226805 55.226805
45 101 28.301943 28.301943 28.301943
46 1 37.735925 37.735925 37.735925
46 2 5.830952 5.830952 5.830952
46 3 7.280110 7.280110 7.280110
46 4 3.605551 3.605551 3.605551
46 5 2.000000 2.000000 2.000000
46 6 3.000000 3.000000 3.000000
46 7 7.280110 7.280110 7.280110
46 8 8.602325 8.602325 8.602325
46 9 5.385165 5.385165 5.385165
46 10 48.052055 48.052055 48.052055
46 11 43.174066 43.174066 43.174066
46 12 43.680659 43.680659 43.680659
46 13 38.897301 38.897301 38.897301
46 14 49.335586 49.335586 49.335586
46 15 39.924930 39.924930 39.924930
46 16 45.694639 45.694639 45.694639
46 17 46.518813 46.518813 46.518813
46 18 42.059482 42.059482 42.059482
46 19 80.653580 80.653580 80.653580
46 20 75.286121 75.286121 75.286121
46 21 70.519501 70.519501 70.519501
46 22 79.555012 79.555012 79.555012
46 23 69.921384 69.921384 69.921384
46 24 79.075913 79.075913 79.075913
46 25 69.375788 69.375788 69.375788
46 26 78.447435 78.447435 78.447435
46 27 91.263355 91.263355 91.263355
46 28 88.509886 88.509886 88.509886
46 29 88.814413 88.814413 88.814413
46 30 84.314886 84.314886 84.314886
46 31 85.603738 85.603738 85.603738
46 32 82.661962 82.661962 82.661962
46 33 84.811556 84.811556 84.811556
46 34 86.452299 86.452299 86.452299
46 35 80.212219 80.212219 80.212219
46 36 47.095647 47.095647 47.095647
46 37 45.099889 45.099889 45.099889
46 38 45.000000 45.000000 45.000000
46 39 42.047592 42.047592 42.047592
46 40 40.049969 40.049969 40.049969
46 41 40.112342 40.112342 40.112342
46 42 38.639358 38.639358 38.639358
46 43 35.057096 35.057096 35.057096
46 44 35.128336 35.128336 35.128336
46 45 35.000000 35.000000 35.000000
46 47 2.828427 2.828427 2.828427
46 48 41.146081 41.146081 41.146081
46 49 80.081209 80.081209 80.081209
46 50 73.375745 73.375745 73.375745
46 51 70.092796 70.092796 70.092796
46 52 71.196910 71.196910 71.196910
46 53 52.239832 52.239832 52.239832
46 54 32.000000 32.000000 32.000000
46 55 41.340053 41.340053 41.340053
46 56 24.166092 24.166092 24.166092
46 57 55.758407 55.758407 55.758407
46 58 57.870545 57.870545 57.870545
46 59 72.173402 72.173402 72.173402
46 60 62.801274 62.801274 62.801274
46 61 22.561028 22.561028 22.561028
46 62 30.232433 30.232433 30.232433
46 63 65.069194 65.069194 65.069194
46 64 76.609399 76.609399 76.609399
46 65 57.697487 57.697487 57.697487
46 66 44.598206 44.598206 44.598206
46 67 49.658836 49.658836 49.658836
46 68 59.464275 59.464275 59.464275
46 69 29.732137 29.732137 29.732137
46 70 31.953091 31.953091 31.953091
46 71 19.849433 19.849433 19.849433
46 72 52.478567 52.478567 52.478567
46 73 46.238512 46.238512 46.238512
46 74 28.425341 28.425341 28.425341
46 75 62.000000 62.000000 62.000000
46 76 78.447435 78.447435 78.447435
46 77 80.622577 80.622577 80.622577
46 78 79.056942 79.056942 79.056942
46 79 28.635642 28.635642 28.635642
46 80 19.798990 19.798990 19.798990
46 81 44.204072 44.204072 44.204072
46 82 37.643060 37.643060 37.643060
46 83 39.623226 39.623226 39.623226
46 84 53.758720 53.758720 53.758720
46 85 64.637450 64.637450 64.637450
46 86 73.006849 73.006849 73.006849
46 87 58.008620 58.008620 58.008620
46 88 58.549125 58.549125 58.549125
46 89 24.331050 24.331050 24.331050
46 90 90.210864 90.210864 90.210864
46 91 38.910153 38.910153 38.910153
46 92 49.406477 49.406477 49.406477
46 93 51.088159 51.088159 51.088159
46 94 50.803543 50.803543 50.803543
46 95 50.249378 50.249378 50.249378

46 95 50.249378 50.249378 50.249378
46 96 57.628118 57.628118 57.628118
46 97 44.821870 44.821870 44.821870
46 98 65.969690 65.969690 65.969690
46 99 30.594117 30.594117 30.594117
46 100 47.381431 47.381431 47.381431
46 101 18.601075 18.601075 18.601075
47 1 37.202150 37.202150 37.202150
47 2 8.602325 8.602325 8.602325
47 3 6.403124 6.403124 6.403124
47 4 6.403124 6.403124 6.403124
47 5 2.000000 2.000000 2.000000
47 6 5.385165 5.385165 5.385165
47 7 5.000000 5.000000 5.000000
47 8 5.830952 5.830952 5.830952
47 9 3.000000 3.000000 3.000000
47 10 45.705580 45.705580 45.705580
47 11 40.792156 40.792156 40.792156
47 12 41.231056 41.231056 41.231056
47 13 36.400549 36.400549 36.400549
47 14 46.840154 46.840154 46.840154
47 15 37.336309 37.336309 37.336309
47 16 43.081318 43.081318 43.081318
47 17 43.863424 43.863424 43.863424
47 18 39.357337 39.357337 39.357337
47 19 79.378838 79.378838 79.378838
47 20 74.000000 74.000000 74.000000
47 21 69.289249 69.289249 69.289249
47 22 78.160092 78.160092 78.160092
47 23 68.622154 68.622154 68.622154
47 24 77.620873 77.620873 77.620873
47 25 68.007353 68.007353 68.007353
47 26 76.902536 76.902536 76.902536
47 27 91.809586 91.809586 91.809586
47 28 89.185201 89.185201 89.185201
47 29 89.308454 89.308454 89.308454
47 30 84.905830 84.905830 84.905830
47 31 86.023253 86.023253 86.023253
47 32 83.216585 83.216585 83.216585
47 33 85.211502 85.211502 85.211502
47 34 86.683332 86.683332 86.683332
47 35 80.709355 80.709355 80.709355
47 36 49.254441 49.254441 49.254441
47 37 47.265209 47.265209 47.265209
47 38 47.042534 47.042534 47.042534
47 39 44.000000 44.000000 44.000000
47 40 42.000000 42.000000 42.000000
47 41 42.296572 42.296572 42.296572
47 42 40.311289 40.311289 40.311289
47 43 37.000000 37.000000 37.000000
47 44 37.336309 37.336309 37.336309
47 45 37.054015 37.054015 37.054015
47 46 2.828427 2.828427 2.828427
47 48 38.483763 38.483763 38.483763
47 49 78.746428 78.746428 78.746428
47 50 72.111026 72.111026 72.111026
47 51 70.292247 70.292247 70.292247
47 52 70.491134 70.491134 70.491134
47 53 50.487622 50.487622 50.487622
47 54 30.066593 30.066593 30.066593
47 55 42.059482 42.059482 42.059482
47 56 23.323808 23.323808 23.323808
47 57 55.217751 55.217751 55.217751
47 58 56.293872 56.293872 56.293872
47 59 70.064256 70.064256 70.064256
47 60 60.530984 60.530984 60.530984
47 61 20.223748 20.223748 20.223748
47 62 30.886890 30.886890 30.886890
47 63 65.069194 65.069194 65.069194
47 64 76.216796 76.216796 76.216796
47 65 56.824291 56.824291 56.824291
47 66 43.462628 43.462628 43.462628
47 67 48.764741 48.764741 48.764741
47 68 59.665736 59.665736 59.665736
47 69 29.732137 29.732137 29.732137
47 70 30.870698 30.870698 30.870698
47 71 20.248457 20.248457 20.248457
47 72 53.235327 53.235327 53.235327
47 73 47.434165 47.434165 47.434165
47 74 25.612497 25.612497 25.612497
47 75 60.033324 60.033324 60.033324
47 76 76.118329 76.118329 76.118329
47 77 79.924965 79.924965 79.924965
47 78 77.162167 77.162167 77.162167
47 79 26.000000 26.000000 26.000000
47 80 16.970563 16.970563 16.970563
47 81 43.931765 43.931765 43.931765
47 82 38.013156 38.013156 38.013156
47 83 38.078866 38.078866 38.078866
47 84 52.554733 52.554733 52.554733
47 85 64.202804 64.202804 64.202804
47 86 72.622311 72.622311 72.622311
47 87 56.080300 56.080300 56.080300
47 88 56.320511 56.320511 56.320511
47 89 22.803509 22.803509 22.803509
47 90 89.587946 89.587946 89.587946

47 90 89.587946 89.587946 89.587946
47 91 38.078866 38.078866 38.078866
47 92 49.040799 49.040799 49.040799
47 93 50.931326 50.931326 50.931326
47 94 51.312766 51.312766 51.312766
47 95 50.447993 50.447993 50.447993
47 96 57.384667 57.384667 57.384667
47 97 45.221676 45.221676 45.221676
47 98 63.560994 63.560994 63.560994
47 99 29.120440 29.120440 29.120440
47 100 45.705580 45.705580 45.705580
47 101 18.384776 18.384776 18.384776
48 1 38.327536 38.327536 38.327536
48 2 46.141088 46.141088 46.141088
48 3 36.055513 36.055513 36.055513
48 4 44.721360 44.721360 44.721360
48 5 39.357337 39.357337 39.357337
48 6 43.863424 43.863424 43.863424
48 7 34.000000 34.000000 34.000000
48 8 32.695565 32.695565 32.695565
48 9 37.336309 37.336309 37.336309
48 10 12.806248 12.806248 12.806248
48 11 9.433981 9.433981 9.433981
48 12 7.810250 7.810250 7.810250
48 13 6.000000 6.000000 6.000000
48 14 10.440307 10.440307 10.440307
48 15 3.000000 3.000000 3.000000
48 16 5.000000 5.000000 5.000000
48 17 5.385165 5.385165 5.385165
48 18 2.000000 2.000000 2.000000
48 19 58.000000 58.000000 58.000000
48 20 53.150729 53.150729 53.150729
48 21 50.000000 50.000000 50.000000
48 22 55.172457 55.172457 55.172457
48 23 48.414874 48.414874 48.414874
48 24 53.814496 53.814496 53.814496
48 25 46.861498 46.861498 46.861498
48 26 51.855569 51.855569 51.855569
48 27 94.201911 94.201911 94.201911
48 28 93.536089 93.536089 93.536089
48 29 91.241438 91.241438 91.241438
48 30 88.566359 88.566359 88.566359
48 31 87.298339 87.298339 87.298339
48 32 86.579443 86.579443 86.579443
48 33 86.313383 86.313383 86.313383
48 34 85.375641 85.375641 85.375641
48 35 83.600239 83.600239 83.600239
48 36 76.321688 76.321688 76.321688
48 37 74.625733 74.625733 74.625733
48 38 73.061618 73.061618 73.061618
48 39 69.462220 69.462220 69.462220
48 40 67.742158 67.742158 67.742158
48 41 70.455660 70.455660 70.455660
48 42 63.529521 63.529521 63.529521
48 43 63.513778 63.513778 63.513778
48 44 66.400301 66.400301 66.400301
48 45 64.637450 64.637450 64.637450
48 46 41.146081 41.146081 41.146081
48 47 38.483763 38.483763 38.483763
48 49 56.568542 56.568542 56.568542
48 50 51.855569 51.855569 51.855569
48 51 70.710678 70.710678 70.710678
48 52 58.600341 58.600341 58.600341
48 53 27.459060 27.459060 27.459060
48 54 18.681542 18.681542 18.681542
48 55 55.081757 55.081757 55.081757
48 56 31.764760 31.764760 31.764760
48 57 49.030603 49.030603 49.030603
48 58 34.409301 34.409301 34.409301
48 59 37.336309 37.336309 37.336309
48 60 26.248809 26.248809 26.248809
48 61 19.849433 19.849433 19.849433
48 62 47.423623 47.423623 47.423623
48 63 63.788714 63.788714 63.788714
48 64 67.779053 67.779053 67.779053
48 65 45.541190 45.541190 45.541190
48 66 33.376639 33.376639 33.376639
48 67 39.812058 39.812058 39.812058
48 68 62.072538 62.072538 62.072538
48 69 40.853396 40.853396 40.853396
48 70 29.832868 29.832868 29.832868
48 71 40.804412 40.804412 40.804412
48 72 63.788714 63.788714 63.788714
48 73 64.195015 64.195015 64.195015
48 74 15.000000 15.000000 15.000000
48 75 30.805844 30.805844 30.805844
48 76 40.112342 40.112342 40.112342
48 77 66.730802 66.730802 66.730802
48 78 46.957428 46.957428 46.957428
48 79 12.529964 12.529964 12.529964
48 80 23.345235 23.345235 23.345235
48 81 45.044423 45.044423 45.044423
48 82 48.764741 48.764741 48.764741
48 83 25.079872 25.079872 25.079872
48 84 37.696154 37.696154 37.696154
48 85 57.280014 57.280014 57.280014

48 86 64.845971 64.845971 64.845971
48 87 28.319605 28.319605 28.319605
48 88 23.259407 23.259407 23.259407
48 89 25.553865 25.553865 25.553865
48 90 76.321688 76.321688 76.321688
48 91 35.057096 35.057096 35.057096
48 92 47.095647 47.095647 47.095647
48 93 51.039201 51.039201 51.039201
48 94 59.413803 59.413803 59.413803
48 95 55.081757 55.081757 55.081757
48 96 54.589376 54.589376 54.589376
48 97 53.758720 53.758720 53.758720
48 98 27.073973 27.073973 27.073973
48 99 25.000000 25.000000 25.000000
48 100 26.000000 26.000000 26.000000
48 101 36.400549 36.400549 36.400549
49 1 45.044423 45.044423 45.044423
49 2 81.786307 81.786307 81.786307
49 3 72.801099 72.801099 72.801099
49 4 82.462113 82.462113 82.462113
49 5 78.160092 78.160092 78.160092
49 6 82.969874 82.969874 82.969874
49 7 74.000000 74.000000 74.000000
49 8 75.026662 75.026662 75.026662
49 9 79.711982 79.711982 79.711982
49 10 43.863424 43.863424 43.863424
49 11 47.423623 47.423623 47.423623
49 12 48.795492 48.795492 48.795492
49 13 52.497619 52.497619 52.497619
49 14 47.634021 47.634021 47.634021
49 15 54.488531 54.488531 54.488531
49 16 53.150729 53.150729 53.150729
49 17 54.671748 54.671748 54.671748
49 18 58.000000 58.000000 58.000000
49 19 2.000000 2.000000 2.000000
49 20 5.000000 5.000000 5.000000
49 21 10.000000 10.000000 10.000000
49 22 2.000000 2.000000 2.000000
49 23 10.198039 10.198039 10.198039
49 24 4.000000 4.000000 4.000000
49 25 10.770330 10.770330 10.770330
49 26 7.000000 7.000000 7.000000
49 27 58.600341 58.600341 58.600341
49 28 60.901560 60.901560 60.901560
49 29 55.901699 55.901699 55.901699
49 30 56.603887 56.603887 56.603887
49 31 52.354560 52.354560 52.354560
49 32 54.918121 54.918121 54.918121
49 33 51.478151 51.478151 51.478151
49 34 47.423623 47.423623 47.423623
49 35 52.430907 52.430907 52.430907
49 36 83.815273 83.815273 83.815273
49 37 83.240615 83.240615 83.240615
49 38 80.361682 80.361682 80.361682
49 39 77.620873 77.620873 77.620873
49 40 77.129761 77.129761 77.129761
49 41 82.000000 82.000000 82.000000
49 42 71.805292 71.805292 71.805292
49 43 76.118329 76.118329 76.118329
49 44 81.049368 81.049368 81.049368
49 45 78.089692 78.089692 78.089692
49 46 80.081209 80.081209 80.081209
49 47 78.746428 78.746428 78.746428
49 48 56.568542 56.568542 56.568542
49 50 7.000000 7.000000 7.000000
49 51 42.426407 42.426407 42.426407
49 52 19.849433 19.849433 19.849433
49 53 30.232433 30.232433 30.232433
49 54 50.089919 50.089919 50.089919
49 55 56.515485 56.515485 56.515485
49 56 56.293872 56.293872 56.293872
49 57 31.048349 31.048349 31.048349
49 58 23.323808 23.323808 23.323808
49 59 27.459060 27.459060 27.459060
49 60 35.341194 35.341194 35.341194
49 61 61.269895 61.269895 61.269895
49 62 60.074953 60.074953 60.074953
49 63 37.802116 37.802116 37.802116
49 64 27.459060 27.459060 27.459060
49 65 25.179357 25.179357 25.179357
49 66 35.693137 35.693137 35.693137
49 67 32.015621 32.015621 32.015621
49 68 43.046487 43.046487 43.046487
49 69 55.036352 55.036352 55.036352
49 70 48.270074 48.270074 48.270074
49 71 64.381674 64.381674 64.381674
49 72 55.036352 55.036352 55.036352
49 73 63.568860 63.568860 63.568860
49 74 68.007353 68.007353 68.007353
49 75 26.627054 26.627054 26.627054
49 76 37.000000 37.000000 37.000000
49 77 19.313208 19.313208 19.313208
49 78 19.104973 19.104973 19.104973
49 79 61.294372 61.294372 61.294372
49 80 72.560320 72.560320 72.560320

49 81 42.296572 42.296572 42.296572
49 82 53.460266 53.460266 53.460266
49 83 40.853396 40.853396 40.853396
49 84 26.476405 26.476405 26.476405
49 85 28.301943 28.301943 28.301943
49 86 27.658633 27.658633 27.658633
49 87 28.319605 28.319605 28.319605
49 88 35.510562 35.510562 35.510562
49 89 55.973208 55.973208 55.973208
49 90 25.000000 25.000000 25.000000
49 91 42.296572 42.296572 42.296572
49 92 37.656341 37.656341 37.656341
49 93 39.560081 39.560081 39.560081
49 94 50.695167 50.695167 50.695167
49 95 45.541190 45.541190 45.541190
49 96 34.928498 34.928498 34.928498
49 97 50.695167 50.695167 50.695167
49 98 40.162171 40.162171 40.162171
49 99 49.648766 49.648766 49.648766
49 100 34.000000 34.000000 34.000000
49 101 62.968246 62.968246 62.968246
50 1 38.052595 38.052595 38.052595
50 2 74.953319 74.953319 74.953319
50 3 66.098411 66.098411 66.098411
50 4 75.690158 75.690158 75.690158
50 5 71.470274 71.470274 71.470274
50 6 76.243032 76.243032 76.243032
50 7 67.416615 67.416615 67.416615
50 8 68.541958 68.541958 68.541958
50 9 73.164199 73.164199 73.164199
50 10 39.408121 39.408121 39.408121
50 11 42.520583 42.520583 42.520583
50 12 44.045431 44.045431 44.045431
50 13 47.381431 47.381431 47.381431
50 14 43.566042 43.566042 43.566042
50 15 49.578221 49.578221 49.578221
50 16 48.826222 48.826222 48.826222
50 17 50.477718 50.477718 50.477718
50 18 53.413481 53.413481 53.413481
50 19 7.280110 7.280110 7.280110
50 20 2.000000 2.000000 2.000000
50 21 3.000000 3.000000 3.000000
50 22 7.280110 7.280110 7.280110
50 23 3.605551 3.605551 3.605551
50 24 8.062258 8.062258 8.062258
50 25 5.000000 5.000000 5.000000
50 26 9.899495 9.899495 9.899495
50 27 55.973208 55.973208 55.973208
50 28 57.775427 57.775427 57.775427
50 29 53.141321 53.141321 53.141321
50 30 53.225934 53.225934 53.225934
50 31 49.396356 49.396356 49.396356
50 32 51.429563 51.429563 51.429563
50 33 48.466483 48.466483 48.466483
50 34 44.922155 44.922155 44.922155
50 35 48.764741 48.764741 48.764741
50 36 77.162167 77.162167 77.162167
50 37 76.537572 76.537572 76.537572
50 38 73.681748 73.681748 73.681748
50 39 70.880181 70.880181 70.880181
50 40 70.342022 70.342022 70.342022
50 41 75.186435 75.186435 75.186435
50 42 65.000000 65.000000 65.000000
50 43 69.231496 69.231496 69.231496
50 44 74.148500 74.148500 74.148500
50 45 71.196910 71.196910 71.196910
50 46 73.375745 73.375745 73.375745
50 47 72.111026 72.111026 72.111026
50 48 51.855569 51.855569 51.855569
50 49 7.000000 7.000000 7.000000
50 51 37.802116 37.802116 37.802116
50 52 15.264338 15.264338 15.264338
50 53 24.758837 24.758837 24.758837
50 54 43.908997 43.908997 43.908997
50 55 49.729267 49.729267 49.729267
50 56 49.477268 49.477268 49.477268
50 57 24.351591 24.351591 24.351591
50 58 17.691806 17.691806 17.691806
50 59 27.073973 27.073973 27.073973
50 60 32.984845 32.984845 32.984845
50 61 55.072679 55.072679 55.072679
50 62 53.084838 53.084838 53.084838
50 63 32.526912 32.526912 32.526912
50 64 24.351591 24.351591 24.351591
50 65 18.248288 18.248288 18.248288
50 66 28.861739 28.861739 28.861739
50 67 25.019992 25.019992 25.019992
50 68 37.202150 37.202150 37.202150
50 69 48.041649 48.041649 48.041649
50 70 41.484937 41.484937 41.484937
50 71 57.428216 57.428216 57.428216
50 72 48.764741 48.764741 48.764741
50 73 57.008771 57.008771 57.008771
50 74 62.481997 62.481997 62.481997
50 75 23.409400 23.409400 23.409400

50 76 37.656341 37.656341 37.656341
50 77 18.000000 18.000000 18.000000
50 78 21.023796 21.023796 21.023796
50 79 55.605755 55.605755 55.605755
50 80 66.573268 66.573268 66.573268
50 81 35.355339 35.355339 35.355339
50 82 46.529560 46.529560 46.529560
50 83 34.438351 34.438351 34.438351
50 84 19.646883 19.646883 19.646883
50 85 22.671568 22.671568 22.671568
50 86 23.706539 23.706539 23.706539
50 87 24.186773 24.186773 24.186773
50 88 32.310989 32.310989 32.310989
50 89 49.396356 49.396356 49.396356
50 90 25.961510 25.961510 25.961510
50 91 35.355339 35.355339 35.355339
50 92 30.805844 30.805844 30.805844
50 93 32.893768 32.893768 32.893768
50 94 44.283180 44.283180 44.283180
50 95 39.000000 39.000000 39.000000
50 96 28.653098 28.653098 28.653098
50 97 43.965896 43.965896 43.965896
50 98 38.470768 38.470768 38.470768
50 99 43.081318 43.081318 43.081318
50 100 28.017851 28.017851 28.017851
50 101 56.089215 56.089215 56.089215
51 1 35.341194 35.341194 35.341194
51 2 68.622154 68.622154 68.622154
51 3 64.031242 64.031242 64.031242
51 4 70.710678 70.710678 70.710678
51 5 68.767725 68.767725 68.767725
51 6 72.138755 72.138755 72.138755
51 7 67.201190 67.201190 67.201190
51 8 69.634761 69.634761 69.634761
51 9 72.622311 72.622311 72.622311
51 10 62.000000 62.000000 62.000000
51 11 62.201286 62.201286 62.201286
51 12 64.195015 64.195015 64.195015
51 13 64.776539 64.776539 64.776539
51 14 67.000000 67.000000 67.000000
51 15 67.742158 67.742158 67.742158
51 16 70.178344 70.178344 70.178344
51 17 72.173402 72.173402 72.173402
51 18 72.691127 72.691127 72.691127
51 19 41.036569 41.036569 41.036569
51 20 39.051248 39.051248 39.051248
51 21 36.055513 36.055513 36.055513
51 22 43.863424 43.863424 43.863424
51 23 37.735925 37.735925 37.735925
51 24 45.343136 45.343136 45.343136
51 25 39.446166 39.446166 39.446166
51 26 47.634021 47.634021 47.634021
51 27 23.537205 23.537205 23.537205
51 28 23.000000 23.000000 23.000000
51 29 20.615528 20.615528 20.615528
51 30 18.000000 18.000000 18.000000
51 31 16.763055 16.763055 16.763055
51 32 16.000000 16.000000 16.000000
51 33 15.811388 15.811388 15.811388
51 34 16.401219 16.401219 16.401219
51 35 13.000000 13.000000 13.000000
51 36 50.249378 50.249378 50.249378
51 37 50.487622 50.487622 50.487622
51 38 47.518417 47.518417 47.518417
51 39 46.097722 46.097722 46.097722
51 40 46.572524 46.572524 46.572524
51 41 51.419841 51.419841 51.419841
51 42 42.379240 42.379240 42.379240
51 43 48.104054 48.104054 48.104054
51 44 52.810984 52.810984 52.810984
51 45 49.979996 49.979996 49.979996
51 46 70.092796 70.092796 70.092796
51 47 70.292247 70.292247 70.292247
51 48 70.710678 70.710678 70.710678
51 49 42.426407 42.426407 42.426407
51 50 37.802116 37.802116 37.802116
51 52 22.671568 22.671568 22.671568
51 53 47.265209 47.265209 47.265209
51 54 54.120237 54.120237 54.120237
51 55 30.232433 30.232433 30.232433
51 56 48.877398 48.877398 48.877398
51 57 22.000000 22.000000 22.000000
51 58 43.174066 43.174066 43.174066
51 59 62.241465 62.241465 62.241465
51 60 63.788714 63.788714 63.788714
51 61 62.241465 62.241465 62.241465
51 62 40.360872 40.360872 40.360872
51 63 7.000000 7.000000 7.000000
51 64 16.552945 16.552945 16.552945
51 65 27.459060 27.459060 27.459060
51 66 37.336309 37.336309 37.336309
51 67 31.064449 31.064449 31.064449
51 68 10.630146 10.630146 10.630146
51 69 40.607881 40.607881 40.607881
51 70 44.384682 44.384682 44.384682

51 71 50.249378 50.249378 50.249378
51 72 21.189620 21.189620 21.189620
51 73 31.320920 31.320920 31.320920
51 74 74.330344 74.330344 74.330344
51 75 54.120237 54.120237 54.120237
51 76 73.409809 73.409809 73.409809
51 77 25.942244 25.942244 25.942244
51 78 58.523500 58.523500 58.523500
51 79 67.357256 67.357256 67.357256
51 80 73.790243 73.790243 73.790243
51 81 27.730849 27.730849 27.730849
51 82 32.526912 32.526912 32.526912
51 83 45.705580 45.705580 45.705580
51 84 35.227830 35.227830 35.227830
51 85 16.155494 16.155494 16.155494
51 86 15.000000 15.000000 15.000000
51 87 52.172790 52.172790 52.172790
51 88 61.000000 61.000000 61.000000
51 89 53.225934 53.225934 53.225934
51 90 30.413813 30.413813 30.413813
51 91 37.000000 37.000000 37.000000
51 92 24.041631 24.041631 24.041631
51 93 20.615528 20.615528 20.615528
51 94 20.248457 20.248457 20.248457
51 95 19.849433 19.849433 19.849433
51 96 16.124515 16.124515 16.124515
51 97 25.495098 25.495098 25.495098
51 98 70.092796 70.092796 70.092796
51 99 49.040799 49.040799 49.040799
51 100 46.000000 46.000000 46.000000
51 101 52.009614 52.009614 52.009614
52 1 33.541020 33.541020 33.541020
52 2 71.589105 71.589105 71.589105
52 3 64.140471 64.140471 64.140471
52 4 72.897188 72.897188 72.897188
52 5 69.462220 69.462220 69.462220
52 6 73.824115 73.824115 73.824115
52 7 66.287254 66.287254 66.287254
52 8 68.007353 68.007353 68.007353
52 9 72.111026 72.111026 72.111026
52 10 47.434165 47.434165 47.434165
52 11 49.244289 49.244289 49.244289
52 12 51.078371 51.078371 51.078371
52 13 53.235327 53.235327 53.235327
52 14 52.201533 52.201533 52.201533
52 15 55.901699 55.901699 55.901699
52 16 56.648036 56.648036 56.648036
52 17 58.523500 58.523500 58.523500
52 18 60.415230 60.415230 60.415230
52 19 18.601075 18.601075 18.601075
52 20 16.401219 16.401219 16.401219
52 21 13.928388 13.928388 13.928388
52 22 21.213203 21.213203 21.213203
52 23 15.811388 15.811388 15.811388
52 24 22.671568 22.671568 22.671568
52 25 17.720045 17.720045 17.720045
52 26 25.000000 25.000000 25.000000
52 27 41.231056 41.231056 41.231056
52 28 42.720019 42.720019 42.720019
52 29 38.327536 38.327536 38.327536
52 30 38.078866 38.078866 38.078866
52 31 34.481879 34.481879 34.481879
52 32 36.249138 36.249138 36.249138
52 33 33.526109 33.526109 33.526109
52 34 30.413813 30.413813 30.413813
52 35 33.541020 33.541020 33.541020
52 36 66.098411 66.098411 66.098411
52 37 65.764732 65.764732 65.764732
52 38 62.801274 62.801274 62.801274
52 39 60.406953 60.406953 60.406953
52 40 60.207973 60.207973 60.207973
52 41 65.192024 65.192024 65.192024
52 42 55.081757 55.081757 55.081757
52 43 60.000000 60.000000 60.000000
52 44 65.000000 65.000000 65.000000
52 45 62.000000 62.000000 62.000000
52 46 71.196910 71.196910 71.196910
52 47 70.491134 70.491134 70.491134
52 48 58.600341 58.600341 58.600341
52 49 19.849433 19.849433 19.849433
52 50 15.264338 15.264338 15.264338
52 51 22.671568 22.671568 22.671568
52 53 31.622777 31.622777 31.622777
52 54 46.097722 46.097722 46.097722
52 55 40.000000 40.000000 40.000000
52 56 47.169906 47.169906 47.169906
52 57 15.811388 15.811388 15.811388
52 58 25.495098 25.495098 25.495098
52 59 41.231056 41.231056 41.231056
52 60 45.000000 45.000000 45.000000
52 61 56.568542 56.568542 56.568542
52 62 46.097722 46.097722 46.097722
52 63 18.027756 18.027756 18.027756
52 64 10.000000 10.000000 10.000000
52 65 14.142136 14.142136 14.142136

52 66 28.284271 28.284271 28.284271
52 67 22.022716 22.022716 22.022716
52 68 23.769729 23.769729 23.769729
52 69 42.720019 42.720019 42.720019
52 70 40.000000 40.000000 40.000000
52 71 52.924474 52.924474 52.924474
52 72 36.400549 36.400549 36.400549
52 73 45.705580 45.705580 45.705580
52 74 66.400301 66.400301 66.400301
52 75 35.000000 35.000000 35.000000
52 76 52.201533 52.201533 52.201533
52 77 9.433981 9.433981 9.433981
52 78 36.235342 36.235342 36.235342
52 79 59.203040 59.203040 59.203040
52 80 68.593003 68.593003 68.593003
52 81 28.160256 28.160256 28.160256
52 82 38.470768 38.470768 38.470768
52 83 36.235342 36.235342 36.235342
52 84 21.095023 21.095023 21.095023
52 85 9.219544 9.219544 9.219544
52 86 8.544004 8.544004 8.544004
52 87 34.234486 34.234486 34.234486
52 88 43.185646 43.185646 43.185646
52 89 49.040799 49.040799 49.040799
52 90 19.209373 19.209373 19.209373
52 91 32.449961 32.449961 32.449961
52 92 22.803509 22.803509 22.803509
52 93 23.086793 23.086793 23.086793
52 94 32.557641 32.557641 32.557641
52 95 28.071338 28.071338 28.071338
52 96 17.029386 17.029386 17.029386
52 97 34.000000 34.000000 34.000000
52 98 51.039201 51.039201 51.039201
52 99 43.185646 43.185646 43.185646
52 100 32.649655 32.649655 32.649655
52 101 52.773099 52.773099 52.773099
53 1 25.000000 25.000000 25.000000
53 2 55.000000 55.000000 55.000000
53 3 45.099889 45.099889 45.099889
53 4 55.081757 55.081757 55.081757
53 5 50.249378 50.249378 50.249378
53 6 55.226805 55.226805 55.226805
53 7 45.541190 45.541190 45.541190
53 8 46.097722 46.097722 46.097722
53 9 50.990195 50.990195 50.990195
53 10 15.811388 15.811388 15.811388
53 11 18.027756 18.027756 18.027756
53 12 19.723083 19.723083 19.723083
53 13 22.671568 22.671568 22.671568
53 14 20.615528 20.615528 20.615528
53 15 25.000000 25.000000 25.000000
53 16 25.079872 25.079872 25.079872
53 17 26.925824 26.925824 26.925824
53 18 29.154759 29.154759 29.154759
53 19 31.400637 31.400637 31.400637
53 20 26.248809 26.248809 26.248809
53 21 22.671568 22.671568 22.671568
53 22 29.154759 29.154759 29.154759
53 23 21.213203 21.213203 21.213203
53 24 28.178006 28.178006 28.178006
53 25 19.849433 19.849433 19.849433
53 26 26.925824 26.925824 26.925824
53 27 70.000000 70.000000 70.000000
53 28 70.178344 70.178344 70.178344
53 29 67.000000 67.000000 67.000000
53 30 65.192024 65.192024 65.192024
53 31 63.000000 63.000000 63.000000
53 32 63.198101 63.198101 63.198101
53 33 62.000000 62.000000 62.000000
53 34 60.207973 60.207973 60.207973
53 35 60.207973 60.207973 60.207973
53 36 69.202601 69.202601 69.202601
53 37 68.007353 68.007353 68.007353
53 38 65.604878 65.604878 65.604878
53 39 62.201286 62.201286 62.201286
53 40 61.032778 61.032778 61.032778
53 41 65.192024 65.192024 65.192024
53 42 55.803226 55.803226 55.803226
53 43 58.309519 58.309519 58.309519
53 44 62.649820 62.649820 62.649820
53 45 60.033324 60.033324 60.033324
53 46 52.239832 52.239832 52.239832
53 47 50.487622 50.487622 50.487622
53 48 27.459060 27.459060 27.459060
53 49 30.232433 30.232433 30.232433
53 50 24.758837 24.758837 24.758837
53 51 47.265209 47.265209 47.265209
53 52 31.622777 31.622777 31.622777
53 54 20.615528 20.615528 20.615528
53 55 42.426407 42.426407 42.426407
53 56 30.413813 30.413813 30.413813
53 57 25.495098 25.495098 25.495098
53 58 7.071068 7.071068 7.071068
53 59 22.360680 22.360680 22.360680
53 60 18.027756 18.027756 18.027756

53 61 31.622777 31.622777 31.622777
53 62 40.311289 40.311289 40.311289
53 63 40.311289 40.311289 40.311289
53 64 41.231056 41.231056 41.231056
53 65 20.000000 20.000000 20.000000
53 66 14.142136 14.142136 14.142136
53 67 17.464249 17.464249 17.464249
53 68 40.804412 40.804412 40.804412
53 69 33.541020 33.541020 33.541020
53 70 22.803509 22.803509 22.803509
53 71 40.261644 40.261644 40.261644
53 72 47.169906 47.169906 47.169906
53 73 51.662365 51.662365 51.662365
53 74 37.802116 37.802116 37.802116
53 75 11.180340 11.180340 11.180340
53 76 32.015621 32.015621 32.015621
53 77 39.357337 39.357337 39.357337
53 78 27.073973 27.073973 27.073973
53 79 31.064449 31.064449 31.064449
53 80 42.485292 42.485292 42.485292
53 81 27.802878 27.802878 27.802878
53 82 36.878178 36.878178 36.878178
53 83 13.152946 13.152946 13.152946
53 84 12.041595 12.041595 12.041595
53 85 32.015621 32.015621 32.015621
53 86 38.639358 38.639358 38.639358
53 87 7.211103 7.211103 7.211103
53 88 14.317821 14.317821 14.317821
53 89 28.017851 28.017851 28.017851
53 90 48.877398 48.877398 48.877398
53 91 20.808652 20.808652 20.808652
53 92 26.832816 26.832816 26.832816
53 93 30.870698 30.870698 30.870698
53 94 42.190046 42.190046 42.190046
53 95 36.715120 36.715120 36.715120
53 96 31.780497 31.780497 31.780497
53 97 38.418745 38.418745 38.418745
53 98 24.186773 24.186773 24.186773
53 99 22.022716 22.022716 22.022716
53 100 5.099020 5.099020 5.099020
53 101 37.483330 37.483330 37.483330
54 1 20.000000 20.000000 20.000000
54 2 35.355339 35.355339 35.355339
54 3 25.079872 25.079872 25.079872
54 4 35.057096 35.057096 35.057096
54 5 30.000000 30.000000 30.000000
54 6 35.000000 35.000000 35.000000
54 7 25.079872 25.079872 25.079872
54 8 25.495098 25.495098 25.495098
54 9 30.413813 30.413813 30.413813
54 10 18.027756 18.027756 18.027756
54 11 14.142136 14.142136 14.142136
54 12 15.620499 15.620499 15.620499
54 13 13.000000 13.000000 13.000000
54 14 21.213203 21.213203 21.213203
54 15 15.811388 15.811388 15.811388
54 16 20.591260 20.591260 20.591260
54 17 22.360680 22.360680 22.360680
54 18 20.615528 20.615528 20.615528
54 19 51.000000 51.000000 51.000000
54 20 45.650849 45.650849 45.650849
54 21 41.340053 41.340053 41.340053
54 22 49.244289 49.244289 49.244289
54 23 40.311289 40.311289 40.311289
54 24 48.466483 48.466483 48.466483
54 25 39.357337 39.357337 39.357337
54 26 47.434165 47.434165 47.434165
54 27 77.620873 77.620873 77.620873
54 28 76.485293 76.485293 76.485293
54 29 74.726167 74.726167 74.726167
54 30 71.589105 71.589105 71.589105
54 31 70.880181 70.880181 70.880181
54 32 69.634761 69.634761 69.634761
54 33 69.921384 69.921384 69.921384
54 34 69.641941 69.641941 69.641941
54 35 66.708320 66.708320 66.708320
54 36 58.600341 58.600341 58.600341
54 37 57.008771 57.008771 57.008771
54 38 55.217751 55.217751 55.217751
54 39 51.613952 51.613952 51.613952
54 40 50.000000 50.000000 50.000000
54 41 53.150729 53.150729 53.150729
54 42 45.486262 45.486262 45.486262
54 43 46.097722 46.097722 46.097722
54 44 49.497475 49.497475 49.497475
54 45 47.423623 47.423623 47.423623
54 46 32.000000 32.000000 32.000000
54 47 30.066593 30.066593 30.066593
54 48 18.681542 18.681542 18.681542
54 49 50.089919 50.089919 50.089919
54 50 43.908997 43.908997 43.908997
54 51 54.120237 54.120237 54.120237
54 52 46.097722 46.097722 46.097722
54 53 20.615528 20.615528 20.615528
54 55 36.400549 36.400549 36.400549

54 56 14.142136 14.142136 14.142136
54 57 33.541020 33.541020 33.541020
54 58 26.925824 26.925824 26.925824
54 59 40.311289 40.311289 40.311289
54 60 31.622777 31.622777 31.622777
54 61 11.180340 11.180340 11.180340
54 62 29.154759 29.154759 29.154759
54 63 47.434165 47.434165 47.434165
54 64 54.083269 54.083269 54.083269
54 65 32.015621 32.015621 32.015621
54 66 18.027756 18.027756 18.027756
54 67 24.698178 24.698178 24.698178
54 68 44.721360 44.721360 44.721360
54 69 22.360680 22.360680 22.360680
54 70 11.180340 11.180340 11.180340
54 71 24.207437 24.207437 24.207437
54 72 45.276926 45.276926 45.276926
54 73 45.541190 45.541190 45.541190
54 74 20.591260 20.591260 20.591260
54 75 30.000000 30.000000 30.000000
54 76 47.434165 47.434165 47.434165
54 77 55.172457 55.172457 55.172457
54 78 47.095647 47.095647 47.095647
54 79 13.416408 13.416408 13.416408
54 80 22.803509 22.803509 22.803509
54 81 27.166155 27.166155 27.166155
54 82 30.083218 30.083218 30.083218
54 83 9.899495 9.899495 9.899495
54 84 25.495098 25.495098 25.495098
54 85 42.544095 42.544095 42.544095
54 86 50.774009 50.774009 50.774009
54 87 26.019224 26.019224 26.019224
54 88 27.202941 27.202941 27.202941
54 89 8.944272 8.944272 8.944272
54 90 65.069194 65.069194 65.069194
54 91 17.262677 17.262677 17.262677
54 92 30.083218 30.083218 30.083218
54 93 33.734256 33.734256 33.734256
54 94 41.048752 41.048752 41.048752
54 95 37.054015 37.054015 37.054015
54 96 38.275318 38.275318 38.275318
54 97 35.227830 35.227830 35.227830
54 98 35.777088 35.777088 35.777088
54 99 6.324555 6.324555 6.324555
54 100 16.155494 16.155494 16.155494
54 101 20.248457 20.248457 20.248457
55 1 18.027756 18.027756 18.027756
55 2 39.051248 39.051248 39.051248
55 3 36.249138 36.249138 36.249138
55 4 41.400483 41.400483 41.400483
55 5 40.311289 40.311289 40.311289
55 6 43.011626 43.011626 43.011626
55 7 39.924930 39.924930 39.924930
55 8 42.720019 42.720019 42.720019
55 9 44.721360 44.721360 44.721360
55 10 51.478151 51.478151 51.478151
55 11 49.244289 49.244289 49.244289
55 12 51.078371 51.078371 51.078371
55 13 49.335586 49.335586 49.335586
55 14 55.901699 55.901699 55.901699
55 15 52.201533 52.201533 52.201533
55 16 56.648036 56.648036 56.648036
55 17 58.523500 58.523500 58.523500
55 18 57.008771 57.008771 57.008771
55 19 56.089215 56.089215 56.089215
55 20 51.662365 51.662365 51.662365
55 21 46.840154 46.840154 46.840154
55 22 57.008771 57.008771 57.008771
55 23 47.434165 47.434165 47.434165
55 24 57.567352 57.567352 57.567352
55 25 48.104054 48.104054 48.104054
55 26 58.523500 58.523500 58.523500
55 27 50.000000 50.000000 50.000000
55 28 47.169906 47.169906 47.169906
55 29 47.634021 47.634021 47.634021
55 30 43.011626 43.011626 43.011626
55 31 44.598206 44.598206 44.598206
55 32 41.400483 41.400483 41.400483
55 33 43.863424 43.863424 43.863424
55 34 46.097722 46.097722 46.097722
55 35 39.051248 39.051248 39.051248
55 36 27.730849 27.730849 27.730849
55 37 26.925824 26.925824 26.925824
55 38 24.166092 24.166092 24.166092
55 39 21.189620 21.189620 21.189620
55 40 20.615528 20.615528 20.615528
55 41 25.495098 25.495098 25.495098
55 42 15.297059 15.297059 15.297059
55 43 20.000000 20.000000 20.000000
55 44 25.000000 25.000000 25.000000
55 45 22.000000 22.000000 22.000000
55 46 41.340053 41.340053 41.340053
55 47 42.059482 42.059482 42.059482
55 48 55.081757 55.081757 55.081757
55 49 56.515485 56.515485 56.515485

55 50 49.729267 49.729267 49.729267
55 51 30.232433 30.232433 30.232433
55 52 40.000000 40.000000 40.000000
55 53 42.426407 42.426407 42.426407
55 54 36.400549 36.400549 36.400549
55 56 25.000000 25.000000 25.000000
55 57 25.495098 25.495098 25.495098
55 58 43.011626 43.011626 43.011626
55 59 64.031242 64.031242 64.031242
55 60 60.207973 60.207973 60.207973
55 61 40.000000 40.000000 40.000000
55 62 11.180340 11.180340 11.180340
55 63 26.925824 26.925824 26.925824
55 64 41.231056 41.231056 41.231056
55 65 31.622777 31.622777 31.622777
55 66 28.284271 28.284271 28.284271
55 67 26.925824 26.925824 26.925824
55 68 20.124612 20.124612 20.124612
55 69 15.000000 15.000000 15.000000
55 70 25.298221 25.298221 25.298221
55 71 21.931712 21.931712 21.931712
55 72 11.180340 11.180340 11.180340
55 73 9.433981 9.433981 9.433981
55 74 53.000000 53.000000 53.000000
55 75 53.150729 53.150729 53.150729
55 76 74.330344 74.330344 74.330344
55 77 48.259714 48.259714 48.259714
55 78 65.368188 65.368188 65.368188
55 79 47.169906 47.169906 47.169906
55 80 49.648766 49.648766 49.648766
55 81 15.264338 15.264338 15.264338
55 82 6.324555 6.324555 6.324555
55 83 32.756679 32.756679 32.756679
55 84 34.132096 34.132096 34.132096
55 85 31.064449 31.064449 31.064449
55 86 37.854986 37.854986 37.854986
55 87 49.517674 49.517674 49.517674
55 88 56.080300 56.080300 56.080300
55 89 31.064449 31.064449 31.064449
55 90 56.293872 56.293872 56.293872
55 91 22.203603 22.203603 22.203603
55 92 18.973666 18.973666 18.973666
55 93 17.117243 17.117243 17.117243
55 94 10.000000 10.000000 10.000000
55 95 12.165525 12.165525 12.165525
55 96 23.021729 23.021729 23.021729
55 97 6.000000 6.000000 6.000000
55 98 66.068147 66.068147 66.068147
55 99 30.083218 30.083218 30.083218
55 100 38.288379 38.288379 38.288379
55 101 25.000000 25.000000 25.000000
56 1 14.142136 14.142136 14.142136
56 2 25.495098 25.495098 25.495098
56 3 17.000000 17.000000 17.000000
56 4 26.248809 26.248809 26.248809
56 5 22.360680 22.360680 22.360680
56 6 26.925824 26.925824 26.925824
56 7 19.209373 19.209373 19.209373
56 8 21.213203 21.213203 21.213203
56 9 25.000000 25.000000 25.000000
56 10 32.015621 32.015621 32.015621
56 11 28.284271 28.284271 28.284271
56 12 29.732137 29.732137 29.732137
56 13 26.627054 26.627054 26.627054
56 14 35.355339 35.355339 35.355339
56 15 29.154759 29.154759 29.154759
56 16 34.409301 34.409301 34.409301
56 17 36.055513 36.055513 36.055513
56 18 33.541020 33.541020 33.541020
56 19 56.753854 56.753854 56.753854
56 20 51.419841 51.419841 51.419841
56 21 46.572524 46.572524 46.572524
56 22 55.901699 55.901699 55.901699
56 23 46.097722 46.097722 46.097722
56 24 55.578773 55.578773 55.578773
56 25 45.705580 45.705580 45.705580
56 26 55.226805 55.226805 55.226805
56 27 71.589105 71.589105 71.589105
56 28 69.641941 69.641941 69.641941
56 29 68.876701 68.876701 68.876701
56 30 65.000000 65.000000 65.000000
56 31 65.299311 65.299311 65.299311
56 32 63.158531 63.158531 63.158531
56 33 64.412732 64.412732 64.412732
56 34 65.192024 65.192024 65.192024
56 35 60.415230 60.415230 60.415230
56 36 44.654227 44.654227 44.654227
56 37 43.011626 43.011626 43.011626
56 38 41.340053 41.340053 41.340053
56 39 37.735925 37.735925 37.735925
56 40 36.055513 36.055513 36.055513
56 41 39.051248 39.051248 39.051248
56 42 31.764760 31.764760 31.764760
56 43 32.015621 32.015621 32.015621
56 44 35.355339 35.355339 35.355339

56 45 33.301652 33.301652 33.301652
56 46 24.166092 24.166092 24.166092
56 47 23.323808 23.323808 23.323808
56 48 31.764760 31.764760 31.764760
56 49 56.293872 56.293872 56.293872
56 50 49.477268 49.477268 49.477268
56 51 48.877398 48.877398 48.877398
56 52 47.169906 47.169906 47.169906
56 53 30.413813 30.413813 30.413813
56 54 14.142136 14.142136 14.142136
56 55 25.000000 25.000000 25.000000
56 57 32.015621 32.015621 32.015621
56 58 35.000000 35.000000 35.000000
56 59 52.201533 52.201533 52.201533
56 60 44.721360 44.721360 44.721360
56 61 15.000000 15.000000 15.000000
56 62 15.811388 15.811388 15.811388
56 63 43.011626 43.011626 43.011626
56 64 53.150729 53.150729 53.150729
56 65 33.541020 33.541020 33.541020
56 66 20.615528 20.615528 20.615528
56 67 25.495098 25.495098 25.495098
56 68 38.470768 38.470768 38.470768
56 69 10.000000 10.000000 10.000000
56 70 8.062258 8.062258 8.062258
56 71 10.295630 10.295630 10.295630
56 72 35.355339 35.355339 35.355339
56 73 33.376639 33.376639 33.376639
56 74 28.000000 28.000000 28.000000
56 75 41.231056 41.231056 41.231056
56 76 60.415230 60.415230 60.415230
56 77 56.603887 56.603887 56.603887
56 78 57.428216 57.428216 57.428216
56 79 22.360680 22.360680 22.360680
56 80 25.298221 25.298221 25.298221
56 81 21.400935 21.400935 21.400935
56 82 19.104973 19.104973 19.104973
56 83 17.262677 17.262677 17.262677
56 84 29.832868 29.832868 29.832868
56 85 41.109610 41.109610 41.109610
56 86 49.578221 49.578221 49.578221
56 87 37.107951 37.107951 37.107951
56 88 40.249224 40.249224 40.249224
56 89 6.324555 6.324555 6.324555
56 90 66.287254 66.287254 66.287254
56 91 14.764823 14.764823 14.764823
56 92 26.172505 26.172505 26.172505
56 93 28.600699 28.600699 28.600699
56 94 32.015621 32.015621 32.015621
56 95 29.546573 29.546573 29.546573
56 96 34.713110 34.713110 34.713110
56 97 25.709920 25.709920 25.709920
56 98 49.396356 49.396356 49.396356
56 99 8.944272 8.944272 8.944272
56 100 25.317978 25.317978 25.317978
56 101 7.071068 7.071068 7.071068
57 1 18.027756 18.027756 18.027756
57 2 55.901699 55.901699 55.901699
57 3 48.826222 48.826222 48.826222
57 4 57.306195 57.306195 57.306195
57 5 54.083269 54.083269 54.083269
57 6 58.309519 58.309519 58.309519
57 7 51.224994 51.224994 51.224994
57 8 53.150729 53.150729 53.150729
57 9 57.008771 57.008771 57.008771
57 10 40.000000 40.000000 40.000000
57 11 40.311289 40.311289 40.311289
57 12 42.296572 42.296572 42.296572
57 13 43.174066 43.174066 43.174066
57 14 45.000000 45.000000 45.000000
57 15 46.097722 46.097722 46.097722
57 16 48.259714 48.259714 48.259714
57 17 50.249378 50.249378 50.249378
57 18 50.990195 50.990195 50.990195
57 19 30.594117 30.594117 30.594117
57 20 26.248809 26.248809 26.248809
57 21 21.540659 21.540659 21.540659
57 22 31.622777 31.622777 31.622777
57 23 22.360680 22.360680 22.360680
57 24 32.310989 32.310989 32.310989
57 25 23.323808 23.323808 23.323808
57 26 33.541020 33.541020 33.541020
57 27 45.276926 45.276926 45.276926
57 28 45.000000 45.000000 45.000000
57 29 42.296572 42.296572 42.296572
57 30 40.000000 40.000000 40.000000
57 31 38.327536 38.327536 38.327536
57 32 38.000000 38.000000 38.000000
57 33 37.336309 37.336309 37.336309
57 34 36.400549 36.400549 36.400549
57 35 35.000000 35.000000 35.000000
57 36 52.810984 52.810984 52.810984
57 37 52.201533 52.201533 52.201533
57 38 49.335586 49.335586 49.335586
57 39 46.572524 46.572524 46.572524

57 40 46.097722 46.097722 46.097722
57 41 50.990195 50.990195 50.990195
57 42 40.792156 40.792156 40.792156
57 43 45.276926 45.276926 45.276926
57 44 50.249378 50.249378 50.249378
57 45 47.265209 47.265209 47.265209
57 46 55.758407 55.758407 55.758407
57 47 55.217751 55.217751 55.217751
57 48 49.030603 49.030603 49.030603
57 49 31.048349 31.048349 31.048349
57 50 24.351591 24.351591 24.351591
57 51 22.000000 22.000000 22.000000
57 52 15.811388 15.811388 15.811388
57 53 25.495098 25.495098 25.495098
57 54 33.541020 33.541020 33.541020
57 55 25.495098 25.495098 25.495098
57 56 32.015621 32.015621 32.015621
57 58 22.360680 22.360680 22.360680
57 59 43.011626 43.011626 43.011626
57 60 42.720019 42.720019 42.720019
57 61 43.011626 43.011626 43.011626
57 62 30.413813 30.413813 30.413813
57 63 15.000000 15.000000 15.000000
57 64 21.213203 21.213203 21.213203
57 65 7.071068 7.071068 7.071068
57 66 15.811388 15.811388 15.811388
57 67 9.219544 9.219544 9.219544
57 68 15.652476 15.652476 15.652476
57 69 26.925824 26.925824 26.925824
57 70 25.495098 25.495098 25.495098
57 71 37.161808 37.161808 37.161808
57 72 25.000000 25.000000 25.000000
57 73 32.695565 32.695565 32.695565
57 74 54.120237 54.120237 54.120237
57 75 33.541020 33.541020 33.541020
57 76 54.083269 54.083269 54.083269
57 77 25.079872 25.079872 25.079872
57 78 41.868843 41.868843 41.868843
57 79 46.957428 46.957428 46.957428
57 80 55.000000 55.000000 55.000000
57 81 12.369317 12.369317 12.369317
57 82 23.021729 23.021729 23.021729
57 83 24.351591 24.351591 24.351591
57 84 13.601471 13.601471 13.601471
57 85 9.219544 9.219544 9.219544
57 86 17.691806 17.691806 17.691806
57 87 31.016125 31.016125 31.016125
57 88 39.560081 39.560081 39.560081
57 89 34.713110 34.713110 34.713110
57 90 34.481879 34.481879 34.481879
57 91 17.691806 17.691806 17.691806
57 92 7.071068 7.071068 7.071068
57 93 8.544004 8.544004 8.544004
57 94 20.248457 20.248457 20.248457
57 95 14.764823 14.764823 14.764823
57 96 6.324555 6.324555 6.324555
57 97 19.646883 19.646883 19.646883
57 98 49.040799 49.040799 49.040799
57 99 29.410882 29.410882 29.410882
57 100 24.000000 24.000000 24.000000
57 101 37.215588 37.215588 37.215588
58 1 26.925824 26.925824 26.925824
58 2 60.207973 60.207973 60.207973
58 3 50.635956 50.635956 50.635956
58 4 60.530984 60.530984 60.530984
58 5 55.901699 55.901699 55.901699
58 6 60.827625 60.827625 60.827625
58 7 51.419841 51.419841 51.419841
58 8 52.201533 52.201533 52.201533
58 9 57.008771 57.008771 57.008771
58 10 22.360680 22.360680 22.360680
58 11 25.000000 25.000000 25.000000
58 12 26.627054 26.627054 26.627054
58 13 29.732137 29.732137 29.732137
58 14 26.925824 26.925824 26.925824
58 15 32.015621 32.015621 32.015621
58 16 31.764760 31.764760 31.764760
58 17 33.541020 33.541020 33.541020
58 18 36.055513 36.055513 36.055513
58 19 24.413111 24.413111 24.413111
58 20 19.209373 19.209373 19.209373
58 21 15.620499 15.620499 15.620499
58 22 22.360680 22.360680 22.360680
58 23 14.142136 14.142136 14.142136
58 24 21.540659 21.540659 21.540659
58 25 12.806248 12.806248 12.806248
58 26 20.615528 20.615528 20.615528
58 27 65.192024 65.192024 65.192024
58 28 65.764732 65.764732 65.764732
58 29 62.201286 62.201286 62.201286
58 30 60.827625 60.827625 60.827625
58 31 58.215118 58.215118 58.215118
58 32 58.855756 58.855756 58.855756
58 33 57.218878 57.218878 57.218878
58 34 55.000000 55.000000 55.000000

58 35 55.901699 55.901699 55.901699
58 36 70.491134 70.491134 70.491134
58 37 69.462220 69.462220 69.462220
58 38 66.887966 66.887966 66.887966
58 39 63.631753 63.631753 63.631753
58 40 62.649820 62.649820 62.649820
58 41 67.082039 67.082039 67.082039
58 42 57.306195 57.306195 57.306195
58 43 60.415230 60.415230 60.415230
58 44 65.000000 65.000000 65.000000
58 45 62.241465 62.241465 62.241465
58 46 57.870545 57.870545 57.870545
58 47 56.293872 56.293872 56.293872
58 48 34.409301 34.409301 34.409301
58 49 23.323808 23.323808 23.323808
58 50 17.691806 17.691806 17.691806
58 51 43.174066 43.174066 43.174066
58 52 25.495098 25.495098 25.495098
58 53 7.071068 7.071068 7.071068
58 54 26.925824 26.925824 26.925824
58 55 43.011626 43.011626 43.011626
58 56 35.000000 35.000000 35.000000
58 57 22.360680 22.360680 22.360680
58 59 21.213203 21.213203 21.213203
58 60 20.615528 20.615528 20.615528
58 61 38.078866 38.078866 38.078866
58 62 42.720019 42.720019 42.720019
58 63 36.400549 36.400549 36.400549
58 64 35.355339 35.355339 35.355339
58 65 15.811388 15.811388 15.811388
58 66 15.811388 15.811388 15.811388
58 67 16.278821 16.278821 16.278821
58 68 38.013156 38.013156 38.013156
58 69 36.400549 36.400549 36.400549
58 70 27.018512 27.018512 27.018512
58 71 44.283180 44.283180 44.283180
58 72 46.097722 46.097722 46.097722
58 73 51.855569 51.855569 51.855569
58 74 44.821870 44.821870 44.821870
58 75 11.180340 11.180340 11.180340
58 76 32.015621 32.015621 32.015621
58 77 32.695565 32.695565 32.695565
58 78 23.086793 23.086793 23.086793
58 79 38.013156 38.013156 38.013156
58 80 49.244289 49.244289 49.244289
58 81 27.802878 27.802878 27.802878
58 82 38.078866 38.078866 38.078866
58 83 18.248288 18.248288 18.248288
58 84 9.219544 9.219544 9.219544
58 85 27.294688 27.294688 27.294688
58 86 33.060551 33.060551 33.060551
58 87 9.055385 9.055385 9.055385
58 88 18.027756 18.027756 18.027756
58 89 33.541020 33.541020 33.541020
58 90 42.059482 42.059482 42.059482
58 91 23.086793 23.086793 23.086793
58 92 25.495098 25.495098 25.495098
58 93 29.206164 29.206164 29.206164
58 94 41.109610 41.109610 41.109610
58 95 35.468296 35.468296 35.468296
58 96 28.635642 28.635642 28.635642
58 97 38.288379 38.288379 38.288379
58 98 26.925824 26.925824 26.925824
58 99 27.294688 27.294688 27.294688
58 100 10.770330 10.770330 10.770330
58 101 42.011903 42.011903 42.011903
59 1 47.169906 47.169906 47.169906
59 2 75.663730 75.663730 75.663730
59 3 65.375837 65.375837 65.375837
59 4 75.325958 75.325958 75.325958
59 5 70.178344 70.178344 70.178344
59 6 75.166482 75.166482 75.166482
59 7 65.069194 65.069194 65.069194
59 8 65.000000 65.000000 65.000000
59 9 70.000000 70.000000 70.000000
59 10 25.495098 25.495098 25.495098
59 11 30.413813 30.413813 30.413813
59 12 30.805844 30.805844 30.805844
59 13 35.693137 35.693137 35.693137
59 14 26.925824 26.925824 26.925824
59 15 36.400549 36.400549 36.400549
59 16 32.695565 32.695565 32.695565
59 17 33.541020 33.541020 33.541020
59 18 38.078866 38.078866 38.078866
59 19 29.427878 29.427878 29.427878
59 20 27.000000 27.000000 27.000000
59 21 27.459060 27.459060 27.459060
59 22 25.495098 25.495098 25.495098
59 23 25.495098 25.495098 25.495098
59 24 23.537205 23.537205 23.537205
59 25 23.537205 23.537205 23.537205
59 26 20.615528 20.615528 20.615528
59 27 82.462113 82.462113 82.462113
59 28 83.815273 83.815273 83.815273
59 29 79.555012 79.555012 79.555012

59 30 79.056942 79.056942 79.056942
59 31 75.690158 75.690158 75.690158
59 32 77.162167 77.162167 77.162167
59 33 74.726167 74.726167 74.726167
59 34 71.589105 71.589105 71.589105
59 35 74.330344 74.330344 74.330344
59 36 91.263355 91.263355 91.263355
59 37 90.138782 90.138782 90.138782
59 38 87.658428 87.658428 87.658428
59 39 84.314886 84.314886 84.314886
59 40 83.216585 83.216585 83.216585
59 41 87.464278 87.464278 87.464278
59 42 77.935871 77.935871 77.935871
59 43 80.622577 80.622577 80.622577
59 44 85.000000 85.000000 85.000000
59 45 82.365041 82.365041 82.365041
59 46 72.173402 72.173402 72.173402
59 47 70.064256 70.064256 70.064256
59 48 37.336309 37.336309 37.336309
59 49 27.459060 27.459060 27.459060
59 50 27.073973 27.073973 27.073973
59 51 62.241465 62.241465 62.241465
59 52 41.231056 41.231056 41.231056
59 53 22.360680 22.360680 22.360680
59 54 40.311289 40.311289 40.311289
59 55 64.031242 64.031242 64.031242
59 56 52.201533 52.201533 52.201533
59 57 43.011626 43.011626 43.011626
59 58 21.213203 21.213203 21.213203
59 60 11.180340 11.180340 11.180340
59 61 50.000000 50.000000 50.000000
59 62 62.649820 62.649820 62.649820
59 63 55.901699 55.901699 55.901699
59 64 50.990195 50.990195 50.990195
59 65 36.055513 36.055513 36.055513
59 66 36.055513 36.055513 36.055513
59 67 37.483330 37.483330 37.483330
59 68 58.523500 58.523500 58.523500
59 69 55.901699 55.901699 55.901699
59 70 44.944410 44.944410 44.944410
59 71 62.297673 62.297673 62.297673
59 72 67.268120 67.268120 67.268120
59 73 73.000000 73.000000 73.000000
59 74 51.662365 51.662365 51.662365
59 75 11.180340 11.180340 11.180340
59 76 11.180340 11.180340 11.180340
59 77 45.044423 45.044423 45.044423
59 78 10.630146 10.630146 10.630146
59 79 46.529560 46.529560 46.529560
59 80 58.694122 58.694122 58.694122
59 81 48.918299 48.918299 48.918299
59 82 58.821765 58.821765 58.821765
59 83 35.114100 35.114100 35.114100
59 84 30.413813 30.413813 30.413813
59 85 46.097722 46.097722 46.097722
59 86 49.729267 49.729267 49.729267
59 87 15.231546 15.231546 15.231546
59 88 14.317821 14.317821 14.317821
59 89 48.836462 48.836462 48.836462
59 90 52.239832 52.239832 52.239832
59 91 43.046487 43.046487 43.046487
59 92 46.690470 46.690470 46.690470
59 93 50.328918 50.328918 50.328918
59 94 62.289646 62.289646 62.289646
59 95 56.639209 56.639209 56.639209
59 96 49.091751 49.091751 49.091751
59 97 59.464275 59.464275 59.464275
59 98 13.601471 13.601471 13.601471
59 99 43.416587 43.416587 43.416587
59 100 27.313001 27.313001 27.313001
59 101 59.203040 59.203040 59.203040
60 1 42.426407 42.426407 42.426407
60 2 66.708320 66.708320 66.708320
60 3 56.293872 56.293872 56.293872
60 4 66.098411 66.098411 66.098411
60 5 60.827625 60.827625 60.827625
60 6 65.764732 65.764732 65.764732
60 7 55.578773 55.578773 55.578773
60 8 55.226805 55.226805 55.226805
60 9 60.207973 60.207973 60.207973
60 10 15.000000 15.000000 15.000000
60 11 20.000000 20.000000 20.000000
60 12 20.099751 20.099751 20.099751
60 13 25.079872 25.079872 25.079872
60 14 15.811388 15.811388 15.811388
60 15 25.495098 25.495098 25.495098
60 16 21.540659 21.540659 21.540659
60 17 22.360680 22.360680 22.360680
60 18 26.925824 26.925824 26.925824
60 19 37.161808 37.161808 37.161808
60 20 33.526109 33.526109 33.526109
60 21 32.388269 32.388269 32.388269
60 22 33.541020 33.541020 33.541020
60 23 30.413813 30.413813 30.413813
60 24 31.764760 31.764760 31.764760

60 25 28.442925 28.442925 28.442925
60 26 29.154759 29.154759 29.154759
60 27 85.586214 85.586214 85.586214
60 28 86.313383 86.313383 86.313383
60 29 82.607506 82.607506 82.607506
60 30 81.394103 81.394103 81.394103
60 31 78.638413 78.638413 78.638413
60 32 79.429214 79.429214 79.429214
60 33 77.646635 77.646635 77.646635
60 34 75.166482 75.166482 75.166482
60 35 76.485293 76.485293 76.485293
60 36 86.452299 86.452299 86.452299
60 37 85.146932 85.146932 85.146932
60 38 82.879430 82.879430 82.879430
60 39 79.397733 79.397733 79.397733
60 40 78.102497 78.102497 78.102497
60 41 82.006097 82.006097 82.006097
60 42 73.000000 73.000000 73.000000
60 43 75.000000 75.000000 75.000000
60 44 79.056942 79.056942 79.056942
60 45 76.609399 76.609399 76.609399
60 46 62.801274 62.801274 62.801274
60 47 60.530984 60.530984 60.530984
60 48 26.248809 26.248809 26.248809
60 49 35.341194 35.341194 35.341194
60 50 32.984845 32.984845 32.984845
60 51 63.788714 63.788714 63.788714
60 52 45.000000 45.000000 45.000000
60 53 18.027756 18.027756 18.027756
60 54 31.622777 31.622777 31.622777
60 55 60.207973 60.207973 60.207973
60 56 44.721360 44.721360 44.721360
60 57 42.720019 42.720019 42.720019
60 58 20.615528 20.615528 20.615528
60 59 11.180340 11.180340 11.180340
60 61 40.311289 40.311289 40.311289
60 62 57.008771 57.008771 57.008771
60 63 57.008771 57.008771 57.008771
60 64 55.000000 55.000000 55.000000
60 65 36.400549 36.400549 36.400549
60 66 32.015621 32.015621 32.015621
60 67 35.355339 35.355339 35.355339
60 68 58.309519 58.309519 58.309519
60 69 50.000000 50.000000 50.000000
60 70 38.275318 38.275318 38.275318
60 71 55.009090 55.009090 55.009090
60 72 65.192024 65.192024 65.192024
60 73 69.526973 69.526973 69.526973
60 74 40.792156 40.792156 40.792156
60 75 10.000000 10.000000 10.000000
60 76 15.811388 15.811388 15.811388
60 77 50.635956 50.635956 50.635956
60 78 21.400935 21.400935 21.400935
60 79 36.055513 36.055513 36.055513
60 80 48.166378 48.166378 48.166378
60 81 45.803930 45.803930 45.803930
60 82 54.451814 54.451814 54.451814
60 83 28.600699 28.600699 28.600699
60 84 29.154759 29.154759 29.154759
60 85 47.853944 47.853944 47.853944
60 86 53.084838 53.084838 53.084838
60 87 11.704700 11.704700 11.704700
60 88 4.472136 4.472136 4.472136
60 89 40.496913 40.496913 40.496913
60 90 58.940648 58.940648 58.940648
60 91 38.183766 38.183766 38.183766
60 92 44.777226 44.777226 44.777226
60 93 48.764741 48.764741 48.764741
60 94 60.207973 60.207973 60.207973
60 95 54.708317 54.708317 54.708317
60 96 49.040799 49.040799 49.040799
60 97 56.400355 56.400355 56.400355
60 98 6.324555 6.324555 6.324555
60 99 35.777088 35.777088 35.777088
60 100 21.931712 21.931712 21.931712
60 101 51.478151 51.478151 51.478151
61 1 26.925824 26.925824 26.925824
61 2 26.925824 26.925824 26.925824
61 3 16.552945 16.552945 16.552945
61 4 25.961510 25.961510 25.961510
61 5 20.615528 20.615528 20.615528
61 6 25.495098 25.495098 25.495098
61 7 15.297059 15.297059 15.297059
61 8 15.000000 15.000000 15.000000
61 9 20.000000 20.000000 20.000000
61 10 25.495098 25.495098 25.495098
61 11 20.615528 20.615528 20.615528
61 12 21.189620 21.189620 21.189620
61 13 16.552945 16.552945 16.552945
61 14 26.925824 26.925824 26.925824
61 15 18.027756 18.027756 18.027756
61 16 23.853721 23.853721 23.853721
61 17 25.000000 25.000000 25.000000
61 18 21.213203 21.213203 21.213203
61 19 62.177166 62.177166 62.177166
61 20 56.824291 56.824291 56.824291

61 20 56.824291 56.824291 56.824291
61 21 52.478567 52.478567 52.478567
61 22 60.415230 60.415230 60.415230
61 23 51.478151 51.478151 51.478151
61 24 59.615434 59.615434 59.615434
61 25 50.537115 50.537115 50.537115
61 26 58.523500 58.523500 58.523500
61 27 85.440037 85.440037 85.440037
61 28 83.815273 83.815273 83.815273
61 29 82.637764 82.637764 82.637764
61 30 79.056942 79.056942 79.056942
61 31 78.924014 78.924014 78.924014
61 32 77.162167 77.162167 77.162167
61 33 78.000000 78.000000 78.000000
61 34 78.262379 78.262379 78.262379
61 35 74.330344 74.330344 74.330344
61 36 57.697487 57.697487 57.697487
61 37 55.901699 55.901699 55.901699
61 38 54.626001 54.626001 54.626001
61 39 51.078371 51.078371 51.078371
61 40 49.244289 49.244289 49.244289
61 41 51.478151 51.478151 51.478151
61 42 45.541190 45.541190 45.541190
61 43 44.721360 44.721360 44.721360
61 44 47.169906 47.169906 47.169906
61 45 45.650849 45.650849 45.650849
61 46 22.561028 22.561028 22.561028
61 47 20.223748 20.223748 20.223748
61 48 19.849433 19.849433 19.849433
61 49 61.269895 61.269895 61.269895
61 50 55.072679 55.072679 55.072679
61 51 62.241465 62.241465 62.241465
61 52 56.568542 56.568542 56.568542
61 53 31.622777 31.622777 31.622777
61 54 11.180340 11.180340 11.180340
61 55 40.000000 40.000000 40.000000
61 56 15.000000 15.000000 15.000000
61 57 43.011626 43.011626 43.011626
61 58 38.078866 38.078866 38.078866
61 59 50.000000 50.000000 50.000000
61 60 40.311289 40.311289 40.311289
61 62 30.413813 30.413813 30.413813
61 63 55.901699 55.901699 55.901699
61 64 64.031242 64.031242 64.031242
61 65 42.426407 42.426407 42.426407
61 66 28.284271 28.284271 28.284271
61 67 34.713110 34.713110 34.713110
61 68 52.201533 52.201533 52.201533
61 69 25.000000 25.000000 25.000000
61 70 17.888544 17.888544 17.888544
61 71 21.931712 21.931712 21.931712
61 72 50.249378 50.249378 50.249378
61 73 48.259714 48.259714 48.259714
61 74 13.000000 13.000000 13.000000
61 75 40.311289 40.311289 40.311289
61 76 55.901699 55.901699 55.901699
61 77 65.795137 65.795137 65.795137
61 78 57.558666 57.558666 57.558666
61 79 8.062258 8.062258 8.062258
61 80 12.041595 12.041595 12.041595
61 81 34.539832 34.539832 34.539832
61 82 34.058773 34.058773 34.058773
61 83 20.808652 20.808652 20.808652
61 84 36.400549 36.400549 36.400549
61 85 52.201533 52.201533 52.201533
61 86 60.605280 60.605280 60.605280
61 87 36.496575 36.496575 36.496575
61 88 36.124784 36.124784 36.124784
61 89 9.219544 9.219544 9.219544
61 90 75.690158 75.690158 75.690158
61 91 25.553865 25.553865 25.553865
61 92 38.470768 38.470768 38.470768
61 93 41.629317 41.629317 41.629317
61 94 46.690470 46.690470 46.690470
61 95 43.680659 43.680659 43.680659
61 96 47.010637 47.010637 47.010637
61 97 40.447497 40.447497 40.447497
61 98 43.416587 43.416587 43.416587
61 99 13.601471 13.601471 13.601471
61 100 27.313001 27.313001 27.313001
61 101 17.464249 17.464249 17.464249
62 1 15.811388 15.811388 15.811388
62 2 28.284271 28.284271 28.284271
62 3 25.079872 25.079872 25.079872
62 4 30.479501 30.479501 30.479501
62 5 29.154759 29.154759 29.154759
62 6 32.015621 32.015621 32.015621
62 7 28.792360 28.792360 28.792360
62 8 31.622777 31.622777 31.622777
62 9 33.541020 33.541020 33.541020
62 10 46.097722 46.097722 46.097722
62 11 43.011626 43.011626 43.011626
62 12 44.654227 44.654227 44.654227
62 13 42.059482 42.059482 42.059482
62 14 50.000000 50.000000 50.000000
62 15 44.721360 44.721360 44.721360

62 15 44.721360 44.721360 44.721360
62 16 49.739320 49.739320 49.739320
62 17 51.478151 51.478151 51.478151
62 18 49.244289 49.244289 49.244289
62 19 60.008333 60.008333 60.008333
62 20 55.081757 55.081757 55.081757
62 21 50.089919 50.089919 50.089919
62 22 60.207973 60.207973 60.207973
62 23 50.249378 50.249378 50.249378
62 24 60.406953 60.406953 60.406953
62 25 50.487622 50.487622 50.487622
62 26 60.827625 60.827625 60.827625
62 27 61.032778 61.032778 61.032778
62 28 58.309519 58.309519 58.309519
62 29 58.600341 58.600341 58.600341
62 30 54.083269 54.083269 54.083269
62 31 55.443665 55.443665 55.443665
62 32 52.430907 52.430907 52.430907
62 33 54.671748 54.671748 54.671748
62 34 56.568542 56.568542 56.568542
62 35 50.000000 50.000000 50.000000
62 36 29.732137 29.732137 29.732137
62 37 28.284271 28.284271 28.284271
62 38 26.248809 26.248809 26.248809
62 39 22.671568 22.671568 22.671568
62 40 21.213203 21.213203 21.213203
62 41 25.000000 25.000000 25.000000
62 42 16.401219 16.401219 16.401219
62 43 18.027756 18.027756 18.027756
62 44 22.360680 22.360680 22.360680
62 45 19.723083 19.723083 19.723083
62 46 30.232433 30.232433 30.232433
62 47 30.886890 30.886890 30.886890
62 48 47.423623 47.423623 47.423623
62 49 60.074953 60.074953 60.074953
62 50 53.084838 53.084838 53.084838
62 51 40.360872 40.360872 40.360872
62 52 46.097722 46.097722 46.097722
62 53 40.311289 40.311289 40.311289
62 54 29.154759 29.154759 29.154759
62 55 11.180340 11.180340 11.180340
62 56 15.811388 15.811388 15.811388
62 57 30.413813 30.413813 30.413813
62 58 42.720019 42.720019 42.720019
62 59 62.649820 62.649820 62.649820
62 60 57.008771 57.008771 57.008771
62 61 30.413813 30.413813 30.413813
62 63 36.055513 36.055513 36.055513
62 64 49.244289 49.244289 49.244289
62 65 35.000000 35.000000 35.000000
62 66 26.925824 26.925824 26.925824
62 67 28.284271 28.284271 28.284271
62 68 29.832868 29.832868 29.832868
62 69 7.071068 7.071068 7.071068
62 70 19.104973 19.104973 19.104973
62 71 10.770330 10.770330 10.770330
62 72 22.360680 22.360680 22.360680
62 73 18.000000 18.000000 18.000000
62 74 43.289722 43.289722 43.289722
62 75 51.478151 51.478151 51.478151
62 76 72.111026 72.111026 72.111026
62 77 55.081757 55.081757 55.081757
62 78 65.787537 65.787537 65.787537
62 79 38.078866 38.078866 38.078866
62 80 39.115214 39.115214 39.115214
62 81 18.110770 18.110770 18.110770
62 82 8.062258 8.062258 8.062258
62 83 28.425341 28.425341 28.425341
62 84 34.928498 34.928498 34.928498
62 85 37.947332 37.947332 37.947332
62 86 45.694639 45.694639 45.694639
62 87 47.507894 47.507894 47.507894
62 88 52.630789 52.630789 52.630789
62 89 22.135944 22.135944 22.135944
62 90 63.906181 63.906181 63.906181
62 91 19.697716 19.697716 19.697716
62 92 23.345235 23.345235 23.345235
62 93 23.409400 23.409400 23.409400
62 94 20.615528 20.615528 20.615528
62 95 20.808652 20.808652 20.808652
62 96 30.083218 30.083218 30.083218
62 97 14.866069 14.866069 14.866069
62 98 62.369865 62.369865 62.369865
62 99 23.021729 23.021729 23.021729
62 100 35.510562 35.510562 35.510562
62 101 14.142136 14.142136 14.142136
63 1 29.154759 29.154759 29.154759
63 2 64.031242 64.031242 64.031242
63 3 58.728187 58.728187 58.728187
63 4 65.946948 65.946948 65.946948
63 5 63.639610 63.639610 63.639610
63 6 67.268120 67.268120 67.268120
63 7 61.717096 61.717096 61.717096
63 8 64.031242 64.031242 64.031242
63 9 67.268120 67.268120 67.268120
63 10 55.000000 55.000000 55.000000

63 11 55.226805 55.226805 55.226805
63 12 57.218878 57.218878 57.218878
63 13 57.870545 57.870545 57.870545
63 14 60.000000 60.000000 60.000000
63 15 60.827625 60.827625 60.827625
63 16 63.198101 63.198101 63.198101
63 17 65.192024 65.192024 65.192024
63 18 65.764732 65.764732 65.764732
63 19 36.619667 36.619667 36.619667
63 20 33.970576 33.970576 33.970576
63 21 30.479501 30.479501 30.479501
63 22 39.051248 39.051248 39.051248
63 23 32.015621 32.015621 32.015621
63 24 40.360872 40.360872 40.360872
63 25 33.600595 33.600595 33.600595
63 26 42.426407 42.426407 42.426407
63 27 30.413813 30.413813 30.413813
63 28 30.000000 30.000000 30.000000
63 29 27.459060 27.459060 27.459060
63 30 25.000000 25.000000 25.000000
63 31 23.537205 23.537205 23.537205
63 32 23.000000 23.000000 23.000000
63 33 22.561028 22.561028 22.561028
63 34 22.360680 22.360680 22.360680
63 35 20.000000 20.000000 20.000000
63 36 50.039984 50.039984 50.039984
63 37 50.000000 50.000000 50.000000
63 38 47.000000 47.000000 47.000000
63 39 45.099889 45.099889 45.099889
63 40 45.276926 45.276926 45.276926
63 41 50.249378 50.249378 50.249378
63 42 40.607881 40.607881 40.607881
63 43 46.097722 46.097722 46.097722
63 44 50.990195 50.990195 50.990195
63 45 48.052055 48.052055 48.052055
63 46 65.069194 65.069194 65.069194
63 47 65.069194 65.069194 65.069194
63 48 63.788714 63.788714 63.788714
63 49 37.802116 37.802116 37.802116
63 50 32.526912 32.526912 32.526912
63 51 7.000000 7.000000 7.000000
63 52 18.027756 18.027756 18.027756
63 53 40.311289 40.311289 40.311289
63 54 47.434165 47.434165 47.434165
63 55 26.925824 26.925824 26.925824
63 56 43.011626 43.011626 43.011626
63 57 15.000000 15.000000 15.000000
63 58 36.400549 36.400549 36.400549
63 59 55.901699 55.901699 55.901699
63 60 57.008771 57.008771 57.008771
63 61 55.901699 55.901699 55.901699
63 62 36.055513 36.055513 36.055513
63 64 15.000000 15.000000 15.000000
63 65 20.615528 20.615528 20.615528
63 66 30.413813 30.413813 30.413813
63 67 24.083189 24.083189 24.083189
63 68 7.071068 7.071068 7.071068
63 69 35.355339 35.355339 35.355339
63 70 38.013156 38.013156 38.013156
63 71 45.343136 45.343136 45.343136
63 72 20.000000 20.000000 20.000000
63 73 30.066593 30.066593 30.066593
63 74 67.779053 67.779053 67.779053
63 75 47.434165 47.434165 47.434165
63 76 67.082039 67.082039 67.082039
63 77 23.537205 23.537205 23.537205
63 78 52.801515 52.801515 52.801515
63 79 60.745370 60.745370 60.745370
63 80 67.601775 67.601775 67.601775
63 81 21.633308 21.633308 21.633308
63 82 28.017851 28.017851 28.017851
63 83 38.832976 38.832976 38.832976
63 84 28.284271 28.284271 28.284271
63 85 10.000000 10.000000 10.000000
63 86 12.165525 12.165525 12.165525
63 87 45.354162 45.354162 45.354162
63 88 54.129474 54.129474 54.129474
63 89 47.010637 47.010637 47.010637
63 90 30.066593 30.066593 30.066593
63 91 30.463092 30.463092 30.463092
63 92 17.464249 17.464249 17.464249
63 93 14.422205 14.422205 14.422205
63 94 17.464249 17.464249 17.464249
63 95 15.264338 15.264338 15.264338
63 96 9.219544 9.219544 9.219544
63 97 21.470911 21.470911 21.470911
63 98 63.324561 63.324561 63.324561
63 99 42.544095 42.544095 42.544095
63 100 39.000000 39.000000 39.000000
63 101 46.690470 46.690470 46.690470
64 1 39.051248 39.051248 39.051248
64 2 76.321688 76.321688 76.321688
64 3 69.814039 69.814039 69.814039
64 4 77.935871 77.935871 77.935871
64 5 75.000000 75.000000 75.000000

64 6 79.056942 79.056942 79.056942
64 7 72.346389 72.346389 72.346389
64 8 74.330344 74.330344 74.330344
64 9 78.102497 78.102497 78.102497
64 10 57.008771 57.008771 57.008771
64 11 58.523500 58.523500 58.523500
64 12 60.406953 60.406953 60.406953
64 13 62.241465 62.241465 62.241465
64 14 61.846584 61.846584 61.846584
64 15 65.000000 65.000000 65.000000
64 16 66.098411 66.098411 66.098411
64 17 68.007353 68.007353 68.007353
64 18 69.641941 69.641941 69.641941
64 19 25.806976 25.806976 25.806976
64 20 25.079872 25.079872 25.079872
64 21 23.537205 23.537205 23.537205
64 22 29.154759 29.154759 29.154759
64 23 25.495098 25.495098 25.495098
64 24 30.886890 30.886890 30.886890
64 25 27.459060 27.459060 27.459060
64 26 33.541020 33.541020 33.541020
64 27 31.622777 31.622777 31.622777
64 28 33.541020 33.541020 33.541020
64 29 28.792360 28.792360 28.792360
64 30 29.154759 29.154759 29.154759
64 31 25.079872 25.079872 25.079872
64 32 27.459060 27.459060 27.459060
64 33 24.166092 24.166092 24.166092
64 34 20.615528 20.615528 20.615528
64 35 25.000000 25.000000 25.000000
64 36 65.030762 65.030762 65.030762
64 37 65.000000 65.000000 65.000000
64 38 62.000000 62.000000 62.000000
64 39 60.074953 60.074953 60.074953
64 40 60.207973 60.207973 60.207973
64 41 65.192024 65.192024 65.192024
64 42 55.443665 55.443665 55.443665
64 43 60.827625 60.827625 60.827625
64 44 65.764732 65.764732 65.764732
64 45 62.801274 62.801274 62.801274
64 46 76.609399 76.609399 76.609399
64 47 76.216796 76.216796 76.216796
64 48 67.779053 67.779053 67.779053
64 49 27.459060 27.459060 27.459060
64 50 24.351591 24.351591 24.351591
64 51 16.552945 16.552945 16.552945
64 52 10.000000 10.000000 10.000000
64 53 41.231056 41.231056 41.231056
64 54 54.083269 54.083269 54.083269
64 55 41.231056 41.231056 41.231056
64 56 53.150729 53.150729 53.150729
64 57 21.213203 21.213203 21.213203
64 58 35.355339 35.355339 35.355339
64 59 50.990195 50.990195 50.990195
64 60 55.000000 55.000000 55.000000
64 61 64.031242 64.031242 64.031242
64 62 49.244289 49.244289 49.244289
64 63 15.000000 15.000000 15.000000
64 65 22.360680 22.360680 22.360680
64 66 36.055513 36.055513 36.055513
64 67 29.410882 29.410882 29.410882
64 68 22.022716 22.022716 22.022716
64 69 47.169906 47.169906 47.169906
64 70 46.690470 46.690470 46.690470
64 71 57.454330 57.454330 57.454330
64 72 35.000000 35.000000 35.000000
64 73 45.044423 45.044423 45.044423
64 74 74.625733 74.625733 74.625733
64 75 45.000000 45.000000 45.000000
64 76 61.846584 61.846584 61.846584
64 77 9.433981 9.433981 9.433981
64 78 45.310043 45.310043 45.310043
64 79 67.416615 67.416615 67.416615
64 80 76.059187 76.059187 76.059187
64 81 32.449961 32.449961 32.449961
64 82 41.231056 41.231056 41.231056
64 83 44.418465 44.418465 44.418465
64 84 30.083218 30.083218 30.083218
64 85 12.041595 12.041595 12.041595
64 86 3.605551 3.605551 3.605551
64 87 44.181444 44.181444 44.181444
64 88 53.150729 53.150729 53.150729
64 89 55.901699 55.901699 55.901699
64 90 15.132746 15.132746 15.132746
64 91 38.897301 38.897301 38.897301
64 92 27.202941 27.202941 27.202941
64 93 25.942244 25.942244 25.942244
64 94 32.249031 32.249031 32.249031
64 95 29.120440 29.120440 29.120440
64 96 19.235384 19.235384 19.235384
64 97 35.440090 35.440090 35.440090
64 98 61.032778 61.032778 61.032778
64 99 50.447993 50.447993 50.447993
64 100 41.785165 41.785165 41.785165
64 101 58.008620 58.008620 58.008620

65 1 20.615528 20.615528 20.615528
65 2 58.523500 58.523500 58.523500
65 3 50.537115 50.537115 50.537115
65 4 59.615434 59.615434 59.615434
65 5 55.901699 55.901699 55.901699
65 6 60.415230 60.415230 60.415230
65 7 52.478567 52.478567 52.478567
65 8 54.083269 54.083269 54.083269
65 9 58.309519 58.309519 58.309519
65 10 35.355339 35.355339 35.355339
65 11 36.400549 36.400549 36.400549
65 12 38.327536 38.327536 38.327536
65 13 39.924930 39.924930 39.924930
65 14 40.311289 40.311289 40.311289
65 15 42.720019 42.720019 42.720019
65 16 44.147480 44.147480 44.147480
65 17 46.097722 46.097722 46.097722
65 18 47.434165 47.434165 47.434165
65 19 25.019992 25.019992 25.019992
65 20 20.223748 20.223748 20.223748
65 21 15.297059 15.297059 15.297059
65 22 25.495098 25.495098 25.495098
65 23 15.811388 15.811388 15.811388
65 24 25.961510 25.961510 25.961510
65 25 16.552945 16.552945 16.552945
65 26 26.925824 26.925824 26.925824
65 27 50.000000 50.000000 50.000000
65 28 50.249378 50.249378 50.249378
65 29 47.000000 47.000000 47.000000
65 30 45.276926 45.276926 45.276926
65 31 43.000000 43.000000 43.000000
65 32 43.289722 43.289722 43.289722
65 33 42.000000 42.000000 42.000000
65 34 40.311289 40.311289 40.311289
65 35 40.311289 40.311289 40.311289
65 36 59.236813 59.236813 59.236813
65 37 58.523500 58.523500 58.523500
65 38 55.713553 55.713553 55.713553
65 39 52.810984 52.810984 52.810984
65 40 52.201533 52.201533 52.201533
65 41 57.008771 57.008771 57.008771
65 42 46.840154 46.840154 46.840154
65 43 50.990195 50.990195 50.990195
65 44 55.901699 55.901699 55.901699
65 45 52.952809 52.952809 52.952809
65 46 57.697487 57.697487 57.697487
65 47 56.824291 56.824291 56.824291
65 48 45.541190 45.541190 45.541190
65 49 25.179357 25.179357 25.179357
65 50 18.248288 18.248288 18.248288
65 51 27.459060 27.459060 27.459060
65 52 14.142136 14.142136 14.142136
65 53 20.000000 20.000000 20.000000
65 54 32.015621 32.015621 32.015621
65 55 31.622777 31.622777 31.622777
65 56 33.541020 33.541020 33.541020
65 57 7.071068 7.071068 7.071068
65 58 15.811388 15.811388 15.811388
65 59 36.055513 36.055513 36.055513
65 60 36.400549 36.400549 36.400549
65 61 42.426407 42.426407 42.426407
65 62 35.000000 35.000000 35.000000
65 63 20.615528 20.615528 20.615528
65 64 22.360680 22.360680 22.360680
65 66 14.142136 14.142136 14.142136
65 67 8.062258 8.062258 8.062258
65 68 22.472205 22.472205 22.472205
65 69 30.413813 30.413813 30.413813
65 70 26.076810 26.076810 26.076810
65 71 40.261644 40.261644 40.261644
65 72 32.015621 32.015621 32.015621
65 73 39.357337 39.357337 39.357337
65 74 52.430907 52.430907 52.430907
65 75 26.925824 26.925824 26.925824
65 76 47.169906 47.169906 47.169906
65 77 23.430749 23.430749 23.430749
65 78 34.828150 34.828150 34.828150
65 79 45.221676 45.221676 45.221676
65 80 54.451814 54.451814 54.451814
65 81 17.117243 17.117243 17.117243
65 82 28.284271 28.284271 28.284271
65 83 22.203603 22.203603 22.203603
65 84 8.062258 8.062258 8.062258
65 85 12.041595 12.041595 12.041595
65 86 19.313208 19.313208 19.313208
65 87 24.738634 24.738634 24.738634
65 88 33.541020 33.541020 33.541020
65 89 35.000000 35.000000 35.000000
65 90 33.301652 33.301652 33.301652
65 91 18.788294 18.788294 18.788294
65 92 12.649111 12.649111 12.649111
65 93 15.264338 15.264338 15.264338
65 94 27.202941 27.202941 27.202941
65 95 21.633308 21.633308 21.633308
65 96 13.038405 13.038405 13.038405

65 97 26.000000 26.000000 26.000000
65 98 42.720019 42.720019 42.720019
65 99 29.068884 29.068884 29.068884
65 100 19.646883 19.646883 19.646883
65 101 39.560081 39.560081 39.560081
66 1 11.180340 11.180340 11.180340
66 2 46.097722 46.097722 46.097722
66 3 37.336309 37.336309 37.336309
66 4 46.840154 46.840154 46.840154
66 5 42.720019 42.720019 42.720019
66 6 47.434165 47.434165 47.434165
66 7 38.910153 38.910153 38.910153
66 8 40.311289 40.311289 40.311289
66 9 44.721360 44.721360 44.721360
66 10 25.495098 25.495098 25.495098
66 11 25.000000 25.000000 25.000000
66 12 27.000000 27.000000 27.000000
66 13 27.459060 27.459060 27.459060
66 14 30.413813 30.413813 30.413813
66 15 30.413813 30.413813 30.413813
66 16 33.000000 33.000000 33.000000
66 17 35.000000 35.000000 35.000000
66 18 35.355339 35.355339 35.355339
66 19 36.138622 36.138622 36.138622
66 20 30.805844 30.805844 30.805844
66 21 25.961510 25.961510 25.961510
66 22 35.355339 35.355339 35.355339
66 23 25.495098 25.495098 25.495098
66 24 35.128336 35.128336 35.128336
66 25 25.179357 25.179357 25.179357
66 26 35.000000 35.000000 35.000000
66 27 60.827625 60.827625 60.827625
66 28 60.207973 60.207973 60.207973
66 29 57.870545 57.870545 57.870545
66 30 55.226805 55.226805 55.226805
66 31 53.935146 53.935146 53.935146
66 32 53.235327 53.235327 53.235327
66 33 52.952809 52.952809 52.952809
66 34 52.201533 52.201533 52.201533
66 35 50.249378 50.249378 50.249378
66 36 55.217751 55.217751 55.217751
66 37 54.083269 54.083269 54.083269
66 38 51.613952 51.613952 51.613952
66 39 48.259714 48.259714 48.259714
66 40 47.169906 47.169906 47.169906
66 41 51.478151 51.478151 51.478151
66 42 41.880783 41.880783 41.880783
66 43 44.721360 44.721360 44.721360
66 44 49.244289 49.244289 49.244289
66 45 46.518813 46.518813 46.518813
66 46 44.598206 44.598206 44.598206
66 47 43.462628 43.462628 43.462628
66 48 33.376639 33.376639 33.376639
66 49 35.693137 35.693137 35.693137
66 50 28.861739 28.861739 28.861739
66 51 37.336309 37.336309 37.336309
66 52 28.284271 28.284271 28.284271
66 53 14.142136 14.142136 14.142136
66 54 18.027756 18.027756 18.027756
66 55 28.284271 28.284271 28.284271
66 56 20.615528 20.615528 20.615528
66 57 15.811388 15.811388 15.811388
66 58 15.811388 15.811388 15.811388
66 59 36.055513 36.055513 36.055513
66 60 32.015621 32.015621 32.015621
66 61 28.284271 28.284271 28.284271
66 62 26.925824 26.925824 26.925824
66 63 30.413813 30.413813 30.413813
66 64 36.055513 36.055513 36.055513
66 65 14.142136 14.142136 14.142136
66 67 6.708204 6.708204 6.708204
66 68 29.068884 29.068884 29.068884
66 69 20.615528 20.615528 20.615528
66 70 12.649111 12.649111 12.649111
66 71 29.000000 29.000000 29.000000
66 72 33.541020 33.541020 33.541020
66 73 37.536649 37.536649 37.536649
66 74 38.587563 38.587563 38.587563
66 75 25.000000 25.000000 25.000000
66 76 46.097722 46.097722 46.097722
66 77 37.536649 37.536649 37.536649
66 78 38.897301 38.897301 38.897301
66 79 31.384710 31.384710 31.384710
66 80 40.311289 40.311289 40.311289
66 81 13.892444 13.892444 13.892444
66 82 22.803509 22.803509 22.803509
66 83 8.544004 8.544004 8.544004
66 84 9.219544 9.219544 9.219544
66 85 24.596748 24.596748 24.596748
66 86 32.756679 32.756679 32.756679
66 87 21.260292 21.260292 21.260292
66 88 28.017851 28.017851 28.017851
66 89 21.095023 21.095023 21.095023
66 90 47.423623 47.423623 47.423623
66 91 7.280110 7.280110 7.280110

66 92 14.142136 14.142136 14.142136
66 93 18.248288 18.248288 18.248288
66 94 28.635642 28.635642 28.635642
66 95 23.409400 23.409400 23.409400
66 96 21.213203 21.213203 21.213203
66 97 24.413111 24.413111 24.413111
66 98 38.013156 38.013156 38.013156
66 99 15.000000 15.000000 15.000000
66 100 10.295630 10.295630 10.295630
66 101 27.294688 27.294688 27.294688
67 1 13.038405 13.038405 13.038405
67 2 50.596443 50.596443 50.596443
67 3 42.485292 42.485292 42.485292
67 4 51.623638 51.623638 51.623638
67 5 47.853944 47.853944 47.853944
67 6 52.392748 52.392748 52.392748
67 7 44.418465 44.418465 44.418465
67 8 46.043458 46.043458 46.043458
67 9 50.249378 50.249378 50.249378
67 10 31.064449 31.064449 31.064449
67 11 31.144823 31.144823 31.144823
67 12 33.136083 33.136083 33.136083
67 13 33.955854 33.955854 33.955854
67 14 36.055513 36.055513 36.055513
67 15 36.878178 36.878178 36.878178
67 16 39.115214 39.115214 39.115214
67 17 41.109610 41.109610 41.109610
67 18 41.773197 41.773197 41.773197
67 19 32.140317 32.140317 32.140317
67 20 27.018512 27.018512 27.018512
67 21 22.022716 22.022716 22.022716
67 22 32.015621 32.015621 32.015621
67 23 22.022716 22.022716 22.022716
67 24 32.140317 32.140317 32.140317
67 25 22.203603 22.203603 22.203603
67 26 32.557641 32.557641 32.557641
67 27 54.451814 54.451814 54.451814
67 28 54.037024 54.037024 54.037024
67 29 51.478151 51.478151 51.478151
67 30 49.040799 49.040799 49.040799
67 31 47.518417 47.518417 47.518417
67 32 47.042534 47.042534 47.042534
67 33 46.529560 46.529560 46.529560
67 34 45.607017 45.607017 45.607017
67 35 44.045431 44.045431 44.045431
67 36 54.589376 54.589376 54.589376
67 37 53.665631 53.665631 53.665631
67 38 51.000000 51.000000 51.000000
67 39 47.853944 47.853944 47.853944
67 40 47.010637 47.010637 47.010637
67 41 51.623638 51.623638 51.623638
67 42 41.629317 41.629317 41.629317
67 43 45.221676 45.221676 45.221676
67 44 50.000000 50.000000 50.000000
67 45 47.127487 47.127487 47.127487
67 46 49.658836 49.658836 49.658836
67 47 48.764741 48.764741 48.764741
67 48 39.812058 39.812058 39.812058
67 49 32.015621 32.015621 32.015621
67 50 25.019992 25.019992 25.019992
67 51 31.064449 31.064449 31.064449
67 52 22.022716 22.022716 22.022716
67 53 17.464249 17.464249 17.464249
67 54 24.698178 24.698178 24.698178
67 55 26.925824 26.925824 26.925824
67 56 25.495098 25.495098 25.495098
67 57 9.219544 9.219544 9.219544
67 58 16.278821 16.278821 16.278821
67 59 37.483330 37.483330 37.483330
67 60 35.355339 35.355339 35.355339
67 61 34.713110 34.713110 34.713110
67 62 28.284271 28.284271 28.284271
67 63 24.083189 24.083189 24.083189
67 64 29.410882 29.410882 29.410882
67 65 8.062258 8.062258 8.062258
67 66 6.708204 6.708204 6.708204
67 68 23.537205 23.537205 23.537205
67 69 23.021729 23.021729 23.021729
67 70 18.027756 18.027756 18.027756
67 71 32.557641 32.557641 32.557641
67 72 30.000000 30.000000 30.000000
67 73 35.608988 35.608988 35.608988
67 74 45.276926 45.276926 45.276926
67 75 27.018512 27.018512 27.018512
67 76 48.166378 48.166378 48.166378
67 77 31.400637 31.400637 31.400637
67 78 38.470768 38.470768 38.470768
67 79 38.078866 38.078866 38.078866
67 80 46.754679 46.754679 46.754679
67 81 11.661904 11.661904 11.661904
67 82 22.472205 22.472205 22.472205
67 83 15.231546 15.231546 15.231546
67 84 7.211103 7.211103 7.211103
67 85 17.888544 17.888544 17.888544
67 86 26.076810 26.076810 26.076810

67 87 23.853721 23.853721 23.853721
67 88 31.780497 31.780497 31.780497
67 89 27.018512 27.018512 27.018512
67 90 41.231056 41.231056 41.231056
67 91 10.770330 10.770330 10.770330
67 92 9.433981 9.433981 9.433981
67 93 13.416408 13.416408 13.416408
67 94 25.000000 25.000000 25.000000
67 95 19.416488 19.416488 19.416488
67 96 15.000000 15.000000 15.000000
67 97 22.022716 22.022716 22.022716
67 98 41.593269 41.593269 41.593269
67 99 21.213203 21.213203 21.213203
67 100 15.132746 15.132746 15.132746
67 101 31.622777 31.622777 31.622777
68 1 25.298221 25.298221 25.298221
68 2 58.051701 58.051701 58.051701
68 3 53.413481 53.413481 53.413481
68 4 60.108236 60.108236 60.108236
68 5 58.137767 58.137767 58.137767
68 6 61.522354 61.522354 61.522354
68 7 56.612719 56.612719 56.612719
68 8 59.076222 59.076222 59.076222
68 9 62.008064 62.008064 62.008064
68 10 54.451814 54.451814 54.451814
68 11 54.037024 54.037024 54.037024
68 12 56.035703 56.035703 56.035703
68 13 56.080300 56.080300 56.080300
68 14 59.413803 59.413803 59.413803
68 15 59.076222 59.076222 59.076222
68 16 62.032250 62.032250 62.032250
68 17 64.031242 64.031242 64.031242
68 18 64.070274 64.070274 64.070274
68 19 42.059482 42.059482 42.059482
68 20 38.832976 38.832976 38.832976
68 21 34.828150 34.828150 34.828150
68 22 44.102154 44.102154 44.102154
68 23 36.124784 36.124784 36.124784
68 24 45.221676 45.221676 45.221676
68 25 37.483330 37.483330 37.483330
68 26 47.010637 47.010637 47.010637
68 27 33.241540 33.241540 33.241540
68 28 31.780497 31.780497 31.780497
68 29 30.463092 30.463092 30.463092
68 30 26.925824 26.925824 26.925824
68 31 26.832816 26.832816 26.832816
68 32 25.000000 25.000000 25.000000
68 33 25.942244 25.942244 25.942244
68 34 27.018512 27.018512 27.018512
68 35 22.135944 22.135944 22.135944
68 36 43.104524 43.104524 43.104524
68 37 43.011626 43.011626 43.011626
68 38 40.012498 40.012498 40.012498
68 39 38.052595 38.052595 38.052595
68 40 38.209946 38.209946 38.209946
68 41 43.185646 43.185646 43.185646
68 42 33.541020 33.541020 33.541020
68 43 39.051248 39.051248 39.051248
68 44 43.931765 43.931765 43.931765
68 45 41.000000 41.000000 41.000000
68 46 59.464275 59.464275 59.464275
68 47 59.665736 59.665736 59.665736
68 48 62.072538 62.072538 62.072538
68 49 43.046487 43.046487 43.046487
68 50 37.202150 37.202150 37.202150
68 51 10.630146 10.630146 10.630146
68 52 23.769729 23.769729 23.769729
68 53 40.804412 40.804412 40.804412
68 54 44.721360 44.721360 44.721360
68 55 20.124612 20.124612 20.124612
68 56 38.470768 38.470768 38.470768
68 57 15.652476 15.652476 15.652476
68 58 38.013156 38.013156 38.013156
68 59 58.523500 58.523500 58.523500
68 60 58.309519 58.309519 58.309519
68 61 52.201533 52.201533 52.201533
68 62 29.832868 29.832868 29.832868
68 63 7.071068 7.071068 7.071068
68 64 22.022716 22.022716 22.022716
68 65 22.472205 22.472205 22.472205
68 66 29.068884 29.068884 29.068884
68 67 23.537205 23.537205 23.537205
68 69 30.000000 30.000000 30.000000
68 70 34.481879 34.481879 34.481879
68 71 39.623226 39.623226 39.623226
68 72 13.038405 13.038405 13.038405
68 73 23.021729 23.021729 23.021729
68 74 64.560050 64.560050 64.560050
68 75 49.193496 49.193496 49.193496
68 76 69.641941 69.641941 69.641941
68 77 30.265492 30.265492 30.265492
68 78 56.586217 56.586217 56.586217
68 79 57.723479 57.723479 57.723479
68 80 63.560994 63.560994 63.560994
68 81 17.720045 17.720045 17.720045

68 82 21.931712 21.931712 21.931712
68 83 37.013511 37.013511 37.013511
68 84 29.154759 29.154759 29.154759
68 85 14.764823 14.764823 14.764823
68 86 19.026298 19.026298 19.026298
68 87 46.615448 46.615448 46.615448
68 88 55.027266 55.027266 55.027266
68 89 43.081318 43.081318 43.081318
68 90 37.121422 37.121422 37.121422
68 91 27.459060 27.459060 27.459060
68 92 15.000000 15.000000 15.000000
68 93 11.045361 11.045361 11.045361
68 94 10.440307 10.440307 10.440307
68 95 9.219544 9.219544 9.219544
68 96 9.433981 9.433981 9.433981
68 97 15.000000 15.000000 15.000000
68 98 64.621978 64.621978 64.621978
68 99 39.293765 39.293765 39.293765
68 100 38.639358 38.639358 38.639358
68 101 41.400483 41.400483 41.400483
69 1 10.000000 10.000000 10.000000
69 2 29.154759 29.154759 29.154759
69 3 23.430749 23.430749 23.430749
69 4 30.805844 30.805844 30.805844
69 5 28.284271 28.284271 28.284271
69 6 32.015621 32.015621 32.015621
69 7 26.627054 26.627054 26.627054
69 8 29.154759 29.154759 29.154759
69 9 32.015621 32.015621 32.015621
69 10 39.051248 39.051248 39.051248
69 11 36.055513 36.055513 36.055513
69 12 37.735925 37.735925 37.735925
69 13 35.341194 35.341194 35.341194
69 14 43.011626 43.011626 43.011626
69 15 38.078866 38.078866 38.078866
69 16 42.941821 42.941821 42.941821
69 17 44.721360 44.721360 44.721360
69 18 42.720019 42.720019 42.720019
69 19 55.145263 55.145263 55.145263
69 20 50.039984 50.039984 50.039984
69 21 45.044423 45.044423 45.044423
69 22 55.000000 55.000000 55.000000
69 23 45.000000 45.000000 45.000000
69 24 55.036352 55.036352 55.036352
69 25 45.044423 45.044423 45.044423
69 26 55.226805 55.226805 55.226805
69 27 62.649820 62.649820 62.649820
69 28 60.415230 60.415230 60.415230
69 29 60.033324 60.033324 60.033324
69 30 55.901699 55.901699 55.901699
69 31 56.603887 56.603887 56.603887
69 32 54.120237 54.120237 54.120237
69 33 55.758407 55.758407 55.758407
69 34 57.008771 57.008771 57.008771
69 35 51.478151 51.478151 51.478151
69 36 36.796739 36.796739 36.796739
69 37 35.355339 35.355339 35.355339
69 38 33.301652 33.301652 33.301652
69 39 29.732137 29.732137 29.732137
69 40 28.284271 28.284271 28.284271
69 41 32.015621 32.015621 32.015621
69 42 23.430749 23.430749 23.430749
69 43 25.000000 25.000000 25.000000
69 44 29.154759 29.154759 29.154759
69 45 26.627054 26.627054 26.627054
69 46 29.732137 29.732137 29.732137
69 47 29.732137 29.732137 29.732137
69 48 40.853396 40.853396 40.853396
69 49 55.036352 55.036352 55.036352
69 50 48.041649 48.041649 48.041649
69 51 40.607881 40.607881 40.607881
69 52 42.720019 42.720019 42.720019
69 53 33.541020 33.541020 33.541020
69 54 22.360680 22.360680 22.360680
69 55 15.000000 15.000000 15.000000
69 56 10.000000 10.000000 10.000000
69 57 26.925824 26.925824 26.925824
69 58 36.400549 36.400549 36.400549
69 59 55.901699 55.901699 55.901699
69 60 50.000000 50.000000 50.000000
69 61 25.000000 25.000000 25.000000
69 62 7.071068 7.071068 7.071068
69 63 35.355339 35.355339 35.355339
69 64 47.169906 47.169906 47.169906
69 65 30.413813 30.413813 30.413813
69 66 20.615528 20.615528 20.615528
69 67 23.021729 23.021729 23.021729
69 68 30.000000 30.000000 30.000000
69 70 12.041595 12.041595 12.041595
69 71 10.295630 10.295630 10.295630
69 72 25.495098 25.495098 25.495098
69 73 23.537205 23.537205 23.537205
69 74 38.000000 38.000000 38.000000
69 75 44.721360 44.721360 44.721360
69 76 65.192024 65.192024 65.192024

69 77 52.000000 52.000000 52.000000
69 78 59.481089 59.481089 59.481089
69 79 32.249031 32.249031 32.249031
69 80 34.928498 34.928498 34.928498
69 81 14.764823 14.764823 14.764823
69 82 9.219544 9.219544 9.219544
69 83 21.400935 21.400935 21.400935
69 84 29.154759 29.154759 29.154759
69 85 35.355339 35.355339 35.355339
69 86 43.566042 43.566042 43.566042
69 87 40.706265 40.706265 40.706265
69 88 45.607017 45.607017 45.607017
69 89 16.124515 16.124515 16.124515
69 90 61.269895 61.269895 61.269895
69 91 13.341664 13.341664 13.341664
69 92 20.124612 20.124612 20.124612
69 93 21.400935 21.400935 21.400935
69 94 22.472205 22.472205 22.472205
69 95 20.808652 20.808652 20.808652
69 96 28.017851 28.017851 28.017851
69 97 16.155494 16.155494 16.155494
69 98 55.317267 55.317267 55.317267
69 99 16.124515 16.124515 16.124515
69 100 28.653098 28.653098 28.653098
69 101 11.401754 11.401754 11.401754
70 1 9.219544 9.219544 9.219544
70 2 33.541020 33.541020 33.541020
70 3 24.698178 24.698178 24.698178
70 4 34.205263 34.205263 34.205263
70 5 30.083218 30.083218 30.083218
70 6 34.785054 34.785054 34.785054
70 7 26.419690 26.419690 26.419690
70 8 28.017851 28.017851 28.017851
70 9 32.249031 32.249031 32.249031
70 10 27.018512 27.018512 27.018512
70 11 24.186773 24.186773 24.186773
70 12 25.942244 25.942244 25.942244
70 13 24.041631 24.041631 24.041631
70 14 31.064449 31.064449 31.064449
70 15 26.925824 26.925824 26.925824
70 16 31.384710 31.384710 31.384710
70 17 33.241540 33.241540 33.241540
70 18 31.780497 31.780497 31.780497
70 19 48.764741 48.764741 48.764741
70 20 43.416587 43.416587 43.416587
70 21 38.600518 38.600518 38.600518
70 22 47.853944 47.853944 47.853944
70 23 38.078866 38.078866 38.078866
70 24 47.518417 47.518417 47.518417
70 25 37.656341 37.656341 37.656341
70 26 47.169906 47.169906 47.169906
70 27 67.675697 67.675697 67.675697
70 28 66.219333 66.219333 66.219333
70 29 64.845971 64.845971 64.845971
70 30 61.400326 61.400326 61.400326
70 31 61.098281 61.098281 61.098281
70 32 59.481089 59.481089 59.481089
70 33 60.166436 60.166436 60.166436
70 34 60.373835 60.373835 60.373835
70 35 56.612719 56.612719 56.612719
70 36 48.836462 48.836462 48.836462
70 37 47.381431 47.381431 47.381431
70 38 45.343136 45.343136 45.343136
70 39 41.773197 41.773197 41.773197
70 40 40.311289 40.311289 40.311289
70 41 43.931765 43.931765 43.931765
70 42 35.468296 35.468296 35.468296
70 43 36.878178 36.878178 36.878178
70 44 40.804412 40.804412 40.804412
70 45 38.418745 38.418745 38.418745
70 46 31.953091 31.953091 31.953091
70 47 30.870698 30.870698 30.870698
70 48 29.832868 29.832868 29.832868
70 49 48.270074 48.270074 48.270074
70 50 41.484937 41.484937 41.484937
70 51 44.384682 44.384682 44.384682
70 52 40.000000 40.000000 40.000000
70 53 22.803509 22.803509 22.803509
70 54 11.180340 11.180340 11.180340
70 55 25.298221 25.298221 25.298221
70 56 8.062258 8.062258 8.062258
70 57 25.495098 25.495098 25.495098
70 58 27.018512 27.018512 27.018512
70 59 44.944410 44.944410 44.944410
70 60 38.275318 38.275318 38.275318
70 61 17.888544 17.888544 17.888544
70 62 19.104973 19.104973 19.104973
70 63 38.013156 38.013156 38.013156
70 64 46.690470 46.690470 46.690470
70 65 26.076810 26.076810 26.076810
70 66 12.649111 12.649111 12.649111
70 67 18.027756 18.027756 18.027756
70 68 34.481879 34.481879 34.481879
70 69 12.041595 12.041595 12.041595
70 71 17.464249 17.464249 17.464249

70 72 34.132096 34.132096 34.132096
70 73 34.539832 34.539832 34.539832
70 74 30.083218 30.083218 30.083218
70 75 33.837849 33.837849 33.837849
70 76 53.712196 53.712196 53.712196
70 77 49.406477 49.406477 49.406477
70 78 49.648766 49.648766 49.648766
70 79 23.345235 23.345235 23.345235
70 80 29.681644 29.681644 29.681644
70 81 16.763055 16.763055 16.763055
70 82 18.973666 18.973666 18.973666
70 83 9.848858 9.848858 9.848858
70 84 21.840330 21.840330 21.840330
70 85 34.713110 34.713110 34.713110
70 86 43.185646 43.185646 43.185646
70 87 29.732137 29.732137 29.732137
70 88 33.837849 33.837849 33.837849
70 89 9.219544 9.219544 9.219544
70 90 59.203040 59.203040 59.203040
70 91 7.810250 7.810250 7.810250
70 92 20.591260 20.591260 20.591260
70 93 23.769729 23.769729 23.769729
70 94 30.000000 30.000000 30.000000
70 95 26.305893 26.305893 26.305893
70 96 29.154759 29.154759 29.154759
70 97 24.083189 24.083189 24.083189
70 98 43.416587 43.416587 43.416587
70 99 5.000000 5.000000 5.000000
70 100 17.720045 17.720045 17.720045
70 101 15.000000 15.000000 15.000000
71 1 19.646883 19.646883 19.646883
71 2 18.867962 18.867962 18.867962
71 3 14.317821 14.317821 14.317821
71 4 20.615528 20.615528 20.615528
71 5 18.601075 18.601075 18.601075
71 6 21.931712 21.931712 21.931712
71 7 18.027756 18.027756 18.027756
71 8 20.880613 20.880613 20.880613
71 9 22.825424 22.825424 22.825424
71 10 42.201896 42.201896 42.201896
71 11 38.288379 38.288379 38.288379
71 12 39.623226 39.623226 39.623226
71 13 36.124784 36.124784 36.124784
71 14 45.343136 45.343136 45.343136
71 15 38.418745 38.418745 38.418745
71 16 43.931765 43.931765 43.931765
71 17 45.453273 45.453273 45.453273
71 18 42.438190 42.438190 42.438190
71 19 64.629715 64.629715 64.629715
71 20 59.413803 59.413803 59.413803
71 21 54.451814 54.451814 54.451814
71 22 64.195015 64.195015 64.195015
71 23 54.230987 54.230987 54.230987
71 24 64.070274 64.070274 64.070274
71 25 54.083269 54.083269 54.083269
71 26 64.000000 64.000000 64.000000
71 27 71.561163 71.561163 71.561163
71 28 68.963759 68.963759 68.963759
71 29 69.065187 69.065187 69.065187
71 30 64.660653 64.660653 64.660653
71 31 65.802736 65.802736 65.802736
71 32 62.968246 62.968246 62.968246
71 33 65.000000 65.000000 65.000000
71 34 66.603303 66.603303 66.603303
71 35 60.464866 60.464866 60.464866
71 36 35.777088 35.777088 35.777088
71 37 34.000000 34.000000 34.000000
71 38 32.695565 32.695565 32.695565
71 39 29.154759 29.154759 29.154759
71 40 27.313001 27.313001 27.313001
71 41 29.681644 29.681644 29.681644
71 42 23.769729 23.769729 23.769729
71 43 22.825424 22.825424 22.825424
71 44 25.612497 25.612497 25.612497
71 45 23.853721 23.853721 23.853721
71 46 19.849433 19.849433 19.849433
71 47 20.248457 20.248457 20.248457
71 48 40.804412 40.804412 40.804412
71 49 64.381674 64.381674 64.381674
71 50 57.428216 57.428216 57.428216
71 51 50.249378 50.249378 50.249378
71 52 52.924474 52.924474 52.924474
71 53 40.261644 40.261644 40.261644
71 54 24.207437 24.207437 24.207437
71 55 21.931712 21.931712 21.931712
71 56 10.295630 10.295630 10.295630
71 57 37.161808 37.161808 37.161808
71 58 44.283180 44.283180 44.283180
71 59 62.297673 62.297673 62.297673
71 60 55.009090 55.009090 55.009090
71 61 21.931712 21.931712 21.931712
71 62 10.770330 10.770330 10.770330
71 63 45.343136 45.343136 45.343136
71 64 57.454330 57.454330 57.454330
71 65 40.261644 40.261644 40.261644

71 66 29.000000 29.000000 29.000000
71 67 32.557641 32.557641 32.557641
71 68 39.623226 39.623226 39.623226
71 69 10.295630 10.295630 10.295630
71 70 17.464249 17.464249 17.464249
71 72 33.105891 33.105891 33.105891
71 73 28.284271 28.284271 28.284271
71 74 34.205263 34.205263 34.205263
71 75 51.244512 51.244512 51.244512
71 76 70.682388 70.682388 70.682388
71 77 62.241465 62.241465 62.241465
71 78 67.082039 67.082039 67.082039
71 79 29.966648 29.966648 29.966648
71 80 29.017236 29.017236 29.017236
71 81 25.059928 25.059928 25.059928
71 82 17.804494 17.804494 17.804494
71 83 27.202941 27.202941 27.202941
71 84 38.052595 38.052595 38.052595
71 85 45.650849 45.650849 45.650849
71 86 53.851648 53.851648 53.851648
71 87 47.127487 47.127487 47.127487
71 88 50.537115 50.537115 50.537115
71 89 15.556349 15.556349 15.556349
71 90 71.554175 71.554175 71.554175
71 91 22.090722 22.090722 22.090722
71 92 30.413813 30.413813 30.413813
71 93 31.622777 31.622777 31.622777
71 94 31.064449 31.064449 31.064449
71 95 30.413813 30.413813 30.413813
71 96 38.275318 38.275318 38.275318
71 97 25.000000 25.000000 25.000000
71 98 59.682493 59.682493 59.682493
71 99 19.235384 19.235384 19.235384
71 100 35.171011 35.171011 35.171011
71 101 4.472136 4.472136 4.472136
72 1 25.495098 25.495098 25.495098
72 2 50.000000 50.000000 50.000000
72 3 47.423623 47.423623 47.423623
72 4 52.430907 52.430907 52.430907
72 5 51.478151 51.478151 51.478151
72 6 54.083269 54.083269 54.083269
72 7 51.078371 51.078371 51.078371
72 8 53.851648 53.851648 53.851648
72 9 55.901699 55.901699 55.901699
72 10 58.523500 58.523500 58.523500
72 11 57.008771 57.008771 57.008771
72 12 58.940648 58.940648 58.940648
72 13 57.870545 57.870545 57.870545
72 14 63.245553 63.245553 63.245553
72 15 60.827625 60.827625 60.827625
72 16 64.761099 64.761099 64.761099
72 17 66.708320 66.708320 66.708320
72 18 65.764732 65.764732 65.764732
72 19 54.230987 54.230987 54.230987
72 20 50.537115 50.537115 50.537115
72 21 46.141088 46.141088 46.141088
72 22 55.901699 55.901699 55.901699
72 23 47.169906 47.169906 47.169906
72 24 56.824291 56.824291 56.824291
72 25 48.259714 48.259714 48.259714
72 26 58.309519 58.309519 58.309519
72 27 39.051248 39.051248 39.051248
72 28 36.055513 36.055513 36.055513
72 29 36.796739 36.796739 36.796739
72 30 32.015621 32.015621 32.015621
72 31 33.970576 33.970576 33.970576
72 32 30.479501 30.479501 30.479501
72 33 33.301652 33.301652 33.301652
72 34 36.055513 36.055513 36.055513
72 35 28.284271 28.284271 28.284271
72 36 30.066593 30.066593 30.066593
72 37 30.000000 30.000000 30.000000
72 38 27.000000 27.000000 27.000000
72 39 25.179357 25.179357 25.179357
72 40 25.495098 25.495098 25.495098
72 41 30.413813 30.413813 30.413813
72 42 21.189620 21.189620 21.189620
72 43 26.925824 26.925824 26.925824
72 44 31.622777 31.622777 31.622777
72 45 28.792360 28.792360 28.792360
72 46 52.478567 52.478567 52.478567
72 47 53.235327 53.235327 53.235327
72 48 63.788714 63.788714 63.788714
72 49 55.036352 55.036352 55.036352
72 50 48.764741 48.764741 48.764741
72 51 21.189620 21.189620 21.189620
72 52 36.400549 36.400549 36.400549
72 53 47.169906 47.169906 47.169906
72 54 45.276926 45.276926 45.276926
72 55 11.180340 11.180340 11.180340
72 56 35.355339 35.355339 35.355339
72 57 25.000000 25.000000 25.000000
72 58 46.097722 46.097722 46.097722
72 59 67.268120 67.268120 67.268120
72 60 65.192024 65.192024 65.192024

72 61 50.249378 50.249378 50.249378
72 62 22.360680 22.360680 22.360680
72 63 20.000000 20.000000 20.000000
72 64 35.000000 35.000000 35.000000
72 65 32.015621 32.015621 32.015621
72 66 33.541020 33.541020 33.541020
72 67 30.000000 30.000000 30.000000
72 68 13.038405 13.038405 13.038405
72 69 25.495098 25.495098 25.495098
72 70 34.132096 34.132096 34.132096
72 71 33.105891 33.105891 33.105891
72 73 10.198039 10.198039 10.198039
72 74 63.198101 63.198101 63.198101
72 75 57.008771 57.008771 57.008771
72 76 78.102497 78.102497 78.102497
72 77 43.289722 43.289722 43.289722
72 78 66.843100 66.843100 66.843100
72 79 57.008771 57.008771 57.008771
72 80 60.415230 60.415230 60.415230
72 81 19.697716 19.697716 19.697716
72 82 16.278821 16.278821 16.278821
72 83 39.849718 39.849718 39.849718
72 84 36.878178 36.878178 36.878178
72 85 27.202941 27.202941 27.202941
72 86 32.062439 32.062439 32.062439
72 87 53.823787 53.823787 53.823787
72 88 61.400326 61.400326 61.400326
72 89 41.109610 41.109610 41.109610
72 90 50.039984 50.039984 50.039984
72 91 29.120440 29.120440 29.120440
72 92 20.615528 20.615528 20.615528
72 93 16.970563 16.970563 16.970563
72 94 5.000000 5.000000 5.000000
72 95 10.630146 10.630146 10.630146
72 96 20.124612 20.124612 20.124612
72 97 10.049876 10.049876 10.049876
72 98 71.344236 71.344236 71.344236
72 99 39.115214 39.115214 39.115214
72 100 43.829214 43.829214 43.829214
72 101 36.055513 36.055513 36.055513
73 1 27.459060 27.459060 27.459060
73 2 42.941821 42.941821 42.941821
73 3 42.201896 42.201896 42.201896
73 4 45.617979 45.617979 45.617979
73 5 45.541190 45.541190 45.541190
73 6 47.423623 47.423623 47.423623
73 7 46.097722 46.097722 46.097722
73 8 49.030603 49.030603 49.030603
73 9 50.289164 50.289164 50.289164
73 10 60.901560 60.901560 60.901560
73 11 58.600341 58.600341 58.600341
73 12 60.415230 60.415230 60.415230
73 13 58.523500 58.523500 58.523500
73 14 65.299311 65.299311 65.299311
73 15 61.351447 61.351447 61.351447
73 16 65.924199 65.924199 65.924199
73 17 67.779053 67.779053 67.779053
73 18 66.098411 66.098411 66.098411
73 19 62.936476 62.936476 62.936476
73 20 58.872744 58.872744 58.872744
73 21 54.230987 54.230987 54.230987
73 22 64.257295 64.257295 64.257295
73 23 55.036352 55.036352 55.036352
73 24 65.000000 65.000000 65.000000
73 25 55.901699 55.901699 55.901699
73 26 66.211781 66.211781 66.211781
73 27 47.423623 47.423623 47.423623
73 28 43.863424 43.863424 43.863424
73 29 45.453273 45.453273 45.453273
73 30 40.360872 40.360872 40.360872
73 31 43.011626 43.011626 43.011626
73 32 39.051248 39.051248 39.051248
73 33 42.438190 42.438190 42.438190
73 34 45.650849 45.650849 45.650849
73 35 37.202150 37.202150 37.202150
73 36 20.396078 20.396078 20.396078
73 37 20.099751 20.099751 20.099751
73 38 17.117243 17.117243 17.117243
73 39 15.033296 15.033296 15.033296
73 40 15.297059 15.297059 15.297059
73 41 20.223748 20.223748 20.223748
73 42 11.180340 11.180340 11.180340
73 43 17.000000 17.000000 17.000000
73 44 21.540659 21.540659 21.540659
73 45 18.788294 18.788294 18.788294
73 46 46.238512 46.238512 46.238512
73 47 47.434165 47.434165 47.434165
73 48 64.195015 64.195015 64.195015
73 49 63.568860 63.568860 63.568860
73 50 57.008771 57.008771 57.008771
73 51 31.320920 31.320920 31.320920
73 52 45.705580 45.705580 45.705580
73 53 51.662365 51.662365 51.662365
73 54 45.541190 45.541190 45.541190
73 55 9.433981 9.433981 9.433981

73 56 33.376639 33.376639 33.376639
73 57 32.695565 32.695565 32.695565
73 58 51.855569 51.855569 51.855569
73 59 73.000000 73.000000 73.000000
73 60 69.526973 69.526973 69.526973
73 61 48.259714 48.259714 48.259714
73 62 18.000000 18.000000 18.000000
73 63 30.066593 30.066593 30.066593
73 64 45.044423 45.044423 45.044423
73 65 39.357337 39.357337 39.357337
73 66 37.536649 37.536649 37.536649
73 67 35.608988 35.608988 35.608988
73 68 23.021729 23.021729 23.021729
73 69 23.537205 23.537205 23.537205
73 70 34.539832 34.539832 34.539832
73 71 28.284271 28.284271 28.284271
73 72 10.198039 10.198039 10.198039
73 74 61.204575 61.204575 61.204575
73 75 62.241465 62.241465 62.241465
73 76 83.450584 83.450584 83.450584
73 77 53.084838 53.084838 53.084838
73 78 73.783467 73.783467 73.783467
73 79 55.731499 55.731499 55.731499
73 80 57.078893 57.078893 57.078893
73 81 24.083189 24.083189 24.083189
73 82 15.652476 15.652476 15.652476
73 83 42.190046 42.190046 42.190046
73 84 42.801869 42.801869 42.801869
73 85 36.496575 36.496575 36.496575
73 86 42.000000 42.000000 42.000000
73 87 58.694122 58.694122 58.694122
73 88 65.436993 65.436993 65.436993
73 89 39.623226 39.623226 39.623226
73 90 60.133186 60.133186 60.133186
73 91 31.622777 31.622777 31.622777
73 92 26.925824 26.925824 26.925824
73 93 24.166092 24.166092 24.166092
73 94 13.152946 13.152946 13.152946
73 95 18.027756 18.027756 18.027756
73 96 28.861739 28.861739 28.861739
73 97 13.601471 13.601471 13.601471
73 98 75.432089 75.432089 75.432089
73 99 39.217343 39.217343 39.217343
73 100 47.634021 47.634021 47.634021
73 101 32.062439 32.062439 32.062439
74 1 39.293765 39.293765 39.293765
74 2 33.970576 33.970576 33.970576
74 3 25.000000 25.000000 25.000000
74 4 32.015621 32.015621 32.015621
74 5 26.907248 26.907248 26.907248
74 6 30.805844 30.805844 30.805844
74 7 21.931712 21.931712 21.931712
74 8 19.849433 19.849433 19.849433
74 9 23.853721 23.853721 23.853721
74 10 26.248809 26.248809 26.248809
74 11 21.540659 21.540659 21.540659
74 12 20.880613 20.880613 20.880613
74 13 16.155494 16.155494 16.155494
74 14 25.179357 25.179357 25.179357
74 15 15.297059 15.297059 15.297059
74 16 20.000000 20.000000 20.000000
74 17 20.099751 20.099751 20.099751
74 18 15.132746 15.132746 15.132746
74 19 69.202601 69.202601 69.202601
74 20 64.031242 64.031242 64.031242
74 21 60.207973 60.207973 60.207973
74 22 66.850580 66.850580 66.850580
74 23 58.898217 58.898217 58.898217
74 24 65.734314 65.734314 65.734314
74 25 57.628118 57.628118 57.628118
74 26 64.140471 64.140471 64.140471
74 27 97.718985 97.718985 97.718985
74 28 96.301610 96.301610 96.301610
74 29 94.868330 94.868330 94.868330
74 30 91.482239 91.482239 91.482239
74 31 91.082380 91.082380 91.082380
74 32 89.560036 89.560036 89.560036
74 33 90.138782 90.138782 90.138782
74 34 90.077744 90.077744 90.077744
74 35 86.683332 86.683332 86.683332
74 36 69.641941 69.641941 69.641941
74 37 67.779053 67.779053 67.779053
74 38 66.730802 66.730802 66.730802
74 39 63.245553 63.245553 63.245553
74 40 61.351447 61.351447 61.351447
74 41 63.158531 63.158531 63.158531
74 42 57.974132 57.974132 57.974132
74 43 56.648036 56.648036 56.648036
74 44 58.600341 58.600341 58.600341
74 45 57.384667 57.384667 57.384667
74 46 28.425341 28.425341 28.425341
74 47 25.612497 25.612497 25.612497
74 48 15.000000 15.000000 15.000000
74 49 68.007353 68.007353 68.007353
74 50 62.481997 62.481997 62.481997
74 51 74.330344 74.330344 74.330344

74 51 74.330344 74.330344 74.330344
74 52 66.400301 66.400301 66.400301
74 53 37.802116 37.802116 37.802116
74 54 20.591260 20.591260 20.591260
74 55 53.000000 53.000000 53.000000
74 56 28.000000 28.000000 28.000000
74 57 54.120237 54.120237 54.120237
74 58 44.821870 44.821870 44.821870
74 59 51.662365 51.662365 51.662365
74 60 40.792156 40.792156 40.792156
74 61 13.000000 13.000000 13.000000
74 62 43.289722 43.289722 43.289722
74 63 67.779053 67.779053 67.779053
74 64 74.625733 74.625733 74.625733
74 65 52.430907 52.430907 52.430907
74 66 38.587563 38.587563 38.587563
74 67 45.276926 45.276926 45.276926
74 68 64.560050 64.560050 64.560050
74 69 38.000000 38.000000 38.000000
74 70 30.083218 30.083218 30.083218
74 71 34.205263 34.205263 34.205263
74 72 63.198101 63.198101 63.198101
74 73 61.204575 61.204575 61.204575
74 75 43.863424 43.863424 43.863424
74 76 55.081757 55.081757 55.081757
74 77 75.286121 75.286121 75.286121
74 78 60.745370 60.745370 60.745370
74 79 7.211103 7.211103 7.211103
74 80 8.944272 8.944272 8.944272
74 81 46.840154 46.840154 46.840154
74 82 47.042534 47.042534 47.042534
74 83 30.232433 30.232433 30.232433
74 84 45.453273 45.453273 45.453273
74 85 63.134776 63.134776 63.134776
74 86 71.344236 71.344236 71.344236
74 87 40.706265 40.706265 40.706265
74 88 37.363083 37.363083 37.363083
74 89 22.090722 22.090722 22.090722
74 90 85.146932 85.146932 85.146932
74 91 37.336309 37.336309 37.336309
74 92 50.328918 50.328918 50.328918
74 93 53.758720 53.758720 53.758720
74 94 59.539903 59.539903 59.539903
74 95 56.293872 56.293872 56.293872
74 96 58.694122 58.694122 58.694122
74 97 53.338541 53.338541 53.338541
74 98 42.047592 42.047592 42.047592
74 99 25.298221 25.298221 25.298221
74 100 34.655447 34.655447 34.655447
74 101 29.832868 29.832868 29.832868
75 1 36.055513 36.055513 36.055513
75 2 65.192024 65.192024 65.192024
75 3 55.036352 55.036352 55.036352
75 4 65.030762 65.030762 65.030762
75 5 60.000000 60.000000 60.000000
75 6 65.000000 65.000000 65.000000
75 7 55.036352 55.036352 55.036352
75 8 55.226805 55.226805 55.226805
75 9 60.207973 60.207973 60.207973
75 10 18.027756 18.027756 18.027756
75 11 22.360680 22.360680 22.360680
75 12 23.323808 23.323808 23.323808
75 13 27.730849 27.730849 27.730849
75 14 21.213203 21.213203 21.213203
75 15 29.154759 29.154759 29.154759
75 16 26.907248 26.907248 26.907248
75 17 28.284271 28.284271 28.284271
75 18 32.015621 32.015621 32.015621
75 19 28.301943 28.301943 28.301943
75 20 24.166092 24.166092 24.166092
75 21 22.561028 22.561028 22.561028
75 22 25.000000 25.000000 25.000000
75 23 20.615528 20.615528 20.615528
75 24 23.430749 23.430749 23.430749
75 25 18.681542 18.681542 18.681542
75 26 21.213203 21.213203 21.213203
75 27 75.663730 75.663730 75.663730
75 28 76.485293 76.485293 76.485293
75 29 72.691127 72.691127 72.691127
75 30 71.589105 71.589105 71.589105
75 31 68.731361 68.731361 68.731361
75 32 69.634761 69.634761 69.634761
75 33 67.742158 67.742158 67.742158
75 34 65.192024 65.192024 65.192024
75 35 66.708320 66.708320 66.708320
75 36 80.212219 80.212219 80.212219
75 37 79.056942 79.056942 79.056942
75 38 76.609399 76.609399 76.609399
75 39 73.239334 73.239334 73.239334
75 40 72.111026 72.111026 72.111026
75 41 76.321688 76.321688 76.321688
75 42 66.850580 66.850580 66.850580
75 43 69.462220 69.462220 69.462220
75 44 73.824115 73.824115 73.824115
75 45 71.196910 71.196910 71.196910
75 46 62.000000 62.000000 62.000000

75 46 62.000000 62.000000 62.000000
75 47 60.033324 60.033324 60.033324
75 48 30.805844 30.805844 30.805844
75 49 26.627054 26.627054 26.627054
75 50 23.409400 23.409400 23.409400
75 51 54.120237 54.120237 54.120237
75 52 35.000000 35.000000 35.000000
75 53 11.180340 11.180340 11.180340
75 54 30.000000 30.000000 30.000000
75 55 53.150729 53.150729 53.150729
75 56 41.231056 41.231056 41.231056
75 57 33.541020 33.541020 33.541020
75 58 11.180340 11.180340 11.180340
75 59 11.180340 11.180340 11.180340
75 60 10.000000 10.000000 10.000000
75 61 40.311289 40.311289 40.311289
75 62 51.478151 51.478151 51.478151
75 63 47.434165 47.434165 47.434165
75 64 45.000000 45.000000 45.000000
75 65 26.925824 26.925824 26.925824
75 66 25.000000 25.000000 25.000000
75 67 27.018512 27.018512 27.018512
75 68 49.193496 49.193496 49.193496
75 69 44.721360 44.721360 44.721360
75 70 33.837849 33.837849 33.837849
75 71 51.244512 51.244512 51.244512
75 72 57.008771 57.008771 57.008771
75 73 62.241465 62.241465 62.241465
75 74 43.863424 43.863424 43.863424
75 76 21.213203 21.213203 21.213203
75 77 40.792156 40.792156 40.792156
75 78 17.262677 17.262677 17.262677
75 79 37.947332 37.947332 37.947332
75 80 50.000000 50.000000 50.000000
75 81 38.183766 38.183766 38.183766
75 82 47.801674 47.801674 47.801674
75 83 24.041631 24.041631 24.041631
75 84 20.248457 20.248457 20.248457
75 85 38.078866 38.078866 38.078866
75 86 43.104524 43.104524 43.104524
75 87 4.123106 4.123106 4.123106
75 88 8.944272 8.944272 8.944272
75 89 38.209946 38.209946 38.209946
75 90 49.335586 49.335586 49.335586
75 91 31.906112 31.906112 31.906112
75 92 36.400549 36.400549 36.400549
75 93 40.224371 40.224371 40.224371
75 94 52.009614 52.009614 52.009614
75 95 46.400431 46.400431 46.400431
75 96 39.812058 39.812058 39.812058
75 97 48.795492 48.795492 48.795492
75 98 16.124515 16.124515 16.124515
75 99 32.557641 32.557641 32.557641
75 100 16.155494 16.155494 16.155494
75 101 48.270074 48.270074 48.270074
76 1 57.008771 57.008771 57.008771
76 2 82.462113 82.462113 82.462113
76 3 72.034714 72.034714 72.034714
76 4 81.786307 81.786307 81.786307
76 5 76.485293 76.485293 76.485293
76 6 81.394103 81.394103 81.394103
76 7 71.196910 71.196910 71.196910
76 8 70.710678 70.710678 70.710678
76 9 75.663730 75.663730 75.663730
76 10 30.413813 30.413813 30.413813
76 11 35.355339 35.355339 35.355339
76 12 35.128336 35.128336 35.128336
76 13 40.112342 40.112342 40.112342
76 14 30.000000 30.000000 30.000000
76 15 40.000000 40.000000 40.000000
76 16 35.128336 35.128336 35.128336
76 17 35.355339 35.355339 35.355339
76 18 40.311289 40.311289 40.311289
76 19 39.000000 39.000000 39.000000
76 20 37.336309 37.336309 37.336309
76 21 38.327536 38.327536 38.327536
76 22 35.000000 35.000000 35.000000
76 23 36.400549 36.400549 36.400549
76 24 33.000000 33.000000 33.000000
76 25 34.481879 34.481879 34.481879
76 26 30.000000 30.000000 30.000000
76 27 93.407708 93.407708 93.407708
76 28 94.868330 94.868330 94.868330
76 29 90.520716 90.520716 90.520716
76 30 90.138782 90.138782 90.138782
76 31 86.683332 86.683332 86.683332
76 32 88.255311 88.255311 88.255311
76 33 85.726309 85.726309 85.726309
76 34 82.462113 82.462113 82.462113
76 35 85.440037 85.440037 85.440037
76 36 101.212647 101.212647 101.212647
76 37 100.000000 100.000000 100.000000
76 38 97.616597 97.616597 97.616597
76 39 94.201911 94.201911 94.201911
76 40 93.005376 93.005376 93.005376
76 41 97.082439 97.082439 97.082439

76 41 97.082439 97.082439 97.082439
76 42 87.800911 87.800911 87.800911
76 43 90.138782 90.138782 90.138782
76 44 94.339811 94.339811 94.339811
76 45 91.809586 91.809586 91.809586
76 46 78.447435 78.447435 78.447435
76 47 76.118329 76.118329 76.118329
76 48 40.112342 40.112342 40.112342
76 49 37.000000 37.000000 37.000000
76 50 37.656341 37.656341 37.656341
76 51 73.409809 73.409809 73.409809
76 52 52.201533 52.201533 52.201533
76 53 32.015621 32.015621 32.015621
76 54 47.434165 47.434165 47.434165
76 55 74.330344 74.330344 74.330344
76 56 60.415230 60.415230 60.415230
76 57 54.083269 54.083269 54.083269
76 58 32.015621 32.015621 32.015621
76 59 11.180340 11.180340 11.180340
76 60 15.811388 15.811388 15.811388
76 61 55.901699 55.901699 55.901699
76 62 72.111026 72.111026 72.111026
76 63 67.082039 67.082039 67.082039
76 64 61.846584 61.846584 61.846584
76 65 47.169906 47.169906 47.169906
76 66 46.097722 46.097722 46.097722
76 67 48.166378 48.166378 48.166378
76 68 69.641941 69.641941 69.641941
76 69 65.192024 65.192024 65.192024
76 70 53.712196 53.712196 53.712196
76 71 70.682388 70.682388 70.682388
76 72 78.102497 78.102497 78.102497
76 73 83.450584 83.450584 83.450584
76 74 55.081757 55.081757 55.081757
76 75 21.213203 21.213203 21.213203
76 77 55.443665 55.443665 55.443665
76 78 18.110770 18.110770 18.110770
76 79 51.088159 51.088159 51.088159
76 80 63.007936 63.007936 63.007936
76 81 59.396970 59.396970 59.396970
76 82 68.883960 68.883960 68.883960
76 83 43.908997 43.908997 43.908997
76 84 41.231056 41.231056 41.231056
76 85 57.271284 57.271284 57.271284
76 86 60.728906 60.728906 60.728906
76 87 24.839485 24.839485 24.839485
76 88 20.248457 20.248457 20.248457
76 89 56.302753 56.302753 56.302753
76 90 62.000000 62.000000 62.000000
76 91 52.801515 52.801515 52.801515
76 92 57.489129 57.489129 57.489129
76 93 61.220911 61.220911 61.220911
76 94 73.109507 73.109507 73.109507
76 95 67.475922 67.475922 67.475922
76 96 60.207973 60.207973 60.207973
76 97 70.007142 70.007142 70.007142
76 98 13.038405 13.038405 13.038405
76 99 51.478151 51.478151 51.478151
76 100 36.619667 36.619667 36.619667
76 101 67.230945 67.230945 67.230945
77 1 42.941821 42.941821 42.941821
77 2 80.956779 80.956779 80.956779
77 3 73.573093 73.573093 73.573093
77 4 82.298238 82.298238 82.298238
77 5 78.892332 78.892332 78.892332
77 6 83.240615 83.240615 83.240615
77 7 75.716577 75.716577 75.716577
77 8 77.420927 77.420927 77.420927
77 9 81.541401 81.541401 81.541401
77 10 55.036352 55.036352 55.036352
77 11 57.306195 57.306195 57.306195
77 12 59.059292 59.059292 59.059292
77 13 61.587336 61.587336 61.587336
77 14 59.615434 59.615434 59.615434
77 15 64.140471 64.140471 64.140471
77 16 64.404969 64.404969 64.404969
77 17 66.211781 66.211781 66.211781
77 18 68.476273 68.476273 68.476273
77 19 17.464249 17.464249 17.464249
77 20 18.110770 18.110770 18.110770
77 21 18.248288 18.248288 18.248288
77 22 21.189620 21.189620 21.189620
77 23 20.223748 20.223748 20.223748
77 24 23.086793 23.086793 23.086793
77 25 22.203603 22.203603 22.203603
77 26 25.961510 25.961510 25.961510
77 27 39.357337 39.357337 39.357337
77 28 41.880783 41.880783 41.880783
77 29 36.715120 36.715120 36.715120
77 30 37.802116 37.802116 37.802116
77 31 33.286634 33.286634 33.286634
77 32 36.235342 36.235342 36.235342
77 33 32.449961 32.449961 32.449961
77 34 28.178006 28.178006 28.178006
77 35 33.970576 33.970576 33.970576
77 36 73.334848 73.334848 73.334848

77 37 73.171033 73.171033 73.171033
77 38 70.178344 70.178344 70.178344
77 39 68.029405 68.029405 68.029405
77 40 68.000000 68.000000 68.000000
77 41 73.000000 73.000000 73.000000
77 42 63.031738 63.031738 63.031738
77 43 68.183576 68.183576 68.183576
77 44 73.171033 73.171033 73.171033
77 45 70.178344 70.178344 70.178344
77 46 80.622577 80.622577 80.622577
77 47 79.924965 79.924965 79.924965
77 48 66.730802 66.730802 66.730802
77 49 19.313208 19.313208 19.313208
77 50 18.000000 18.000000 18.000000
77 51 25.942244 25.942244 25.942244
77 52 9.433981 9.433981 9.433981
77 53 39.357337 39.357337 39.357337
77 54 55.172457 55.172457 55.172457
77 55 48.259714 48.259714 48.259714
77 56 56.603887 56.603887 56.603887
77 57 25.079872 25.079872 25.079872
77 58 32.695565 32.695565 32.695565
77 59 45.044423 45.044423 45.044423
77 60 50.635956 50.635956 50.635956
77 61 65.795137 65.795137 65.795137
77 62 55.081757 55.081757 55.081757
77 63 23.537205 23.537205 23.537205
77 64 9.433981 9.433981 9.433981
77 65 23.430749 23.430749 23.430749
77 66 37.536649 37.536649 37.536649
77 67 31.400637 31.400637 31.400637
77 68 30.265492 30.265492 30.265492
77 69 52.000000 52.000000 52.000000
77 70 49.406477 49.406477 49.406477
77 71 62.241465 62.241465 62.241465
77 72 43.289722 43.289722 43.289722
77 73 53.084838 53.084838 53.084838
77 74 75.286121 75.286121 75.286121
77 75 40.792156 40.792156 40.792156
77 76 55.443665 55.443665 55.443665
77 78 38.078866 38.078866 38.078866
77 79 68.117545 68.117545 68.117545
77 80 77.794601 77.794601 77.794601
77 81 37.336309 37.336309 37.336309
77 82 47.296934 47.296934 47.296934
77 83 45.276926 45.276926 45.276926
77 84 29.832868 29.832868 29.832868
77 85 17.262677 17.262677 17.262677
77 86 11.401754 11.401754 11.401754
77 87 40.804412 40.804412 40.804412
77 88 49.477268 49.477268 49.477268
77 89 58.412327 58.412327 58.412327
77 90 9.899495 9.899495 9.899495
77 91 41.880783 41.880783 41.880783
77 92 31.953091 31.953091 31.953091
77 93 31.780497 31.780497 31.780497
77 94 40.012498 40.012498 40.012498
77 95 36.124784 36.124784 36.124784
77 96 25.317978 25.317978 25.317978
77 97 42.296572 42.296572 42.296572
77 98 56.320511 56.320511 56.320511
77 99 52.497619 52.497619 52.497619
77 100 41.048752 41.048752 41.048752
77 101 62.177166 62.177166 62.177166
78 1 49.979996 49.979996 49.979996
78 2 82.024387 82.024387 82.024387
78 3 72.006944 72.006944 72.006944
78 4 82.006097 82.006097 82.006097
78 5 77.058419 77.058419 77.058419
78 6 82.054860 82.054860 82.054860
78 7 72.173402 72.173402 72.173402
78 8 72.443081 72.443081 72.443081
78 9 77.414469 77.414469 77.414469
78 10 34.539832 34.539832 34.539832
78 11 39.217343 39.217343 39.217343
78 12 39.924930 39.924930 39.924930
78 13 44.598206 44.598206 44.598206
78 14 36.715120 36.715120 36.715120
78 15 45.694639 45.694639 45.694639
78 16 42.544095 42.544095 42.544095
78 17 43.566042 43.566042 43.566042
78 18 47.885280 47.885280 47.885280
78 19 21.095023 21.095023 21.095023
78 20 20.248457 20.248457 20.248457
78 21 22.472205 22.472205 22.472205
78 22 17.117243 17.117243 17.117243
78 23 20.808652 20.808652 20.808652
78 24 15.132746 15.132746 15.132746
78 25 19.209373 19.209373 19.209373
78 26 12.165525 12.165525 12.165525
78 27 76.896034 76.896034 76.896034
78 28 78.790862 78.790862 78.790862
78 29 74.094534 74.094534 74.094534
78 30 74.249579 74.249579 74.249579
78 31 70.384657 70.384657 70.384657

78 32 72.449983 72.449983 72.449983
78 33 69.462220 69.462220 69.462220
78 34 65.787537 65.787537 65.787537
78 35 69.771054 69.771054 69.771054
78 36 93.059121 93.059121 93.059121
78 37 92.130342 92.130342 92.130342
78 38 89.470666 89.470666 89.470666
78 39 86.313383 86.313383 86.313383
78 40 85.428333 85.428333 85.428333
78 41 89.961103 89.961103 89.961103
78 42 80.056230 80.056230 80.056230
78 43 83.384651 83.384651 83.384651
78 44 88.022724 88.022724 88.022724
78 45 85.234969 85.234969 85.234969
78 46 79.056942 79.056942 79.056942
78 47 77.162167 77.162167 77.162167
78 48 46.957428 46.957428 46.957428
78 49 19.104973 19.104973 19.104973
78 50 21.023796 21.023796 21.023796
78 51 58.523500 58.523500 58.523500
78 52 36.235342 36.235342 36.235342
78 53 27.073973 27.073973 27.073973
78 54 47.095647 47.095647 47.095647
78 55 65.368188 65.368188 65.368188
78 56 57.428216 57.428216 57.428216
78 57 41.868843 41.868843 41.868843
78 58 23.086793 23.086793 23.086793
78 59 10.630146 10.630146 10.630146
78 60 21.400935 21.400935 21.400935
78 61 57.558666 57.558666 57.558666
78 62 65.787537 65.787537 65.787537
78 63 52.801515 52.801515 52.801515
78 64 45.310043 45.310043 45.310043
78 65 34.828150 34.828150 34.828150
78 66 38.897301 38.897301 38.897301
78 67 38.470768 38.470768 38.470768
78 68 56.586217 56.586217 56.586217
78 69 59.481089 59.481089 59.481089
78 70 49.648766 49.648766 49.648766
78 71 67.082039 67.082039 67.082039
78 72 66.843100 66.843100 66.843100
78 73 73.783467 73.783467 73.783467
78 74 60.745370 60.745370 60.745370
78 75 17.262677 17.262677 17.262677
78 76 18.110770 18.110770 18.110770
78 77 38.078866 38.078866 38.078866
78 79 55.081757 55.081757 55.081757
78 80 67.186308 67.186308 67.186308
78 81 50.119856 50.119856 50.119856
78 82 60.835845 60.835845 60.835845
78 83 40.199502 40.199502 40.199502
78 84 31.304952 31.304952 31.304952
78 85 42.801869 42.801869 42.801869
78 86 44.721360 44.721360 44.721360
78 87 21.095023 21.095023 21.095023
78 88 23.706539 23.706539 23.706539
78 89 55.009090 55.009090 55.009090
78 90 44.045431 44.045431 44.045431
78 91 46.173586 46.173586 46.173586
78 92 46.872167 46.872167 46.872167
78 93 50.000000 50.000000 50.000000
78 94 62.008064 62.008064 62.008064
78 95 56.400355 56.400355 56.400355
78 96 47.381431 47.381431 47.381431
78 97 60.207973 60.207973 60.207973
78 98 24.207437 24.207437 24.207437
78 99 49.091751 49.091751 49.091751
78 100 32.140317 32.140317 32.140317
78 101 64.498062 64.498062 64.498062
79 1 32.557641 32.557641 32.557641
79 2 33.615473 33.615473 33.615473
79 3 23.600847 23.600847 23.600847
79 4 32.202484 32.202484 32.202484
79 5 26.832816 26.832816 26.832816
79 6 31.384710 31.384710 31.384710
79 7 21.470911 21.470911 21.470911
79 8 20.248457 20.248457 20.248457
79 9 25.000000 25.000000 25.000000
79 10 21.095023 21.095023 21.095023
79 11 16.124515 16.124515 16.124515
79 12 16.000000 16.000000 16.000000
79 13 11.000000 11.000000 11.000000
79 14 21.213203 21.213203 21.213203
79 15 11.401754 11.401754 11.401754
79 16 17.088007 17.088007 17.088007
79 17 17.888544 17.888544 17.888544
79 18 13.601471 13.601471 13.601471
79 19 62.425956 62.425956 62.425956
79 20 57.201399 57.201399 57.201399
79 21 53.263496 53.263496 53.263496
79 22 60.207973 60.207973 60.207973
79 23 52.009614 52.009614 52.009614
79 24 59.169249 59.169249 59.169249
79 25 50.803543 50.803543 50.803543
79 26 57.706152 57.706152 57.706152

79 27 90.801982 90.801982 90.801982
79 28 89.498603 89.498603 89.498603
79 29 87.931792 87.931792 87.931792
79 30 84.646323 84.646323 84.646323
79 31 84.118963 84.118963 84.118963
79 32 82.710338 82.710338 82.710338
79 33 83.168504 83.168504 83.168504
79 34 83.006024 83.006024 83.006024
79 35 79.812280 79.812280 79.812280
79 36 65.741920 65.741920 65.741920
79 37 63.953108 63.953108 63.953108
79 38 62.649820 62.649820 62.649820
79 39 59.093147 59.093147 59.093147
79 40 57.271284 57.271284 57.271284
79 41 59.539903 59.539903 59.539903
79 42 53.488316 53.488316 53.488316
79 43 52.773099 52.773099 52.773099
79 44 55.226805 55.226805 55.226805
79 45 53.712196 53.712196 53.712196
79 46 28.635642 28.635642 28.635642
79 47 26.000000 26.000000 26.000000
79 48 12.529964 12.529964 12.529964
79 49 61.294372 61.294372 61.294372
79 50 55.605755 55.605755 55.605755
79 51 67.357256 67.357256 67.357256
79 52 59.203040 59.203040 59.203040
79 53 31.064449 31.064449 31.064449
79 54 13.416408 13.416408 13.416408
79 55 47.169906 47.169906 47.169906
79 56 22.360680 22.360680 22.360680
79 57 46.957428 46.957428 46.957428
79 58 38.013156 38.013156 38.013156
79 59 46.529560 46.529560 46.529560
79 60 36.055513 36.055513 36.055513
79 61 8.062258 8.062258 8.062258
79 62 38.078866 38.078866 38.078866
79 63 60.745370 60.745370 60.745370
79 64 67.416615 67.416615 67.416615
79 65 45.221676 45.221676 45.221676
79 66 31.384710 31.384710 31.384710
79 67 38.078866 38.078866 38.078866
79 68 57.723479 57.723479 57.723479
79 69 32.249031 32.249031 32.249031
79 70 23.345235 23.345235 23.345235
79 71 29.966648 29.966648 29.966648
79 72 57.008771 57.008771 57.008771
79 73 55.731499 55.731499 55.731499
79 74 7.211103 7.211103 7.211103
79 75 37.947332 37.947332 37.947332
79 76 51.088159 51.088159 51.088159
79 77 68.117545 68.117545 68.117545
79 78 55.081757 55.081757 55.081757
79 80 12.165525 12.165525 12.165525
79 81 40.024992 40.024992 40.024992
79 82 41.048752 41.048752 41.048752
79 83 23.021729 23.021729 23.021729
79 84 38.288379 38.288379 38.288379
79 85 55.946403 55.946403 55.946403
79 86 64.140471 64.140471 64.140471
79 87 34.539832 34.539832 34.539832
79 88 32.249031 32.249031 32.249031
79 89 16.124515 16.124515 16.124515
79 90 77.987178 77.987178 77.987178
79 91 30.364453 30.364453 30.364453
79 92 43.324358 43.324358 43.324358
79 93 46.840154 46.840154 46.840154
79 94 53.150729 53.150729 53.150729
79 95 49.648766 49.648766 49.648766
79 96 51.623638 51.623638 51.623638
79 97 47.042534 47.042534 47.042534
79 98 38.209946 38.209946 38.209946
79 99 18.439089 18.439089 18.439089
79 100 27.658633 27.658633 27.658633
79 101 25.495098 25.495098 25.495098
80 1 38.470768 38.470768 38.470768
80 2 25.495098 25.495098 25.495098
80 3 17.464249 17.464249 17.464249
80 4 23.345235 23.345235 23.345235
80 5 18.439089 18.439089 18.439089
80 6 22.022716 22.022716 22.022716
80 7 13.892444 13.892444 13.892444
80 8 11.401754 11.401754 11.401754
80 9 15.000000 15.000000 15.000000
80 10 33.241540 33.241540 33.241540
80 11 28.284271 28.284271 28.284271
80 12 28.071338 28.071338 28.071338
80 13 23.086793 23.086793 23.086793
80 14 33.015148 33.015148 33.015148
80 15 23.021729 23.021729 23.021729
80 16 28.284271 28.284271 28.284271
80 17 28.635642 28.635642 28.635642
80 18 23.769729 23.769729 23.769729
80 19 73.573093 73.573093 73.573093
80 20 68.264193 68.264193 68.264193
80 21 64.070274 64.070274 64.070274

80 22 71.589105 71.589105 71.589105
80 23 62.968246 62.968246 62.968246
80 24 70.661163 70.661163 70.661163
80 25 61.911227 61.911227 61.911227
80 26 69.354164 69.354164 69.354164
80 27 96.772930 96.772930 96.772930
80 28 94.921020 94.921020 94.921020
80 29 94.021274 94.021274 94.021274
80 30 90.249654 90.249654 90.249654
80 31 90.376988 90.376988 90.376988
80 32 88.391176 88.391176 88.391176
80 33 89.470666 89.470666 89.470666
80 34 89.944427 89.944427 89.944427
80 35 85.615419 85.615419 85.615419
80 36 63.324561 63.324561 63.324561
80 37 61.400326 61.400326 61.400326
80 38 60.638272 60.638272 60.638272
80 39 57.271284 57.271284 57.271284
80 40 55.317267 55.317267 55.317267
80 41 56.612719 56.612719 56.612719
80 42 52.469038 52.469038 52.469038
80 43 50.447993 50.447993 50.447993
80 44 51.865210 51.865210 51.865210
80 45 50.960769 50.960769 50.960769
80 46 19.798990 19.798990 19.798990
80 47 16.970563 16.970563 16.970563
80 48 23.345235 23.345235 23.345235
80 49 72.560320 72.560320 72.560320
80 50 66.573268 66.573268 66.573268
80 51 73.790243 73.790243 73.790243
80 52 68.593003 68.593003 68.593003
80 53 42.485292 42.485292 42.485292
80 54 22.803509 22.803509 22.803509
80 55 49.648766 49.648766 49.648766
80 56 25.298221 25.298221 25.298221
80 57 55.000000 55.000000 55.000000
80 58 49.244289 49.244289 49.244289
80 59 58.694122 58.694122 58.694122
80 60 48.166378 48.166378 48.166378
80 61 12.041595 12.041595 12.041595
80 62 39.115214 39.115214 39.115214
80 63 67.601775 67.601775 67.601775
80 64 76.059187 76.059187 76.059187
80 65 54.451814 54.451814 54.451814
80 66 40.311289 40.311289 40.311289
80 67 46.754679 46.754679 46.754679
80 68 63.560994 63.560994 63.560994
80 69 34.928498 34.928498 34.928498
80 70 29.681644 29.681644 29.681644
80 71 29.017236 29.017236 29.017236
80 72 60.415230 60.415230 60.415230
80 73 57.078893 57.078893 57.078893
80 74 8.944272 8.944272 8.944272
80 75 50.000000 50.000000 50.000000
80 76 63.007936 63.007936 63.007936
80 77 77.794601 77.794601 77.794601
80 78 67.186308 67.186308 67.186308
80 79 12.165525 12.165525 12.165525
80 81 46.065171 46.065171 46.065171
80 82 44.147480 44.147480 44.147480
80 83 32.649655 32.649655 32.649655
80 84 48.270074 48.270074 48.270074
80 85 64.202804 64.202804 64.202804
80 86 72.622311 72.622311 72.622311
80 87 46.486557 46.486557 46.486557
80 88 44.407207 44.407207 44.407207
80 89 20.591260 20.591260 20.591260
80 90 87.692645 87.692645 87.692645
80 91 37.443290 37.443290 37.443290
80 92 50.249378 50.249378 50.249378
80 93 53.235327 53.235327 53.235327
80 94 57.280014 57.280014 57.280014
80 95 54.781384 54.781384 54.781384
80 96 58.830264 58.830264 58.830264
80 97 50.960769 50.960769 50.960769
80 98 50.039984 50.039984 50.039984
80 99 25.612497 25.612497 25.612497
80 100 38.587563 38.587563 38.587563
80 101 25.019992 25.019992 25.019992
81 1 7.615773 7.615773 7.615773
81 2 43.908997 43.908997 43.908997
81 3 37.536649 37.536649 37.536649
81 4 45.486262 45.486262 45.486262
81 5 42.638011 42.638011 42.638011
81 6 46.615448 46.615448 46.615448
81 7 40.311289 40.311289 40.311289
81 8 42.520583 42.520583 42.520583
81 9 45.967380 45.967380 45.967380
81 10 38.897301 38.897301 38.897301
81 11 37.656341 37.656341 37.656341
81 12 39.623226 39.623226 39.623226
81 13 39.051248 39.051248 39.051248
81 14 43.680659 43.680659 43.680659
81 15 42.047592 42.047592 42.047592
81 16 45.541190 45.541190 45.541190

81 17 47.518417 47.518417 47.518417
81 18 47.042534 47.042534 47.042534
81 19 42.107007 42.107007 42.107007
81 20 37.336309 37.336309 37.336309
81 21 32.388269 32.388269 32.388269
81 22 42.579338 42.579338 42.579338
81 23 32.756679 32.756679 32.756679
81 24 42.953463 42.953463 42.953463
81 25 33.241540 33.241540 33.241540
81 26 43.680659 43.680659 43.680659
81 27 50.921508 50.921508 50.921508
81 28 49.477268 49.477268 49.477268
81 29 48.104054 48.104054 48.104054
81 30 44.643029 44.643029 44.643029
81 31 44.384682 44.384682 44.384682
81 32 42.720019 42.720019 42.720019
81 33 43.462628 43.462628 43.462628
81 34 43.908997 43.908997 43.908997
81 35 39.849718 39.849718 39.849718
81 36 42.941821 42.941821 42.941821
81 37 42.047592 42.047592 42.047592
81 38 39.357337 39.357337 39.357337
81 39 36.249138 36.249138 36.249138
81 40 35.468296 35.468296 35.468296
81 41 40.162171 40.162171 40.162171
81 42 30.083218 30.083218 30.083218
81 43 33.955854 33.955854 33.955854
81 44 38.832976 38.832976 38.832976
81 45 35.902646 35.902646 35.902646
81 46 44.204072 44.204072 44.204072
81 47 43.931765 43.931765 43.931765
81 48 45.044423 45.044423 45.044423
81 49 42.296572 42.296572 42.296572
81 50 35.355339 35.355339 35.355339
81 51 27.730849 27.730849 27.730849
81 52 28.160256 28.160256 28.160256
81 53 27.802878 27.802878 27.802878
81 54 27.166155 27.166155 27.166155
81 55 15.264338 15.264338 15.264338
81 56 21.400935 21.400935 21.400935
81 57 12.369317 12.369317 12.369317
81 58 27.802878 27.802878 27.802878
81 59 48.918299 48.918299 48.918299
81 60 45.803930 45.803930 45.803930
81 61 34.539832 34.539832 34.539832
81 62 18.110770 18.110770 18.110770
81 63 21.633308 21.633308 21.633308
81 64 32.449961 32.449961 32.449961
81 65 17.117243 17.117243 17.117243
81 66 13.892444 13.892444 13.892444
81 67 11.661904 11.661904 11.661904
81 68 17.720045 17.720045 17.720045
81 69 14.764823 14.764823 14.764823
81 70 16.763055 16.763055 16.763055
81 71 25.059928 25.059928 25.059928
81 72 19.697716 19.697716 19.697716
81 73 24.083189 24.083189 24.083189
81 74 46.840154 46.840154 46.840154
81 75 38.183766 38.183766 38.183766
81 76 59.396970 59.396970 59.396970
81 77 37.336309 37.336309 37.336309
81 78 50.119856 50.119856 50.119856
81 79 40.024992 40.024992 40.024992
81 80 46.065171 46.065171 46.065171
81 82 11.180340 11.180340 11.180340
81 83 20.396078 20.396078 20.396078
81 84 18.867962 18.867962 18.867962
81 85 20.591260 20.591260 20.591260
81 86 28.844410 28.844410 28.844410
81 87 34.713110 34.713110 34.713110
81 88 41.880783 41.880783 41.880783
81 89 25.495098 25.495098 25.495098
81 90 46.518813 46.518813 46.518813
81 91 10.000000 10.000000 10.000000
81 92 5.385165 5.385165 5.385165
81 93 7.211103 7.211103 7.211103
81 94 14.866069 14.866069 14.866069
81 95 10.049876 10.049876 10.049876
81 96 13.453624 13.453624 13.453624
81 97 10.630146 10.630146 10.630146
81 98 51.865210 51.865210 51.865210
81 99 21.587033 21.587033 21.587033
81 100 24.186773 24.186773 24.186773
81 101 25.612497 25.612497 25.612497
82 1 12.041595 12.041595 12.041595
82 2 36.124784 36.124784 36.124784
82 3 31.906112 31.906112 31.906112
82 4 38.183766 38.183766 38.183766
82 5 36.400549 36.400549 36.400549
82 6 39.623226 39.623226 39.623226
82 7 35.355339 35.355339 35.355339
82 8 38.013156 38.013156 38.013156
82 9 40.496913 40.496913 40.496913
82 10 45.276926 45.276926 45.276926
82 11 42.953463 42.953463 42.953463

82 12 44.777226 44.777226 44.777226
82 13 43.011626 43.011626 43.011626
82 14 49.648766 49.648766 49.648766
82 15 45.880279 45.880279 45.880279
82 16 50.328918 50.328918 50.328918
82 17 52.201533 52.201533 52.201533
82 18 50.695167 50.695167 50.695167
82 19 53.235327 53.235327 53.235327
82 20 48.507731 48.507731 48.507731
82 21 43.566042 43.566042 43.566042
82 22 53.758720 53.758720 53.758720
82 23 43.931765 43.931765 43.931765
82 24 54.129474 54.129474 54.129474
82 25 44.384682 44.384682 44.384682
82 26 54.817880 54.817880 54.817880
82 27 53.851648 53.851648 53.851648
82 28 51.429563 51.429563 51.429563
82 29 51.312766 51.312766 51.312766
82 30 47.010637 47.010637 47.010637
82 31 48.010416 48.010416 48.010416
82 32 45.276926 45.276926 45.276926
82 33 47.201695 47.201695 47.201695
82 34 48.836462 48.836462 48.836462
82 35 42.720019 42.720019 42.720019
82 36 32.449961 32.449961 32.449961
82 37 31.384710 31.384710 31.384710
82 38 28.844410 28.844410 28.844410
82 39 25.553865 25.553865 25.553865
82 40 24.596748 24.596748 24.596748
82 41 29.154759 29.154759 29.154759
82 42 19.235384 19.235384 19.235384
82 43 22.803509 22.803509 22.803509
82 44 27.658633 27.658633 27.658633
82 45 24.738634 24.738634 24.738634
82 46 37.643060 37.643060 37.643060
82 47 38.013156 38.013156 38.013156
82 48 48.764741 48.764741 48.764741
82 49 53.460266 53.460266 53.460266
82 50 46.529560 46.529560 46.529560
82 51 32.526912 32.526912 32.526912
82 52 38.470768 38.470768 38.470768
82 53 36.878178 36.878178 36.878178
82 54 30.083218 30.083218 30.083218
82 55 6.324555 6.324555 6.324555
82 56 19.104973 19.104973 19.104973
82 57 23.021729 23.021729 23.021729
82 58 38.078866 38.078866 38.078866
82 59 58.821765 58.821765 58.821765
82 60 54.451814 54.451814 54.451814
82 61 34.058773 34.058773 34.058773
82 62 8.062258 8.062258 8.062258
82 63 28.017851 28.017851 28.017851
82 64 41.231056 41.231056 41.231056
82 65 28.284271 28.284271 28.284271
82 66 22.803509 22.803509 22.803509
82 67 22.472205 22.472205 22.472205
82 68 21.931712 21.931712 21.931712
82 69 9.219544 9.219544 9.219544
82 70 18.973666 18.973666 18.973666
82 71 17.804494 17.804494 17.804494
82 72 16.278821 16.278821 16.278821
82 73 15.652476 15.652476 15.652476
82 74 47.042534 47.042534 47.042534
82 75 47.801674 47.801674 47.801674
82 76 68.883960 68.883960 68.883960
82 77 47.296934 47.296934 47.296934
82 78 60.835845 60.835845 60.835845
82 79 41.048752 41.048752 41.048752
82 80 44.147480 44.147480 44.147480
82 81 11.180340 11.180340 11.180340
82 83 26.627054 26.627054 26.627054
82 84 29.546573 29.546573 29.546573
82 85 30.083218 30.083218 30.083218
82 86 37.696154 37.696154 37.696154
82 87 44.045431 44.045431 44.045431
82 88 50.249378 50.249378 50.249378
82 89 25.000000 25.000000 25.000000
82 90 55.973208 55.973208 55.973208
82 91 16.278821 16.278821 16.278821
82 92 16.000000 16.000000 16.000000
82 93 15.524175 15.524175 15.524175
82 94 13.416408 13.416408 13.416408
82 95 12.806248 12.806248 12.806248
82 96 22.135944 22.135944 22.135944
82 97 7.211103 7.211103 7.211103
82 98 60.207973 60.207973 60.207973
82 99 23.769729 23.769729 23.769729
82 100 32.526912 32.526912 32.526912
82 101 20.124612 20.124612 20.124612
83 1 14.764823 14.764823 14.764823
83 2 42.047592 42.047592 42.047592
83 3 32.388269 32.388269 32.388269
83 4 42.296572 42.296572 42.296572
83 5 37.656341 37.656341 37.656341
83 6 42.579338 42.579338 42.579338

83 7 33.241540 33.241540 33.241540
83 8 34.176015 34.176015 34.176015
83 9 38.897301 38.897301 38.897301
83 10 18.788294 18.788294 18.788294
83 11 17.262677 17.262677 17.262677
83 12 19.235384 19.235384 19.235384
83 13 19.104973 19.104973 19.104973
83 14 23.409400 23.409400 23.409400
83 15 22.090722 22.090722 22.090722
83 16 25.179357 25.179357 25.179357
83 17 27.166155 27.166155 27.166155
83 18 27.073973 27.073973 27.073973
83 19 41.629317 41.629317 41.629317
83 20 36.249138 36.249138 36.249138
83 21 31.764760 31.764760 31.764760
83 22 40.162171 40.162171 40.162171
83 23 30.870698 30.870698 30.870698
83 24 39.560081 39.560081 39.560081
83 25 30.083218 30.083218 30.083218
83 26 38.832976 38.832976 38.832976
83 27 69.231496 69.231496 69.231496
83 28 68.468971 68.468971 68.468971
83 29 66.287254 66.287254 66.287254
83 30 63.505905 63.505905 63.505905
83 31 62.369865 62.369865 62.369865
83 32 61.522354 61.522354 61.522354
83 33 61.392182 61.392182 61.392182
83 34 60.728906 60.728906 60.728906
83 35 58.549125 58.549125 58.549125
83 36 58.000000 58.000000 58.000000
83 37 56.639209 56.639209 56.639209
83 38 54.451814 54.451814 54.451814
83 39 50.931326 50.931326 50.931326
83 40 49.578221 49.578221 49.578221
83 41 53.413481 53.413481 53.413481
83 42 44.553339 44.553339 44.553339
83 43 46.400431 46.400431 46.400431
83 44 50.477718 50.477718 50.477718
83 45 48.010416 48.010416 48.010416
83 46 39.623226 39.623226 39.623226
83 47 38.078866 38.078866 38.078866
83 48 25.079872 25.079872 25.079872
83 49 40.853396 40.853396 40.853396
83 50 34.438351 34.438351 34.438351
83 51 45.705580 45.705580 45.705580
83 52 36.235342 36.235342 36.235342
83 53 13.152946 13.152946 13.152946
83 54 9.899495 9.899495 9.899495
83 55 32.756679 32.756679 32.756679
83 56 17.262677 17.262677 17.262677
83 57 24.351591 24.351591 24.351591
83 58 18.248288 18.248288 18.248288
83 59 35.114100 35.114100 35.114100
83 60 28.600699 28.600699 28.600699
83 61 20.808652 20.808652 20.808652
83 62 28.425341 28.425341 28.425341
83 63 38.832976 38.832976 38.832976
83 64 44.418465 44.418465 44.418465
83 65 22.203603 22.203603 22.203603
83 66 8.544004 8.544004 8.544004
83 67 15.231546 15.231546 15.231546
83 68 37.013511 37.013511 37.013511
83 69 21.400935 21.400935 21.400935
83 70 9.848858 9.848858 9.848858
83 71 27.202941 27.202941 27.202941
83 72 39.849718 39.849718 39.849718
83 73 42.190046 42.190046 42.190046
83 74 30.232433 30.232433 30.232433
83 75 24.041631 24.041631 24.041631
83 76 43.908997 43.908997 43.908997
83 77 45.276926 45.276926 45.276926
83 78 40.199502 40.199502 40.199502
83 79 23.021729 23.021729 23.021729
83 80 32.649655 32.649655 32.649655
83 81 20.396078 20.396078 20.396078
83 82 26.627054 26.627054 26.627054
83 84 15.620499 15.620499 15.620499
83 85 33.105891 33.105891 33.105891
83 86 41.182521 41.182521 41.182521
83 87 19.924859 19.924859 19.924859
83 88 24.207437 24.207437 24.207437
83 89 15.297059 15.297059 15.297059
83 90 55.172457 55.172457 55.172457
83 91 10.770330 10.770330 10.770330
83 92 22.022716 22.022716 22.022716
83 93 26.000000 26.000000 26.000000
83 94 35.171011 35.171011 35.171011
83 95 30.413813 30.413813 30.413813
83 96 29.614186 29.614186 29.614186
83 97 30.083218 30.083218 30.083218
83 98 33.970576 33.970576 33.970576
83 99 9.055385 9.055385 9.055385
83 100 8.062258 8.062258 8.062258
83 101 24.331050 24.331050 24.331050
84 1 19.235384 19.235384 19.235384

84 2 55.317267 55.317267 55.317267
84 3 46.486557 46.486557 46.486557
84 4 56.044625 56.044625 56.044625
84 5 51.865210 51.865210 51.865210
84 6 56.612719 56.612719 56.612719
84 7 47.927028 47.927028 47.927028
84 8 49.193496 49.193496 49.193496
84 9 53.712196 53.712196 53.712196
84 10 27.294688 27.294688 27.294688
84 11 28.460499 28.460499 28.460499
84 12 30.364453 30.364453 30.364453
84 13 32.202484 32.202484 32.202484
84 14 32.249031 32.249031 32.249031
84 15 34.928498 34.928498 34.928498
84 16 36.138622 36.138622 36.138622
84 17 38.078866 38.078866 38.078866
84 18 39.560081 39.560081 39.560081
84 19 26.925824 26.925824 26.925824
84 20 21.587033 21.587033 21.587033
84 21 16.763055 16.763055 16.763055
84 22 26.172505 26.172505 26.172505
84 23 16.278821 16.278821 16.278821
84 24 26.019224 26.019224 26.019224
84 25 16.031220 16.031220 16.031220
84 26 26.076810 26.076810 26.076810
84 27 58.008620 58.008620 58.008620
84 28 58.137767 58.137767 58.137767
84 29 55.009090 55.009090 55.009090
84 30 53.150729 53.150729 53.150729
84 31 51.009803 51.009803 51.009803
84 32 51.156622 51.156622 51.156622
84 33 50.009999 50.009999 50.009999
84 34 48.373546 48.373546 48.373546
84 35 48.166378 48.166378 48.166378
84 36 61.773781 61.773781 61.773781
84 37 60.827625 60.827625 60.827625
84 38 58.180753 58.180753 58.180753
84 39 55.009090 55.009090 55.009090
84 40 54.129474 54.129474 54.129474
84 41 58.694122 58.694122 58.694122
84 42 48.754487 48.754487 48.754487
84 43 52.201533 52.201533 52.201533
84 44 56.920998 56.920998 56.920998
84 45 54.083269 54.083269 54.083269
84 46 53.758720 53.758720 53.758720
84 47 52.554733 52.554733 52.554733
84 48 37.696154 37.696154 37.696154
84 49 26.476405 26.476405 26.476405
84 50 19.646883 19.646883 19.646883
84 51 35.227830 35.227830 35.227830
84 52 21.095023 21.095023 21.095023
84 53 12.041595 12.041595 12.041595
84 54 25.495098 25.495098 25.495098
84 55 34.132096 34.132096 34.132096
84 56 29.832868 29.832868 29.832868
84 57 13.601471 13.601471 13.601471
84 58 9.219544 9.219544 9.219544
84 59 30.413813 30.413813 30.413813
84 60 29.154759 29.154759 29.154759
84 61 36.400549 36.400549 36.400549
84 62 34.928498 34.928498 34.928498
84 63 28.284271 28.284271 28.284271
84 64 30.083218 30.083218 30.083218
84 65 8.062258 8.062258 8.062258
84 66 9.219544 9.219544 9.219544
84 67 7.211103 7.211103 7.211103
84 68 29.154759 29.154759 29.154759
84 69 29.154759 29.154759 29.154759
84 70 21.840330 21.840330 21.840330
84 71 38.052595 38.052595 38.052595
84 72 36.878178 36.878178 36.878178
84 73 42.801869 42.801869 42.801869
84 74 45.453273 45.453273 45.453273
84 75 20.248457 20.248457 20.248457
84 76 41.231056 41.231056 41.231056
84 77 29.832868 29.832868 29.832868
84 78 31.304952 31.304952 31.304952
84 79 38.288379 38.288379 38.288379
84 80 48.270074 48.270074 48.270074
84 81 18.867962 18.867962 18.867962
84 82 29.546573 29.546573 29.546573
84 83 15.620499 15.620499 15.620499
84 85 20.099751 20.099751 20.099751
84 86 27.202941 27.202941 27.202941
84 87 17.464249 17.464249 17.464249
84 88 25.961510 25.961510 25.961510
84 89 29.966648 29.966648 29.966648
84 90 39.698866 39.698866 39.698866
84 91 16.000000 16.000000 16.000000
84 92 16.278821 16.278821 16.278821
84 93 20.000000 20.000000 20.000000
84 94 31.890437 31.890437 31.890437
84 95 26.248809 26.248809 26.248809
84 96 19.924859 19.924859 19.924859
84 97 29.206164 29.206164 29.206164

84 98 35.468296 35.468296 35.468296
84 99 23.706539 23.706539 23.706539
84 100 11.704700 11.704700 11.704700
84 101 36.496575 36.496575 36.496575
85 1 27.018512 27.018512 27.018512
85 2 64.498062 64.498062 64.498062
85 3 57.801384 57.801384 57.801384
85 4 66.037868 66.037868 66.037868
85 5 63.007936 63.007936 63.007936
85 6 67.119297 67.119297 67.119297
85 7 60.307545 60.307545 60.307545
85 8 62.289646 62.289646 62.289646
85 9 66.068147 66.068147 66.068147
85 10 47.381431 47.381431 47.381431
85 11 48.270074 48.270074 48.270074
85 12 50.219518 50.219518 50.219518
85 13 51.546096 51.546096 51.546096
85 14 52.345009 52.345009 52.345009
85 15 54.405882 54.405882 54.405882
85 16 56.089215 56.089215 56.089215
85 17 58.051701 58.051701 58.051701
85 18 59.203040 59.203040 59.203040
85 19 27.294688 27.294688 27.294688
85 20 24.207437 24.207437 24.207437
85 21 20.518285 20.518285 20.518285
85 22 29.410882 29.410882 29.410882
85 23 22.022716 22.022716 22.022716
85 24 30.610456 30.610456 30.610456
85 25 23.600847 23.600847 23.600847
85 26 32.557641 32.557641 32.557641
85 27 38.013156 38.013156 38.013156
85 28 38.470768 38.470768 38.470768
85 29 35.014283 35.014283 35.014283
85 30 33.541020 33.541020 33.541020
85 31 31.016125 31.016125 31.016125
85 32 31.575307 31.575307 31.575307
85 33 30.016662 30.016662 30.016662
85 34 28.284271 28.284271 28.284271
85 35 28.635642 28.635642 28.635642
85 36 56.885851 56.885851 56.885851
85 37 56.568542 56.568542 56.568542
85 38 53.600373 53.600373 53.600373
85 39 51.244512 51.244512 51.244512
85 40 51.088159 51.088159 51.088159
85 41 56.080300 56.080300 56.080300
85 42 46.010868 46.010868 46.010868
85 43 51.039201 51.039201 51.039201
85 44 56.035703 56.035703 56.035703
85 45 53.037722 53.037722 53.037722
85 46 64.637450 64.637450 64.637450
85 47 64.202804 64.202804 64.202804
85 48 57.280014 57.280014 57.280014
85 49 28.301943 28.301943 28.301943
85 50 22.671568 22.671568 22.671568
85 51 16.155494 16.155494 16.155494
85 52 9.219544 9.219544 9.219544
85 53 32.015621 32.015621 32.015621
85 54 42.544095 42.544095 42.544095
85 55 31.064449 31.064449 31.064449
85 56 41.109610 41.109610 41.109610
85 57 9.219544 9.219544 9.219544
85 58 27.294688 27.294688 27.294688
85 59 46.097722 46.097722 46.097722
85 60 47.853944 47.853944 47.853944
85 61 52.201533 52.201533 52.201533
85 62 37.947332 37.947332 37.947332
85 63 10.000000 10.000000 10.000000
85 64 12.041595 12.041595 12.041595
85 65 12.041595 12.041595 12.041595
85 66 24.596748 24.596748 24.596748
85 67 17.888544 17.888544 17.888544
85 68 14.764823 14.764823 14.764823
85 69 35.355339 35.355339 35.355339
85 70 34.713110 34.713110 34.713110
85 71 45.650849 45.650849 45.650849
85 72 27.202941 27.202941 27.202941
85 73 36.496575 36.496575 36.496575
85 74 63.134776 63.134776 63.134776
85 75 38.078866 38.078866 38.078866
85 76 57.271284 57.271284 57.271284
85 77 17.262677 17.262677 17.262677
85 78 42.801869 42.801869 42.801869
85 79 55.946403 55.946403 55.946403
85 80 64.202804 64.202804 64.202804
85 81 20.591260 20.591260 20.591260
85 82 30.083218 30.083218 30.083218
85 83 33.105891 33.105891 33.105891
85 84 20.099751 20.099751 20.099751
85 86 8.485281 8.485281 8.485281
85 87 36.345564 36.345564 36.345564
85 88 45.276926 45.276926 45.276926
85 89 43.931765 43.931765 43.931765
85 90 26.000000 26.000000 26.000000
85 91 26.907248 26.907248 26.907248
85 92 15.264338 15.264338 15.264338

85 93 14.560220 14.560220 14.560220
85 94 23.345235 23.345235 23.345235
85 95 19.000000 19.000000 19.000000
85 96 8.062258 8.062258 8.062258
85 97 25.079872 25.079872 25.079872
85 98 54.129474 54.129474 54.129474
85 99 38.600518 38.600518 38.600518
85 100 31.575307 31.575307 31.575307
85 101 46.043458 46.043458 46.043458
86 1 35.468296 35.468296 35.468296
86 2 72.718636 72.718636 72.718636
86 3 66.219333 66.219333 66.219333
86 4 74.330344 74.330344 74.330344
86 5 71.400280 71.400280 71.400280
86 6 75.451971 75.451971 75.451971
86 7 68.767725 68.767725 68.767725
86 8 70.767224 70.767224 70.767224
86 9 74.518454 74.518454 74.518454
86 10 54.341513 54.341513 54.341513
86 11 55.659680 55.659680 55.659680
86 12 57.567352 57.567352 57.567352
86 13 59.236813 59.236813 59.236813
86 14 59.228372 59.228372 59.228372
86 15 62.032250 62.032250 62.032250
86 16 63.324561 63.324561 63.324561
86 17 65.253352 65.253352 65.253352
86 18 66.730802 66.730802 66.730802
86 19 26.172505 26.172505 26.172505
86 20 24.698178 24.698178 24.698178
86 21 22.472205 22.472205 22.472205
86 22 29.206164 29.206164 29.206164
86 23 24.351591 24.351591 24.351591
86 24 30.805844 30.805844 30.805844
86 25 26.248809 26.248809 26.248809
86 26 33.286634 33.286634 33.286634
86 27 32.756679 32.756679 32.756679
86 28 34.176015 34.176015 34.176015
86 29 29.832868 29.832868 29.832868
86 30 29.546573 29.546573 29.546573
86 31 25.961510 25.961510 25.961510
86 32 27.730849 27.730849 27.730849
86 33 25.000000 25.000000 25.000000
86 34 22.090722 22.090722 22.090722
86 35 25.059928 25.059928 25.059928
86 36 62.128898 62.128898 62.128898
86 37 62.032250 62.032250 62.032250
86 38 59.033889 59.033889 59.033889
86 39 57.008771 57.008771 57.008771
86 40 57.078893 57.078893 57.078893
86 41 62.072538 62.072538 62.072538
86 42 52.239832 52.239832 52.239832
86 43 57.558666 57.558666 57.558666
86 44 62.513998 62.513998 62.513998
86 45 59.539903 59.539903 59.539903
86 46 73.006849 73.006849 73.006849
86 47 72.622311 72.622311 72.622311
86 48 64.845971 64.845971 64.845971
86 49 27.658633 27.658633 27.658633
86 50 23.706539 23.706539 23.706539
86 51 15.000000 15.000000 15.000000
86 52 8.544004 8.544004 8.544004
86 53 38.639358 38.639358 38.639358
86 54 50.774009 50.774009 50.774009
86 55 37.854986 37.854986 37.854986
86 56 49.578221 49.578221 49.578221
86 57 17.691806 17.691806 17.691806
86 58 33.060551 33.060551 33.060551
86 59 49.729267 49.729267 49.729267
86 60 53.084838 53.084838 53.084838
86 61 60.605280 60.605280 60.605280
86 62 45.694639 45.694639 45.694639
86 63 12.165525 12.165525 12.165525
86 64 3.605551 3.605551 3.605551
86 65 19.313208 19.313208 19.313208
86 66 32.756679 32.756679 32.756679
86 67 26.076810 26.076810 26.076810
86 68 19.026298 19.026298 19.026298
86 69 43.566042 43.566042 43.566042
86 70 43.185646 43.185646 43.185646
86 71 53.851648 53.851648 53.851648
86 72 32.062439 32.062439 32.062439
86 73 42.000000 42.000000 42.000000
86 74 71.344236 71.344236 71.344236
86 75 43.104524 43.104524 43.104524
86 76 60.728906 60.728906 60.728906
86 77 11.401754 11.401754 11.401754
86 78 44.721360 44.721360 44.721360
86 79 64.140471 64.140471 64.140471
86 80 72.622311 72.622311 72.622311
86 81 28.844410 28.844410 28.844410
86 82 37.696154 37.696154 37.696154
86 83 41.182521 41.182521 41.182521
86 84 27.202941 27.202941 27.202941
86 85 8.485281 8.485281 8.485281
86 87 42.011903 42.011903 42.011903

86 88 51.009803 51.009803 51.009803
86 89 52.402290 52.402290 52.402290
86 90 18.439089 18.439089 18.439089
86 91 35.383612 35.383612 35.383612
86 92 23.600847 23.600847 23.600847
86 93 22.360680 22.360680 22.360680
86 94 29.068884 29.068884 29.068884
86 95 25.709920 25.709920 25.709920
86 96 15.652476 15.652476 15.652476
86 97 32.015621 32.015621 32.015621
86 98 59.211485 59.211485 59.211485
86 99 47.010637 47.010637 47.010637
86 100 38.897301 38.897301 38.897301
86 101 54.405882 54.405882 54.405882
87 1 32.202484 32.202484 32.202484
87 2 61.131007 61.131007 61.131007
87 3 51.009803 51.009803 51.009803
87 4 61.008196 61.008196 61.008196
87 5 56.008928 56.008928 56.008928
87 6 61.008196 61.008196 61.008196
87 7 51.088159 51.088159 51.088159
87 8 51.351728 51.351728 51.351728
87 9 56.320511 56.320511 56.320511
87 10 15.556349 15.556349 15.556349
87 11 19.416488 19.416488 19.416488
87 12 20.615528 20.615528 20.615528
87 13 24.698178 24.698178 24.698178
87 14 19.416488 19.416488 19.416488
87 15 26.400758 26.400758 26.400758
87 16 24.839485 24.839485 24.839485
87 17 26.400758 26.400758 26.400758
87 18 29.698485 29.698485 29.698485
87 19 29.832868 29.832868 29.832868
87 20 25.238859 25.238859 25.238859
87 21 22.847319 22.847319 22.847319
87 22 26.870058 26.870058 26.870058
87 23 21.023796 21.023796 21.023796
87 24 25.495098 25.495098 25.495098
87 25 19.235384 19.235384 19.235384
87 26 23.600847 23.600847 23.600847
87 27 74.242845 74.242845 74.242845
87 28 74.813100 74.813100 74.813100
87 29 71.253070 71.253070 71.253070
87 30 69.871310 69.871310 69.871310
87 31 67.268120 67.268120 67.268120
87 32 67.896981 67.896981 67.896981
87 33 66.272166 66.272166 66.272166
87 34 64.007812 64.007812 64.007812
87 35 64.938432 64.938432 64.938432
87 36 76.400262 76.400262 76.400262
87 37 75.213031 75.213031 75.213031
87 38 72.801099 72.801099 72.801099
87 39 69.404611 69.404611 69.404611
87 40 68.242216 68.242216 68.242216
87 41 72.401657 72.401657 72.401657
87 42 63.007936 63.007936 63.007936
87 43 65.513357 65.513357 65.513357
87 44 69.835521 69.835521 69.835521
87 45 67.230945 67.230945 67.230945
87 46 58.008620 58.008620 58.008620
87 47 56.080300 56.080300 56.080300
87 48 28.319605 28.319605 28.319605
87 49 28.319605 28.319605 28.319605
87 50 24.186773 24.186773 24.186773
87 51 52.172790 52.172790 52.172790
87 52 34.234486 34.234486 34.234486
87 53 7.211103 7.211103 7.211103
87 54 26.019224 26.019224 26.019224
87 55 49.517674 49.517674 49.517674
87 56 37.107951 37.107951 37.107951
87 57 31.016125 31.016125 31.016125
87 58 9.055385 9.055385 9.055385
87 59 15.231546 15.231546 15.231546
87 60 11.704700 11.704700 11.704700
87 61 36.496575 36.496575 36.496575
87 62 47.507894 47.507894 47.507894
87 63 45.354162 45.354162 45.354162
87 64 44.181444 44.181444 44.181444
87 65 24.738634 24.738634 24.738634
87 66 21.260292 21.260292 21.260292
87 67 23.853721 23.853721 23.853721
87 68 46.615448 46.615448 46.615448
87 69 40.706265 40.706265 40.706265
87 70 29.732137 29.732137 29.732137
87 71 47.127487 47.127487 47.127487
87 72 53.823787 53.823787 53.823787
87 73 58.694122 58.694122 58.694122
87 74 40.706265 40.706265 40.706265
87 75 4.123106 4.123106 4.123106
87 76 24.839485 24.839485 24.839485
87 77 40.804412 40.804412 40.804412
87 78 21.095023 21.095023 21.095023
87 79 34.539832 34.539832 34.539832
87 80 46.486557 46.486557 46.486557
87 81 34.713110 34.713110 34.713110

87 82 44.045431 44.045431 44.045431
87 83 19.924859 19.924859 19.924859
87 84 17.464249 17.464249 17.464249
87 85 36.345564 36.345564 36.345564
87 86 42.011903 42.011903 42.011903
87 88 9.000000 9.000000 9.000000
87 89 34.132096 34.132096 34.132096
87 90 49.769469 49.769469 49.769469
87 91 28.017851 28.017851 28.017851
87 92 33.286634 33.286634 33.286634
87 93 37.215588 37.215588 37.215588
87 94 48.826222 48.826222 48.826222
87 95 43.266615 43.266615 43.266615
87 96 37.336309 37.336309 37.336309
87 97 45.343136 45.343136 45.343136
87 98 18.027756 18.027756 18.027756
87 99 28.442925 28.442925 28.442925
87 100 12.083046 12.083046 12.083046
87 101 44.147480 44.147480 44.147480
88 1 38.209946 38.209946 38.209946
88 2 62.369865 62.369865 62.369865
88 3 51.971146 51.971146 51.971146
88 4 61.814238 61.814238 61.814238
88 5 56.568542 56.568542 56.568542
88 6 61.522354 61.522354 61.522354
88 7 51.351728 51.351728 51.351728
88 8 51.088159 51.088159 51.088159
88 9 56.080300 56.080300 56.080300
88 10 11.180340 11.180340 11.180340
88 11 16.124515 16.124515 16.124515
88 12 16.492423 16.492423 16.492423
88 13 21.377558 21.377558 21.377558
88 14 13.038405 13.038405 13.038405
88 15 22.135944 22.135944 22.135944
88 16 18.867962 18.867962 18.867962
88 17 20.000000 20.000000 20.000000
88 18 24.186773 24.186773 24.186773
88 19 37.215588 37.215588 37.215588
88 20 33.105891 33.105891 33.105891
88 21 31.320920 31.320920 31.320920
88 22 33.837849 33.837849 33.837849
88 23 29.410882 29.410882 29.410882
88 24 32.202484 32.202484 32.202484
88 25 27.513633 27.513633 27.513633
88 26 29.832868 29.832868 29.832868
88 27 83.216585 83.216585 83.216585
88 28 83.725743 83.725743 83.725743
88 29 80.224684 80.224684 80.224684
88 30 78.771822 78.771822 78.771822
88 31 76.236474 76.236474 76.236474
88 32 76.791927 76.791927 76.791927
88 33 75.239617 75.239617 75.239617
88 34 73.006849 73.006849 73.006849
88 35 73.824115 73.824115 73.824115
88 36 82.134037 82.134037 82.134037
88 37 80.808415 80.808415 80.808415
88 38 78.568442 78.568442 78.568442
88 39 75.073298 75.073298 75.073298
88 40 73.756356 73.756356 73.756356
88 41 77.620873 77.620873 77.620873
88 42 68.680419 68.680419 68.680419
88 43 70.604532 70.604532 70.604532
88 44 74.632433 74.632433 74.632433
88 45 72.201108 72.201108 72.201108
88 46 58.549125 58.549125 58.549125
88 47 56.320511 56.320511 56.320511
88 48 23.259407 23.259407 23.259407
88 49 35.510562 35.510562 35.510562
88 50 32.310989 32.310989 32.310989
88 51 61.000000 61.000000 61.000000
88 52 43.185646 43.185646 43.185646
88 53 14.317821 14.317821 14.317821
88 54 27.202941 27.202941 27.202941
88 55 56.080300 56.080300 56.080300
88 56 40.249224 40.249224 40.249224
88 57 39.560081 39.560081 39.560081
88 58 18.027756 18.027756 18.027756
88 59 14.317821 14.317821 14.317821
88 60 4.472136 4.472136 4.472136
88 61 36.124784 36.124784 36.124784
88 62 52.630789 52.630789 52.630789
88 63 54.129474 54.129474 54.129474
88 64 53.150729 53.150729 53.150729
88 65 33.541020 33.541020 33.541020
88 66 28.017851 28.017851 28.017851
88 67 31.780497 31.780497 31.780497
88 68 55.027266 55.027266 55.027266
88 69 45.607017 45.607017 45.607017
88 70 33.837849 33.837849 33.837849
88 71 50.537115 50.537115 50.537115
88 72 61.400326 61.400326 61.400326
88 73 65.436993 65.436993 65.436993
88 74 37.363083 37.363083 37.363083
88 75 8.944272 8.944272 8.944272
88 76 20.248457 20.248457 20.248457

88 77 49.477268 49.477268 49.477268
88 78 23.706539 23.706539 23.706539
88 79 32.249031 32.249031 32.249031
88 80 44.407207 44.407207 44.407207
88 81 41.880783 41.880783 41.880783
88 82 50.249378 50.249378 50.249378
88 83 24.207437 24.207437 24.207437
88 84 25.961510 25.961510 25.961510
88 85 45.276926 45.276926 45.276926
88 86 51.009803 51.009803 51.009803
88 87 9.000000 9.000000 9.000000
88 89 36.055513 36.055513 36.055513
88 90 58.189346 58.189346 58.189346
88 91 33.970576 33.970576 33.970576
88 92 41.146081 41.146081 41.146081
88 93 45.188494 45.188494 45.188494
88 94 56.435804 56.435804 56.435804
88 95 51.000000 51.000000 51.000000
88 96 45.880279 45.880279 45.880279
88 97 52.430907 52.430907 52.430907
88 98 10.000000 10.000000 10.000000
88 99 31.304952 31.304952 31.304952
88 100 17.804494 17.804494 17.804494
88 101 47.010637 47.010637 47.010637
89 1 17.888544 17.888544 17.888544
89 2 27.018512 27.018512 27.018512
89 3 17.117243 17.117243 17.117243
89 4 27.073973 27.073973 27.073973
89 5 22.360680 22.360680 22.360680
89 6 27.294688 27.294688 27.294688
89 7 18.027756 18.027756 18.027756
89 8 19.235384 19.235384 19.235384
89 9 23.769729 23.769729 23.769729
89 10 26.925824 26.925824 26.925824
89 11 22.803509 22.803509 22.803509
89 12 24.083189 24.083189 24.083189
89 13 20.615528 20.615528 20.615528
89 14 29.832868 29.832868 29.832868
89 15 23.021729 23.021729 23.021729
89 16 28.425341 28.425341 28.425341
89 17 30.000000 30.000000 30.000000
89 18 27.294688 27.294688 27.294688
89 19 56.648036 56.648036 56.648036
89 20 51.264022 51.264022 51.264022
89 21 46.615448 46.615448 46.615448
89 22 55.362442 55.362442 55.362442
89 23 45.880279 45.880279 45.880279
89 24 54.817880 54.817880 54.817880
89 25 45.221676 45.221676 45.221676
89 26 54.129474 54.129474 54.129474
89 27 76.321688 76.321688 76.321688
89 28 74.632433 74.632433 74.632433
89 29 73.539105 73.539105 73.539105
89 30 69.892775 69.892775 69.892775
89 31 69.856997 69.856997 69.856997
89 32 68.007353 68.007353 68.007353
89 33 68.942005 68.942005 68.942005
89 34 69.354164 69.354164 69.354164
89 35 65.192024 65.192024 65.192024
89 36 50.774009 50.774009 50.774009
89 37 49.091751 49.091751 49.091751
89 38 47.507894 47.507894 47.507894
89 39 43.908997 43.908997 43.908997
89 40 42.190046 42.190046 42.190046
89 41 45.000000 45.000000 45.000000
89 42 38.013156 38.013156 38.013156
89 43 38.013156 38.013156 38.013156
89 44 41.109610 41.109610 41.109610
89 45 39.204592 39.204592 39.204592
89 46 24.331050 24.331050 24.331050
89 47 22.803509 22.803509 22.803509
89 48 25.553865 25.553865 25.553865
89 49 55.973208 55.973208 55.973208
89 50 49.396356 49.396356 49.396356
89 51 53.225934 53.225934 53.225934
89 52 49.040799 49.040799 49.040799
89 53 28.017851 28.017851 28.017851
89 54 8.944272 8.944272 8.944272
89 55 31.064449 31.064449 31.064449
89 56 6.324555 6.324555 6.324555
89 57 34.713110 34.713110 34.713110
89 58 33.541020 33.541020 33.541020
89 59 48.836462 48.836462 48.836462
89 60 40.496913 40.496913 40.496913
89 61 9.219544 9.219544 9.219544
89 62 22.135944 22.135944 22.135944
89 63 47.010637 47.010637 47.010637
89 64 55.901699 55.901699 55.901699
89 65 35.000000 35.000000 35.000000
89 66 21.095023 21.095023 21.095023
89 67 27.018512 27.018512 27.018512
89 68 43.081318 43.081318 43.081318
89 69 16.124515 16.124515 16.124515
89 70 9.219544 9.219544 9.219544
89 71 15.556349 15.556349 15.556349
89 72 41.109610 41.109610 41.109610

89 72 41.109610 41.109610 41.109610
89 73 39.623226 39.623226 39.623226
89 74 22.090722 22.090722 22.090722
89 75 38.209946 38.209946 38.209946
89 76 56.302753 56.302753 56.302753
89 77 58.412327 58.412327 58.412327
89 78 55.009090 55.009090 55.009090
89 79 16.124515 16.124515 16.124515
89 80 20.591260 20.591260 20.591260
89 81 25.495098 25.495098 25.495098
89 82 25.000000 25.000000 25.000000
89 83 15.297059 15.297059 15.297059
89 84 29.966648 29.966648 29.966648
89 85 43.931765 43.931765 43.931765
89 86 52.402290 52.402290 52.402290
89 87 34.132096 34.132096 34.132096
89 88 36.055513 36.055513 36.055513
89 90 68.249542 68.249542 68.249542
89 91 17.029386 17.029386 17.029386
89 92 29.681644 29.681644 29.681644
89 93 32.649655 32.649655 32.649655
89 94 37.483330 37.483330 37.483330
89 95 34.481879 34.481879 34.481879
89 96 38.275318 38.275318 38.275318
89 97 31.256999 31.256999 31.256999
89 98 44.721360 44.721360 44.721360
89 99 6.324555 6.324555 6.324555
89 100 23.086793 23.086793 23.086793
89 101 11.401754 11.401754 11.401754
90 1 52.478567 52.478567 52.478567
90 2 90.354856 90.354856 90.354856
90 3 83.216585 83.216585 83.216585
90 4 91.787799 91.787799 91.787799
90 5 88.509886 88.509886 88.509886
90 6 92.784697 92.784697 92.784697
90 7 85.445889 85.445889 85.445889
90 8 87.200917 87.200917 87.200917
90 9 91.263355 91.263355 91.263355
90 10 64.412732 64.412732 64.412732
90 11 66.887966 66.887966 66.887966
90 12 68.600292 68.600292 68.600292
90 13 71.281134 71.281134 71.281134
90 14 68.876701 68.876701 68.876701
90 15 73.783467 73.783467 73.783467
90 16 73.824115 73.824115 73.824115
90 17 75.591005 75.591005 75.591005
90 18 78.032045 78.032045 78.032045
90 19 23.000000 23.000000 23.000000
90 20 25.495098 25.495098 25.495098
90 21 26.925824 26.925824 26.925824
90 22 27.000000 27.000000 27.000000
90 23 28.792360 28.792360 28.792360
90 24 29.000000 29.000000 29.000000
90 25 30.675723 30.675723 30.675723
90 26 32.000000 32.000000 32.000000
90 27 37.536649 37.536649 37.536649
90 28 41.036569 41.036569 41.036569
90 29 35.355339 35.355339 35.355339
90 30 37.802116 37.802116 37.802116
90 31 32.649655 32.649655 32.649655
90 32 36.619667 36.619667 36.619667
90 33 32.015621 32.015621 32.015621
90 34 26.907248 26.907248 26.907248
90 35 34.985711 34.985711 34.985711
90 36 80.000000 80.000000 80.000000
90 37 80.024996 80.024996 80.024996
90 38 77.025970 77.025970 77.025970
90 39 75.166482 75.166482 75.166482
90 40 75.325958 75.325958 75.325958
90 41 80.305666 80.305666 80.305666
90 42 70.576200 70.576200 70.576200
90 43 75.953933 75.953933 75.953933
90 44 80.894994 80.894994 80.894994
90 45 77.929455 77.929455 77.929455
90 46 90.210864 90.210864 90.210864
90 47 89.587946 89.587946 89.587946
90 48 76.321688 76.321688 76.321688
90 49 25.000000 25.000000 25.000000
90 50 25.961510 25.961510 25.961510
90 51 30.413813 30.413813 30.413813
90 52 19.209373 19.209373 19.209373
90 53 48.877398 48.877398 48.877398
90 54 65.069194 65.069194 65.069194
90 55 56.293872 56.293872 56.293872
90 56 66.287254 66.287254 66.287254
90 57 34.481879 34.481879 34.481879
90 58 42.059482 42.059482 42.059482
90 59 52.239832 52.239832 52.239832
90 60 58.940648 58.940648 58.940648
90 61 75.690158 75.690158 75.690158
90 62 63.906181 63.906181 63.906181
90 63 30.066593 30.066593 30.066593
90 64 15.132746 15.132746 15.132746
90 65 33.301652 33.301652 33.301652
90 66 47.423623 47.423623 47.423623
90 67 41.231056 41.231056 41.231056

90 67 41.231056 41.231056 41.231056
90 68 37.121422 37.121422 37.121422
90 69 61.269895 61.269895 61.269895
90 70 59.203040 59.203040 59.203040
90 71 71.554175 71.554175 71.554175
90 72 50.039984 50.039984 50.039984
90 73 60.133186 60.133186 60.133186
90 74 85.146932 85.146932 85.146932
90 75 49.335586 49.335586 49.335586
90 76 62.000000 62.000000 62.000000
90 77 9.899495 9.899495 9.899495
90 78 44.045431 44.045431 44.045431
90 79 77.987178 77.987178 77.987178
90 80 87.692645 87.692645 87.692645
90 81 46.518813 46.518813 46.518813
90 82 55.973208 55.973208 55.973208
90 83 55.172457 55.172457 55.172457
90 84 39.698866 39.698866 39.698866
90 85 26.000000 26.000000 26.000000
90 86 18.439089 18.439089 18.439089
90 87 49.769469 49.769469 49.769469
90 88 58.189346 58.189346 58.189346
90 89 68.249542 68.249542 68.249542
90 91 51.613952 51.613952 51.613952
90 92 41.146081 41.146081 41.146081
90 93 40.496913 40.496913 40.496913
90 94 47.381431 47.381431 47.381431
90 95 44.147480 44.147480 44.147480
90 96 33.837849 33.837849 33.837849
90 97 50.447993 50.447993 50.447993
90 98 64.327288 64.327288 64.327288
90 99 62.369865 62.369865 62.369865
90 100 50.803543 50.803543 50.803543
90 101 71.693793 71.693793 71.693793
91 1 4.242641 4.242641 4.242641
91 2 39.849718 39.849718 39.849718
91 3 31.764760 31.764760 31.764760
91 4 40.853396 40.853396 40.853396
91 5 37.121422 37.121422 37.121422
91 6 41.629317 41.629317 41.629317
91 7 33.837849 33.837849 33.837849
91 8 35.608988 35.608988 35.608988
91 9 39.661064 39.661064 39.661064
91 10 29.546573 29.546573 29.546573
91 11 27.892651 27.892651 27.892651
91 12 29.832868 29.832868 29.832868
91 13 29.068884 29.068884 29.068884
91 14 34.176015 34.176015 34.176015
91 15 32.062439 32.062439 32.062439
91 16 35.693137 35.693137 35.693137
91 17 37.656341 37.656341 37.656341
91 18 37.054015 37.054015 37.054015
91 19 42.579338 42.579338 42.579338
91 20 37.336309 37.336309 37.336309
91 21 32.388269 32.388269 32.388269
91 22 42.107007 42.107007 42.107007
91 23 32.140317 32.140317 32.140317
91 24 42.011903 42.011903 42.011903
91 25 32.015621 32.015621 32.015621
91 26 42.047592 42.047592 42.047592
91 27 60.440053 60.440053 60.440053
91 28 59.228372 59.228372 59.228372
91 29 57.567352 57.567352 57.567352
91 30 54.341513 54.341513 54.341513
91 31 53.758720 53.758720 53.758720
91 32 52.392748 52.392748 52.392748
91 33 52.810984 52.810984 52.810984
91 34 52.801515 52.801515 52.801515
91 35 49.477268 49.477268 49.477268
91 36 48.414874 48.414874 48.414874
91 37 47.201695 47.201695 47.201695
91 38 44.821870 44.821870 44.821870
91 39 41.400483 41.400483 41.400483
91 40 40.224371 40.224371 40.224371
91 41 44.418465 44.418465 44.418465
91 42 35.000000 35.000000 35.000000
91 43 37.589892 37.589892 37.589892
91 44 42.047592 42.047592 42.047592
91 45 39.357337 39.357337 39.357337
91 46 38.910153 38.910153 38.910153
91 47 38.078866 38.078866 38.078866
91 48 35.057096 35.057096 35.057096
91 49 42.296572 42.296572 42.296572
91 50 35.355339 35.355339 35.355339
91 51 37.000000 37.000000 37.000000
91 52 32.449961 32.449961 32.449961
91 53 20.808652 20.808652 20.808652
91 54 17.262677 17.262677 17.262677
91 55 22.203603 22.203603 22.203603
91 56 14.764823 14.764823 14.764823
91 57 17.691806 17.691806 17.691806
91 58 23.086793 23.086793 23.086793
91 59 43.046487 43.046487 43.046487
91 60 38.183766 38.183766 38.183766
91 61 25.553865 25.553865 25.553865
91 62 19.697716 19.697716 19.697716

91 63 30.463092 30.463092 30.463092
91 64 38.897301 38.897301 38.897301
91 65 18.788294 18.788294 18.788294
91 66 7.280110 7.280110 7.280110
91 67 10.770330 10.770330 10.770330
91 68 27.459060 27.459060 27.459060
91 69 13.341664 13.341664 13.341664
91 70 7.810250 7.810250 7.810250
91 71 22.090722 22.090722 22.090722
91 72 29.120440 29.120440 29.120440
91 73 31.622777 31.622777 31.622777
91 74 37.336309 37.336309 37.336309
91 75 31.906112 31.906112 31.906112
91 76 52.801515 52.801515 52.801515
91 77 41.880783 41.880783 41.880783
91 78 46.173586 46.173586 46.173586
91 79 30.364453 30.364453 30.364453
91 80 37.443290 37.443290 37.443290
91 81 10.000000 10.000000 10.000000
91 82 16.278821 16.278821 16.278821
91 83 10.770330 10.770330 10.770330
91 84 16.000000 16.000000 16.000000
91 85 26.907248 26.907248 26.907248
91 86 35.383612 35.383612 35.383612
91 87 28.017851 28.017851 28.017851
91 88 33.970576 33.970576 33.970576
91 89 17.029386 17.029386 17.029386
91 90 51.613952 51.613952 51.613952
91 92 13.000000 13.000000 13.000000
91 93 16.492423 16.492423 16.492423
91 94 24.515301 24.515301 24.515301
91 95 20.024984 20.024984 20.024984
91 96 21.470911 21.470911 21.470911
91 97 19.313208 19.313208 19.313208
91 98 43.931765 43.931765 43.931765
91 99 12.083046 12.083046 12.083046
91 100 16.278821 16.278821 16.278821
91 101 20.880613 20.880613 20.880613
92 1 12.041595 12.041595 12.041595
92 2 49.244289 49.244289 49.244289
92 3 42.638011 42.638011 42.638011
92 4 50.774009 50.774009 50.774009
92 5 47.801674 47.801674 47.801674
92 6 51.865210 51.865210 51.865210
92 7 45.276926 45.276926 45.276926
92 8 47.381431 47.381431 47.381431
92 9 50.990195 50.990195 50.990195
92 10 39.623226 39.623226 39.623226
92 11 39.051248 39.051248 39.051248
92 12 41.048752 41.048752 41.048752
92 13 41.109610 41.109610 41.109610
92 14 44.553339 44.553339 44.553339
92 15 44.102154 44.102154 44.102154
92 16 47.042534 47.042534 47.042534
92 17 49.040799 49.040799 49.040799
92 18 49.091751 49.091751 49.091751
92 19 37.336309 37.336309 37.336309
92 20 32.756679 32.756679 32.756679
92 21 27.892651 27.892651 27.892651
92 22 38.078866 38.078866 38.078866
92 23 28.460499 28.460499 28.460499
92 24 38.600518 38.600518 38.600518
92 25 29.154759 29.154759 29.154759
92 26 39.560081 39.560081 39.560081
92 27 47.539457 47.539457 47.539457
92 28 46.529560 46.529560 46.529560
92 29 44.643029 44.643029 44.643029
92 30 41.593269 41.593269 41.593269
92 31 40.804412 40.804412 40.804412
92 32 39.623226 39.623226 39.623226
92 33 39.849718 39.849718 39.849718
92 34 39.812058 39.812058 39.812058
92 35 36.674242 36.674242 36.674242
92 36 46.615448 46.615448 46.615448
92 37 45.880279 45.880279 45.880279
92 38 43.081318 43.081318 43.081318
92 39 40.162171 40.162171 40.162171
92 40 39.560081 39.560081 39.560081
92 41 44.384682 44.384682 44.384682
92 42 34.205263 34.205263 34.205263
92 43 38.470768 38.470768 38.470768
92 44 43.416587 43.416587 43.416587
92 45 40.447497 40.447497 40.447497
92 46 49.406477 49.406477 49.406477
92 47 49.040799 49.040799 49.040799
92 48 47.095647 47.095647 47.095647
92 49 37.656341 37.656341 37.656341
92 50 30.805844 30.805844 30.805844
92 51 24.041631 24.041631 24.041631
92 52 22.803509 22.803509 22.803509
92 53 26.832816 26.832816 26.832816
92 54 30.083218 30.083218 30.083218
92 55 18.973666 18.973666 18.973666
92 56 26.172505 26.172505 26.172505
92 57 7.071068 7.071068 7.071068

92 58 25.495098 25.495098 25.495098
92 59 46.690470 46.690470 46.690470
92 60 44.777226 44.777226 44.777226
92 61 38.470768 38.470768 38.470768
92 62 23.345235 23.345235 23.345235
92 63 17.464249 17.464249 17.464249
92 64 27.202941 27.202941 27.202941
92 65 12.649111 12.649111 12.649111
92 66 14.142136 14.142136 14.142136
92 67 9.433981 9.433981 9.433981
92 68 15.000000 15.000000 15.000000
92 69 20.124612 20.124612 20.124612
92 70 20.591260 20.591260 20.591260
92 71 30.413813 30.413813 30.413813
92 72 20.615528 20.615528 20.615528
92 73 26.925824 26.925824 26.925824
92 74 50.328918 50.328918 50.328918
92 75 36.400549 36.400549 36.400549
92 76 57.489129 57.489129 57.489129
92 77 31.953091 31.953091 31.953091
92 78 46.872167 46.872167 46.872167
92 79 43.324358 43.324358 43.324358
92 80 50.249378 50.249378 50.249378
92 81 5.385165 5.385165 5.385165
92 82 16.000000 16.000000 16.000000
92 83 22.022716 22.022716 22.022716
92 84 16.278821 16.278821 16.278821
92 85 15.264338 15.264338 15.264338
92 86 23.600847 23.600847 23.600847
92 87 33.286634 33.286634 33.286634
92 88 41.146081 41.146081 41.146081
92 89 29.681644 29.681644 29.681644
92 90 41.146081 41.146081 41.146081
92 91 13.000000 13.000000 13.000000
92 93 4.123106 4.123106 4.123106
92 94 15.620499 15.620499 15.620499
92 95 10.000000 10.000000 10.000000
92 96 8.602325 8.602325 8.602325
92 97 13.416408 13.416408 13.416408
92 98 51.000000 51.000000 51.000000
92 99 25.079872 25.079872 25.079872
92 100 24.041631 24.041631 24.041631
92 101 30.805844 30.805844 30.805844
93 1 14.764823 14.764823 14.764823
93 2 50.477718 50.477718 50.477718
93 3 44.553339 44.553339 44.553339
93 4 52.201533 52.201533 52.201533
93 5 49.578221 49.578221 49.578221
93 6 53.413481 53.413481 53.413481
93 7 47.423623 47.423623 47.423623
93 8 49.678969 49.678969 49.678969
93 9 53.037722 53.037722 53.037722
93 10 43.737855 43.737855 43.737855
93 11 43.104524 43.104524 43.104524
93 12 45.099889 45.099889 45.099889
93 13 45.044423 45.044423 45.044423
93 14 48.662100 48.662100 48.662100
93 15 48.041649 48.041649 48.041649
93 16 51.088159 51.088159 51.088159
93 17 53.084838 53.084838 53.084838
93 18 53.037722 53.037722 53.037722
93 19 39.051248 39.051248 39.051248
93 20 34.785054 34.785054 34.785054
93 21 30.083218 30.083218 30.083218
93 22 40.162171 40.162171 40.162171
93 23 30.870698 30.870698 30.870698
93 24 40.853396 40.853396 40.853396
93 25 31.764760 31.764760 31.764760
93 26 42.047592 42.047592 42.047592
93 27 43.965896 43.965896 43.965896
93 28 42.755117 42.755117 42.755117
93 29 41.109610 41.109610 41.109610
93 30 37.854986 37.854986 37.854986
93 31 37.336309 37.336309 37.336309
93 32 35.902646 35.902646 35.902646
93 33 36.400549 36.400549 36.400549
93 34 36.715120 36.715120 36.715120
93 35 32.984845 32.984845 32.984845
93 36 44.271887 44.271887 44.271887
93 37 43.680659 43.680659 43.680659
93 38 40.804412 40.804412 40.804412
93 39 38.078866 38.078866 38.078866
93 40 37.656341 37.656341 37.656341
93 41 42.579338 42.579338 42.579338
93 42 32.388269 32.388269 32.388269
93 43 37.054015 37.054015 37.054015
93 44 42.047592 42.047592 42.047592
93 45 39.051248 39.051248 39.051248
93 46 51.088159 51.088159 51.088159
93 47 50.931326 50.931326 50.931326
93 48 51.039201 51.039201 51.039201
93 49 39.560081 39.560081 39.560081
93 50 32.893768 32.893768 32.893768
93 51 20.615528 20.615528 20.615528
93 52 23.086793 23.086793 23.086793

93 53 30.870698 30.870698 30.870698
93 54 33.734256 33.734256 33.734256
93 55 17.117243 17.117243 17.117243
93 56 28.600699 28.600699 28.600699
93 57 8.544004 8.544004 8.544004
93 58 29.206164 29.206164 29.206164
93 59 50.328918 50.328918 50.328918
93 60 48.764741 48.764741 48.764741
93 61 41.629317 41.629317 41.629317
93 62 23.409400 23.409400 23.409400
93 63 14.422205 14.422205 14.422205
93 64 25.942244 25.942244 25.942244
93 65 15.264338 15.264338 15.264338
93 66 18.248288 18.248288 18.248288
93 67 13.416408 13.416408 13.416408
93 68 11.045361 11.045361 11.045361
93 69 21.400935 21.400935 21.400935
93 70 23.769729 23.769729 23.769729
93 71 31.622777 31.622777 31.622777
93 72 16.970563 16.970563 16.970563
93 73 24.166092 24.166092 24.166092
93 74 53.758720 53.758720 53.758720
93 75 40.224371 40.224371 40.224371
93 76 61.220911 61.220911 61.220911
93 77 31.780497 31.780497 31.780497
93 78 50.000000 50.000000 50.000000
93 79 46.840154 46.840154 46.840154
93 80 53.235327 53.235327 53.235327
93 81 7.211103 7.211103 7.211103
93 82 15.524175 15.524175 15.524175
93 83 26.000000 26.000000 26.000000
93 84 20.000000 20.000000 20.000000
93 85 14.560220 14.560220 14.560220
93 86 22.360680 22.360680 22.360680
93 87 37.215588 37.215588 37.215588
93 88 45.188494 45.188494 45.188494
93 89 32.649655 32.649655 32.649655
93 90 40.496913 40.496913 40.496913
93 91 16.492423 16.492423 16.492423
93 92 4.123106 4.123106 4.123106
93 94 12.041595 12.041595 12.041595
93 95 6.403124 6.403124 6.403124
93 96 6.708204 6.708204 6.708204
93 97 11.180340 11.180340 11.180340
93 98 55.009090 55.009090 55.009090
93 99 28.460499 28.460499 28.460499
93 100 28.160256 28.160256 28.160256
93 101 32.557641 32.557641 32.557641
94 1 21.095023 21.095023 21.095023
94 2 48.836462 48.836462 48.836462
94 3 45.276926 45.276926 45.276926
94 4 51.088159 51.088159 51.088159
94 5 49.648766 49.648766 49.648766
94 6 52.630789 52.630789 52.630789
94 7 48.764741 48.764741 48.764741
94 8 51.429563 51.429563 51.429563
94 9 53.851648 53.851648 53.851648
94 10 53.758720 53.758720 53.758720
94 11 52.392748 52.392748 52.392748
94 12 54.341513 54.341513 54.341513
94 13 53.460266 53.460266 53.460266
94 14 58.523500 58.523500 58.523500
94 15 56.435804 56.435804 56.435804
94 16 60.207973 60.207973 60.207973
94 17 62.169124 62.169124 62.169124
94 18 61.400326 61.400326 61.400326
94 19 49.979996 49.979996 49.979996
94 20 46.097722 46.097722 46.097722
94 21 41.593269 41.593269 41.593269
94 22 51.478151 51.478151 51.478151
94 23 42.544095 42.544095 42.544095
94 24 52.325902 52.325902 52.325902
94 25 43.566042 43.566042 43.566042
94 26 53.712196 53.712196 53.712196
94 27 40.496913 40.496913 40.496913
94 28 38.013156 38.013156 38.013156
94 29 38.013156 38.013156 38.013156
94 30 33.615473 33.615473 33.615473
94 31 34.828150 34.828150 34.828150
94 32 31.906112 31.906112 31.906112
94 33 34.058773 34.058773 34.058773
94 34 36.124784 36.124784 36.124784
94 35 29.410882 29.410882 29.410882
94 36 33.541020 33.541020 33.541020
94 37 33.241540 33.241540 33.241540
94 38 30.265492 30.265492 30.265492
94 39 28.017851 28.017851 28.017851
94 40 28.017851 28.017851 28.017851
94 41 33.015148 33.015148 33.015148
94 42 23.194827 23.194827 23.194827
94 43 28.635642 28.635642 28.635642
94 44 33.541020 33.541020 33.541020
94 45 30.594117 30.594117 30.594117
94 46 50.803543 50.803543 50.803543
94 47 51.312766 51.312766 51.312766

94 48 59.413803 59.413803 59.413803
94 49 50.695167 50.695167 50.695167
94 50 44.283180 44.283180 44.283180
94 51 20.248457 20.248457 20.248457
94 52 32.557641 32.557641 32.557641
94 53 42.190046 42.190046 42.190046
94 54 41.048752 41.048752 41.048752
94 55 10.000000 10.000000 10.000000
94 56 32.015621 32.015621 32.015621
94 57 20.248457 20.248457 20.248457
94 58 41.109610 41.109610 41.109610
94 59 62.289646 62.289646 62.289646
94 60 60.207973 60.207973 60.207973
94 61 46.690470 46.690470 46.690470
94 62 20.615528 20.615528 20.615528
94 63 17.464249 17.464249 17.464249
94 64 32.249031 32.249031 32.249031
94 65 27.202941 27.202941 27.202941
94 66 28.635642 28.635642 28.635642
94 67 25.000000 25.000000 25.000000
94 68 10.440307 10.440307 10.440307
94 69 22.472205 22.472205 22.472205
94 70 30.000000 30.000000 30.000000
94 71 31.064449 31.064449 31.064449
94 72 5.000000 5.000000 5.000000
94 73 13.152946 13.152946 13.152946
94 74 59.539903 59.539903 59.539903
94 75 52.009614 52.009614 52.009614
94 76 73.109507 73.109507 73.109507
94 77 40.012498 40.012498 40.012498
94 78 62.008064 62.008064 62.008064
94 79 53.150729 53.150729 53.150729
94 80 57.280014 57.280014 57.280014
94 81 14.866069 14.866069 14.866069
94 82 13.416408 13.416408 13.416408
94 83 35.171011 35.171011 35.171011
94 84 31.890437 31.890437 31.890437
94 85 23.345235 23.345235 23.345235
94 86 29.068884 29.068884 29.068884
94 87 48.826222 48.826222 48.826222
94 88 56.435804 56.435804 56.435804
94 89 37.483330 37.483330 37.483330
94 90 47.381431 47.381431 47.381431
94 91 24.515301 24.515301 24.515301
94 92 15.620499 15.620499 15.620499
94 93 12.041595 12.041595 12.041595
94 95 5.656854 5.656854 5.656854
94 96 15.811388 15.811388 15.811388
94 97 6.324555 6.324555 6.324555
94 98 66.370174 66.370174 66.370174
94 99 35.000000 35.000000 35.000000
94 100 38.910153 38.910153 38.910153
94 101 33.541020 33.541020 33.541020
95 1 17.117243 17.117243 17.117243
95 2 48.918299 48.918299 48.918299
95 3 44.204072 44.204072 44.204072
95 4 50.931326 50.931326 50.931326
95 5 48.918299 48.918299 48.918299
95 6 52.325902 52.325902 52.325902
95 7 47.434165 47.434165 47.434165
95 8 49.929951 49.929951 49.929951
95 9 52.801515 52.801515 52.801515
95 10 48.764741 48.764741 48.764741
95 11 47.675990 47.675990 47.675990
95 12 49.648766 49.648766 49.648766
95 13 49.091751 49.091751 49.091751
95 14 53.600373 53.600373 53.600373
95 15 52.086467 52.086467 52.086467
95 16 55.578773 55.578773 55.578773
95 17 57.558666 57.558666 57.558666
95 18 57.078893 57.078893 57.078893
95 19 44.922155 44.922155 44.922155
95 20 40.853396 40.853396 40.853396
95 21 36.249138 36.249138 36.249138
95 22 46.238512 46.238512 46.238512
95 23 37.121422 37.121422 37.121422
95 24 47.010637 47.010637 47.010637
95 25 38.078866 38.078866 38.078866
95 26 48.301139 48.301139 48.301139
95 27 42.047592 42.047592 42.047592
95 28 40.162171 40.162171 40.162171
95 29 39.357337 39.357337 39.357337
95 30 35.468296 35.468296 35.468296
95 31 35.846897 35.846897 35.846897
95 32 33.615473 33.615473 33.615473
95 33 34.985711 34.985711 34.985711
95 34 36.235342 36.235342 36.235342
95 35 30.870698 30.870698 30.870698
95 36 38.327536 38.327536 38.327536
95 37 37.854986 37.854986 37.854986
95 38 34.928498 34.928498 34.928498
95 39 32.388269 32.388269 32.388269
95 40 32.140317 32.140317 32.140317
95 41 37.121422 37.121422 37.121422
95 42 27.018512 27.018512 27.018512

95 43 32.062439 32.062439 32.062439
95 44 37.054015 37.054015 37.054015
95 45 34.058773 34.058773 34.058773
95 46 50.249378 50.249378 50.249378
95 47 50.447993 50.447993 50.447993
95 48 55.081757 55.081757 55.081757
95 49 45.541190 45.541190 45.541190
95 50 39.000000 39.000000 39.000000
95 51 19.849433 19.849433 19.849433
95 52 28.071338 28.071338 28.071338
95 53 36.715120 36.715120 36.715120
95 54 37.054015 37.054015 37.054015
95 55 12.165525 12.165525 12.165525
95 56 29.546573 29.546573 29.546573
95 57 14.764823 14.764823 14.764823
95 58 35.468296 35.468296 35.468296
95 59 56.639209 56.639209 56.639209
95 60 54.708317 54.708317 54.708317
95 61 43.680659 43.680659 43.680659
95 62 20.808652 20.808652 20.808652
95 63 15.264338 15.264338 15.264338
95 64 29.120440 29.120440 29.120440
95 65 21.633308 21.633308 21.633308
95 66 23.409400 23.409400 23.409400
95 67 19.416488 19.416488 19.416488
95 68 9.219544 9.219544 9.219544
95 69 20.808652 20.808652 20.808652
95 70 26.305893 26.305893 26.305893
95 71 30.413813 30.413813 30.413813
95 72 10.630146 10.630146 10.630146
95 73 18.027756 18.027756 18.027756
95 74 56.293872 56.293872 56.293872
95 75 46.400431 46.400431 46.400431
95 76 67.475922 67.475922 67.475922
95 77 36.124784 36.124784 36.124784
95 78 56.400355 56.400355 56.400355
95 79 49.648766 49.648766 49.648766
95 80 54.781384 54.781384 54.781384
95 81 10.049876 10.049876 10.049876
95 82 12.806248 12.806248 12.806248
95 83 30.413813 30.413813 30.413813
95 84 26.248809 26.248809 26.248809
95 85 19.000000 19.000000 19.000000
95 86 25.709920 25.709920 25.709920
95 87 43.266615 43.266615 43.266615
95 88 51.000000 51.000000 51.000000
95 89 34.481879 34.481879 34.481879
95 90 44.147480 44.147480 44.147480
95 91 20.024984 20.024984 20.024984
95 92 10.000000 10.000000 10.000000
95 93 6.403124 6.403124 6.403124
95 94 5.656854 5.656854 5.656854
95 96 11.045361 11.045361 11.045361
95 97 6.324555 6.324555 6.324555
95 98 60.901560 60.901560 60.901560
95 99 31.256999 31.256999 31.256999
95 100 33.615473 33.615473 33.615473
95 101 32.202484 32.202484 32.202484
96 1 20.615528 20.615528 20.615528
96 2 57.140179 57.140179 57.140179
96 3 50.990195 50.990195 50.990195
96 4 58.821765 58.821765 58.821765
96 5 56.080300 56.080300 56.080300
96 6 60.000000 60.000000 60.000000
96 7 53.740115 53.740115 53.740115
96 8 55.901699 55.901699 55.901699
96 9 59.413803 59.413803 59.413803
96 10 46.043458 46.043458 46.043458
96 11 46.097722 46.097722 46.097722
96 12 48.093659 48.093659 48.093659
96 13 48.662100 48.662100 48.662100
96 14 51.039201 51.039201 51.039201
96 15 51.623638 51.623638 51.623638
96 16 54.083269 54.083269 54.083269
96 17 56.080300 56.080300 56.080300
96 18 56.568542 56.568542 56.568542
96 19 34.176015 34.176015 34.176015
96 20 30.413813 30.413813 30.413813
96 21 26.076810 26.076810 26.076810
96 22 35.777088 35.777088 35.777088
96 23 27.202941 27.202941 27.202941
96 24 36.715120 36.715120 36.715120
96 25 28.425341 28.425341 28.425341
96 26 38.275318 38.275318 38.275318
96 27 39.623226 39.623226 39.623226
96 28 39.051248 39.051248 39.051248
96 29 36.674242 36.674242 36.674242
96 30 34.058773 34.058773 34.058773
96 31 32.756679 32.756679 32.756679
96 32 32.062439 32.062439 32.062439
96 33 31.780497 31.780497 31.780497
96 34 31.384710 31.384710 31.384710
96 35 29.068884 29.068884 29.068884
96 36 49.244289 49.244289 49.244289
96 37 48.836462 48.836462 48.836462

96 38 45.891176 45.891176 45.891176
96 39 43.416587 43.416587 43.416587
96 40 43.185646 43.185646 43.185646
96 41 48.166378 48.166378 48.166378
96 42 38.052595 38.052595 38.052595
96 43 43.011626 43.011626 43.011626
96 44 48.010416 48.010416 48.010416
96 45 45.011110 45.011110 45.011110
96 46 57.628118 57.628118 57.628118
96 47 57.384667 57.384667 57.384667
96 48 54.589376 54.589376 54.589376
96 49 34.928498 34.928498 34.928498
96 50 28.653098 28.653098 28.653098
96 51 16.124515 16.124515 16.124515
96 52 17.029386 17.029386 17.029386
96 53 31.780497 31.780497 31.780497
96 54 38.275318 38.275318 38.275318
96 55 23.021729 23.021729 23.021729
96 56 34.713110 34.713110 34.713110
96 57 6.324555 6.324555 6.324555
96 58 28.635642 28.635642 28.635642
96 59 49.091751 49.091751 49.091751
96 60 49.040799 49.040799 49.040799
96 61 47.010637 47.010637 47.010637
96 62 30.083218 30.083218 30.083218
96 63 9.219544 9.219544 9.219544
96 64 19.235384 19.235384 19.235384
96 65 13.038405 13.038405 13.038405
96 66 21.213203 21.213203 21.213203
96 67 15.000000 15.000000 15.000000
96 68 9.433981 9.433981 9.433981
96 69 28.017851 28.017851 28.017851
96 70 29.154759 29.154759 29.154759
96 71 38.275318 38.275318 38.275318
96 72 20.124612 20.124612 20.124612
96 73 28.861739 28.861739 28.861739
96 74 58.694122 58.694122 58.694122
96 75 39.812058 39.812058 39.812058
96 76 60.207973 60.207973 60.207973
96 77 25.317978 25.317978 25.317978
96 78 47.381431 47.381431 47.381431
96 79 51.623638 51.623638 51.623638
96 80 58.830264 58.830264 58.830264
96 81 13.453624 13.453624 13.453624
96 82 22.135944 22.135944 22.135944
96 83 29.614186 29.614186 29.614186
96 84 19.924859 19.924859 19.924859
96 85 8.062258 8.062258 8.062258
96 86 15.652476 15.652476 15.652476
96 87 37.336309 37.336309 37.336309
96 88 45.880279 45.880279 45.880279
96 89 38.275318 38.275318 38.275318
96 90 33.837849 33.837849 33.837849
96 91 21.470911 21.470911 21.470911
96 92 8.602325 8.602325 8.602325
96 93 6.708204 6.708204 6.708204
96 94 15.811388 15.811388 15.811388
96 95 11.045361 11.045361 11.045361
96 97 17.029386 17.029386 17.029386
96 98 55.362442 55.362442 55.362442
96 99 33.541020 33.541020 33.541020
96 100 30.066593 30.066593 30.066593
96 101 39.051248 39.051248 39.051248
97 1 15.524175 15.524175 15.524175
97 2 43.139309 43.139309 43.139309
97 3 39.115214 39.115214 39.115214
97 4 45.276926 45.276926 45.276926
97 5 43.600459 43.600459 43.600459
97 6 46.754679 46.754679 46.754679
97 7 42.544095 42.544095 42.544095
97 8 45.177428 45.177428 45.177428
97 9 47.707442 47.707442 47.707442
97 10 48.846699 48.846699 48.846699
97 11 47.127487 47.127487 47.127487
97 12 49.040799 49.040799 49.040799
97 13 47.853944 47.853944 47.853944
97 14 53.488316 53.488316 53.488316
97 15 50.803543 50.803543 50.803543
97 16 54.817880 54.817880 54.817880
97 17 56.753854 56.753854 56.753854
97 18 55.731499 55.731499 55.731499
97 19 50.219518 50.219518 50.219518
97 20 45.880279 45.880279 45.880279
97 21 41.109610 41.109610 41.109610
97 22 51.244512 51.244512 51.244512
97 23 41.785165 41.785165 41.785165
97 24 51.865210 51.865210 51.865210
97 25 42.544095 42.544095 42.544095
97 26 52.924474 52.924474 52.924474
97 27 46.647615 46.647615 46.647615
97 28 44.283180 44.283180 44.283180
97 29 44.102154 44.102154 44.102154
97 30 39.824616 39.824616 39.824616
97 31 40.804412 40.804412 40.804412
97 32 38.078866 38.078866 38.078866

97 33 40.000000 40.000000 40.000000
97 34 41.725292 41.725292 41.725292
97 35 35.510562 35.510562 35.510562
97 36 33.241540 33.241540 33.241540
97 37 32.572995 32.572995 32.572995
97 38 29.732137 29.732137 29.732137
97 39 26.925824 26.925824 26.925824
97 40 26.476405 26.476405 26.476405
97 41 31.400637 31.400637 31.400637
97 42 21.213203 21.213203 21.213203
97 43 26.000000 26.000000 26.000000
97 44 31.000000 31.000000 31.000000
97 45 28.000000 28.000000 28.000000
97 46 44.821870 44.821870 44.821870
97 47 45.221676 45.221676 45.221676
97 48 53.758720 53.758720 53.758720
97 49 50.695167 50.695167 50.695167
97 50 43.965896 43.965896 43.965896
97 51 25.495098 25.495098 25.495098
97 52 34.000000 34.000000 34.000000
97 53 38.418745 38.418745 38.418745
97 54 35.227830 35.227830 35.227830
97 55 6.000000 6.000000 6.000000
97 56 25.709920 25.709920 25.709920
97 57 19.646883 19.646883 19.646883
97 58 38.288379 38.288379 38.288379
97 59 59.464275 59.464275 59.464275
97 60 56.400355 56.400355 56.400355
97 61 40.447497 40.447497 40.447497
97 62 14.866069 14.866069 14.866069
97 63 21.470911 21.470911 21.470911
97 64 35.440090 35.440090 35.440090
97 65 26.000000 26.000000 26.000000
97 66 24.413111 24.413111 24.413111
97 67 22.022716 22.022716 22.022716
97 68 15.000000 15.000000 15.000000
97 69 16.155494 16.155494 16.155494
97 70 24.083189 24.083189 24.083189
97 71 25.000000 25.000000 25.000000
97 72 10.049876 10.049876 10.049876
97 73 13.601471 13.601471 13.601471
97 74 53.338541 53.338541 53.338541
97 75 48.795492 48.795492 48.795492
97 76 70.007142 70.007142 70.007142
97 77 42.296572 42.296572 42.296572
97 78 60.207973 60.207973 60.207973
97 79 47.042534 47.042534 47.042534
97 80 50.960769 50.960769 50.960769
97 81 10.630146 10.630146 10.630146
97 82 7.211103 7.211103 7.211103
97 83 30.083218 30.083218 30.083218
97 84 29.206164 29.206164 29.206164
97 85 25.079872 25.079872 25.079872
97 86 32.015621 32.015621 32.015621
97 87 45.343136 45.343136 45.343136
97 88 52.430907 52.430907 52.430907
97 89 31.256999 31.256999 31.256999
97 90 50.447993 50.447993 50.447993
97 91 19.313208 19.313208 19.313208
97 92 13.416408 13.416408 13.416408
97 93 11.180340 11.180340 11.180340
97 94 6.324555 6.324555 6.324555
97 95 6.324555 6.324555 6.324555
97 96 17.029386 17.029386 17.029386
97 98 62.425956 62.425956 62.425956
97 99 29.068884 29.068884 29.068884
97 100 34.669872 34.669872 34.669872
97 101 27.294688 27.294688 27.294688
98 1 48.166378 48.166378 48.166378
98 2 70.213959 70.213959 70.213959
98 3 59.774577 59.774577 59.774577
98 4 69.375788 69.375788 69.375788
98 5 64.031242 64.031242 64.031242
98 6 68.883960 68.883960 68.883960
98 7 58.694122 58.694122 58.694122
98 8 58.051701 58.051701 58.051701
98 9 62.968246 62.968246 62.968246
98 10 18.027756 18.027756 18.027756
98 11 22.803509 22.803509 22.803509
98 12 22.360680 22.360680 22.360680
98 13 27.294688 27.294688 27.294688
98 14 17.029386 17.029386 17.029386
98 15 27.018512 27.018512 27.018512
98 16 22.090722 22.090722 22.090722
98 17 22.360680 22.360680 22.360680
98 18 27.294688 27.294688 27.294688
98 19 42.059482 42.059482 42.059482
98 20 38.832976 38.832976 38.832976
98 21 38.118237 38.118237 38.118237
98 22 38.275318 38.275318 38.275318
98 23 36.124784 36.124784 36.124784
98 24 36.400549 36.400549 36.400549
98 25 34.132096 34.132096 34.132096
98 26 33.615473 33.615473 33.615473
98 27 91.787799 91.787799 91.787799

98 28 92.574294 92.574294 92.574294
98 29 88.814413 88.814413 88.814413
98 30 87.664132 87.664132 87.664132
98 31 84.852814 84.852814 84.852814
98 32 85.702975 85.702975 85.702975
98 33 83.862983 83.862983 83.862983
98 34 81.301906 81.301906 81.301906
98 35 82.764727 82.764727 82.764727
98 36 91.967386 91.967386 91.967386
98 37 90.609050 90.609050 90.609050
98 38 88.413800 88.413800 88.413800
98 39 84.899941 84.899941 84.899941
98 40 83.546394 83.546394 83.546394
98 41 87.321246 87.321246 87.321246
98 42 78.517514 78.517514 78.517514
98 43 80.280757 80.280757 80.280757
98 44 84.202138 84.202138 84.202138
98 45 81.835200 81.835200 81.835200
98 46 65.969690 65.969690 65.969690
98 47 63.560994 63.560994 63.560994
98 48 27.073973 27.073973 27.073973
98 49 40.162171 40.162171 40.162171
98 50 38.470768 38.470768 38.470768
98 51 70.092796 70.092796 70.092796
98 52 51.039201 51.039201 51.039201
98 53 24.186773 24.186773 24.186773
98 54 35.777088 35.777088 35.777088
98 55 66.068147 66.068147 66.068147
98 56 49.396356 49.396356 49.396356
98 57 49.040799 49.040799 49.040799
98 58 26.925824 26.925824 26.925824
98 59 13.601471 13.601471 13.601471
98 60 6.324555 6.324555 6.324555
98 61 43.416587 43.416587 43.416587
98 62 62.369865 62.369865 62.369865
98 63 63.324561 63.324561 63.324561
98 64 61.032778 61.032778 61.032778
98 65 42.720019 42.720019 42.720019
98 66 38.013156 38.013156 38.013156
98 67 41.593269 41.593269 41.593269
98 68 64.621978 64.621978 64.621978
98 69 55.317267 55.317267 55.317267
98 70 43.416587 43.416587 43.416587
98 71 59.682493 59.682493 59.682493
98 72 71.344236 71.344236 71.344236
98 73 75.432089 75.432089 75.432089
98 74 42.047592 42.047592 42.047592
98 75 16.124515 16.124515 16.124515
98 76 13.038405 13.038405 13.038405
98 77 56.320511 56.320511 56.320511
98 78 24.207437 24.207437 24.207437
98 79 38.209946 38.209946 38.209946
98 80 50.039984 50.039984 50.039984
98 81 51.865210 51.865210 51.865210
98 82 60.207973 60.207973 60.207973
98 83 33.970576 33.970576 33.970576
98 84 35.468296 35.468296 35.468296
98 85 54.129474 54.129474 54.129474
98 86 59.211485 59.211485 59.211485
98 87 18.027756 18.027756 18.027756
98 88 10.000000 10.000000 10.000000
98 89 44.721360 44.721360 44.721360
98 90 64.327288 64.327288 64.327288
98 91 43.931765 43.931765 43.931765
98 92 51.000000 51.000000 51.000000
98 93 55.009090 55.009090 55.009090
98 94 66.370174 66.370174 66.370174
98 95 60.901560 60.901560 60.901560
98 96 55.362442 55.362442 55.362442
98 97 62.425956 62.425956 62.425956
98 99 40.496913 40.496913 40.496913
98 100 27.802878 27.802878 27.802878
98 101 55.946403 55.946403 55.946403
99 1 14.142136 14.142136 14.142136
99 2 33.015148 33.015148 33.015148
99 3 23.345235 23.345235 23.345235
99 4 33.241540 33.241540 33.241540
99 5 28.635642 28.635642 28.635642
99 6 33.541020 33.541020 33.541020
99 7 24.351591 24.351591 24.351591
99 8 25.495098 25.495098 25.495098
99 9 30.083218 30.083218 30.083218
99 10 23.345235 23.345235 23.345235
99 11 20.000000 20.000000 20.000000
99 12 21.633308 21.633308 21.633308
99 13 19.313208 19.313208 19.313208
99 14 27.018512 27.018512 27.018512
99 15 22.135944 22.135944 22.135944
99 16 26.832816 26.832816 26.832816
99 17 28.635642 28.635642 28.635642
99 18 26.925824 26.925824 26.925824
99 19 50.328918 50.328918 50.328918
99 20 44.944410 44.944410 44.944410
99 21 40.311289 40.311289 40.311289
99 22 49.040799 49.040799 49.040799

99 23 39.560081 39.560081 39.560081
99 24 48.507731 48.507731 48.507731
99 25 38.897301 38.897301 38.897301
99 26 47.853944 47.853944 47.853944
99 27 72.422372 72.422372 72.422372
99 28 71.063352 71.063352 71.063352
99 29 69.570109 69.570109 69.570109
99 30 66.219333 66.219333 66.219333
99 31 65.787537 65.787537 65.787537
99 32 64.288413 64.288413 64.288413
99 33 64.845971 64.845971 64.845971
99 34 64.884513 64.884513 64.884513
99 35 61.400326 61.400326 61.400326
99 36 52.630789 52.630789 52.630789
99 37 51.088159 51.088159 51.088159
99 38 49.203658 49.203658 49.203658
99 39 45.607017 45.607017 45.607017
99 40 44.045431 44.045431 44.045431
99 41 47.381431 47.381431 47.381431
99 42 39.408121 39.408121 39.408121
99 43 40.311289 40.311289 40.311289
99 44 43.931765 43.931765 43.931765
99 45 41.725292 41.725292 41.725292
99 46 30.594117 30.594117 30.594117
99 47 29.120440 29.120440 29.120440
99 48 25.000000 25.000000 25.000000
99 49 49.648766 49.648766 49.648766
99 50 43.081318 43.081318 43.081318
99 51 49.040799 49.040799 49.040799
99 52 43.185646 43.185646 43.185646
99 53 22.022716 22.022716 22.022716
99 54 6.324555 6.324555 6.324555
99 55 30.083218 30.083218 30.083218
99 56 8.944272 8.944272 8.944272
99 57 29.410882 29.410882 29.410882
99 58 27.294688 27.294688 27.294688
99 59 43.416587 43.416587 43.416587
99 60 35.777088 35.777088 35.777088
99 61 13.601471 13.601471 13.601471
99 62 23.021729 23.021729 23.021729
99 63 42.544095 42.544095 42.544095
99 64 50.447993 50.447993 50.447993
99 65 29.068884 29.068884 29.068884
99 66 15.000000 15.000000 15.000000
99 67 21.213203 21.213203 21.213203
99 68 39.293765 39.293765 39.293765
99 69 16.124515 16.124515 16.124515
99 70 5.000000 5.000000 5.000000
99 71 19.235384 19.235384 19.235384
99 72 39.115214 39.115214 39.115214
99 73 39.217343 39.217343 39.217343
99 74 25.298221 25.298221 25.298221
99 75 32.557641 32.557641 32.557641
99 76 51.478151 51.478151 51.478151
99 77 52.497619 52.497619 52.497619
99 78 49.091751 49.091751 49.091751
99 79 18.439089 18.439089 18.439089
99 80 25.612497 25.612497 25.612497
99 81 21.587033 21.587033 21.587033
99 82 23.769729 23.769729 23.769729
99 83 9.055385 9.055385 9.055385
99 84 23.706539 23.706539 23.706539
99 85 38.600518 38.600518 38.600518
99 86 47.010637 47.010637 47.010637
99 87 28.442925 28.442925 28.442925
99 88 31.304952 31.304952 31.304952
99 89 6.324555 6.324555 6.324555
99 90 62.369865 62.369865 62.369865
99 91 12.083046 12.083046 12.083046
99 92 25.079872 25.079872 25.079872
99 93 28.460499 28.460499 28.460499
99 94 35.000000 35.000000 35.000000
99 95 31.256999 31.256999 31.256999
99 96 33.541020 33.541020 33.541020
99 97 29.068884 29.068884 29.068884
99 98 40.496913 40.496913 40.496913
99 100 17.000000 17.000000 17.000000
99 101 15.811388 15.811388 15.811388
100 1 20.518285 20.518285 20.518285
100 2 50.009999 50.009999 50.009999
100 3 40.199502 40.199502 40.199502
100 4 50.159745 50.159745 50.159745
100 5 45.398238 45.398238 45.398238
100 6 50.358713 50.358713 50.358713
100 7 40.792156 40.792156 40.792156
100 8 41.484937 41.484937 41.484937
100 9 46.324939 46.324939 46.324939
100 10 16.000000 16.000000 16.000000
100 11 16.763055 16.763055 16.763055
100 12 18.681542 18.681542 18.681542
100 13 20.591260 20.591260 20.591260
100 14 21.000000 21.000000 21.000000
100 15 23.259407 23.259407 23.259407
100 16 24.515301 24.515301 24.515301
100 17 26.476405 26.476405 26.476405

100 18 27.856777 27.856777 27.856777
100 19 34.985711 34.985711 34.985711
100 20 29.681644 29.681644 29.681644
100 21 25.612497 25.612497 25.612497
100 22 33.105891 33.105891 33.105891
100 23 24.413111 24.413111 24.413111
100 24 32.310989 32.310989 32.310989
100 25 23.323808 23.323808 23.323808
100 26 31.320920 31.320920 31.320920
100 27 69.180922 69.180922 69.180922
100 28 69.000000 69.000000 69.000000
100 29 66.189123 66.189123 66.189123
100 30 64.000000 64.000000 64.000000
100 31 62.201286 62.201286 62.201286
100 32 62.000000 62.000000 62.000000
100 33 61.204575 61.204575 61.204575
100 34 59.841457 59.841457 59.841457
100 35 59.000000 59.000000 59.000000
100 36 64.660653 64.660653 64.660653
100 37 63.411355 63.411355 63.411355
100 38 61.073726 61.073726 61.073726
100 39 57.628118 57.628118 57.628118
100 40 56.400355 56.400355 56.400355
100 41 60.464866 60.464866 60.464866
100 42 51.224994 51.224994 51.224994
100 43 53.535035 53.535035 53.535035
100 44 57.801384 57.801384 57.801384
100 45 55.226805 55.226805 55.226805
100 46 47.381431 47.381431 47.381431
100 47 45.705580 45.705580 45.705580
100 48 26.000000 26.000000 26.000000
100 49 34.000000 34.000000 34.000000
100 50 28.017851 28.017851 28.017851
100 51 46.000000 46.000000 46.000000
100 52 32.649655 32.649655 32.649655
100 53 5.099020 5.099020 5.099020
100 54 16.155494 16.155494 16.155494
100 55 38.288379 38.288379 38.288379
100 56 25.317978 25.317978 25.317978
100 57 24.000000 24.000000 24.000000
100 58 10.770330 10.770330 10.770330
100 59 27.313001 27.313001 27.313001
100 60 21.931712 21.931712 21.931712
100 61 27.313001 27.313001 27.313001
100 62 35.510562 35.510562 35.510562
100 63 39.000000 39.000000 39.000000
100 64 41.785165 41.785165 41.785165
100 65 19.646883 19.646883 19.646883
100 66 10.295630 10.295630 10.295630
100 67 15.132746 15.132746 15.132746
100 68 38.639358 38.639358 38.639358
100 69 28.653098 28.653098 28.653098
100 70 17.720045 17.720045 17.720045
100 71 35.171011 35.171011 35.171011
100 72 43.829214 43.829214 43.829214
100 73 47.634021 47.634021 47.634021
100 74 34.655447 34.655447 34.655447
100 75 16.155494 16.155494 16.155494
100 76 36.619667 36.619667 36.619667
100 77 41.048752 41.048752 41.048752
100 78 32.140317 32.140317 32.140317
100 79 27.658633 27.658633 27.658633
100 80 38.587563 38.587563 38.587563
100 81 24.186773 24.186773 24.186773
100 82 32.526912 32.526912 32.526912
100 83 8.062258 8.062258 8.062258
100 84 11.704700 11.704700 11.704700
100 85 31.575307 31.575307 31.575307
100 86 38.897301 38.897301 38.897301
100 87 12.083046 12.083046 12.083046
100 88 17.804494 17.804494 17.804494
100 89 23.086793 23.086793 23.086793
100 90 50.803543 50.803543 50.803543
100 91 16.278821 16.278821 16.278821
100 92 24.041631 24.041631 24.041631
100 93 28.160256 28.160256 28.160256
100 94 38.910153 38.910153 38.910153
100 95 33.615473 33.615473 33.615473
100 96 30.066593 30.066593 30.066593
100 97 34.669872 34.669872 34.669872
100 98 27.802878 27.802878 27.802878
100 99 17.000000 17.000000 17.000000
100 101 32.388269 32.388269 32.388269
101 1 19.235384 19.235384 19.235384
101 2 18.973666 18.973666 18.973666
101 3 12.041595 12.041595 12.041595
101 4 20.124612 20.124612 20.124612
101 5 17.029386 17.029386 17.029386
101 6 21.095023 21.095023 21.095023
101 7 15.264338 15.264338 15.264338
101 8 17.888544 17.888544 17.888544
101 9 20.615528 20.615528 20.615528
101 10 38.275318 38.275318 38.275318
101 11 34.205263 34.205263 34.205263
101 12 35.468296 35.468296 35.468296

```
101 13 31.827661 31.827661 31.827661
101 14 41.231056 41.231056 41.231056
101 15 34.058773 34.058773 34.058773
101 16 39.623226 39.623226 39.623226
101 17 41.109610 41.109610 41.109610
101 18 38.013156 38.013156 38.013156
101 19 63.348244 63.348244 63.348244
101 20 58.051701 58.051701 58.051701
101 21 53.150729 53.150729 53.150729
101 22 62.649820 62.649820 62.649820
101 23 52.773099 52.773099 52.773099
101 24 62.393910 62.393910 62.393910
101 25 52.469038 52.469038 52.469038
101 26 62.128898 62.128898 62.128898
101 27 73.925638 73.925638 73.925638
101 28 71.554175 71.554175 71.554175
101 29 71.344236 71.344236 71.344236
101 30 67.119297 67.119297 67.119297
101 31 67.955868 67.955868 67.955868
101 32 65.368188 65.368188 65.368188
101 33 67.119297 67.119297 67.119297
101 34 68.410526 68.410526 68.410526
101 35 62.769419 62.769419 62.769419
101 36 40.249224 40.249224 40.249224
101 37 38.470768 38.470768 38.470768
101 38 37.161808 37.161808 37.161808
101 39 33.615473 33.615473 33.615473
101 40 31.780497 31.780497 31.780497
101 41 34.132096 34.132096 34.132096
101 42 28.160256 28.160256 28.160256
101 43 27.294688 27.294688 27.294688
101 44 30.000000 30.000000 30.000000
101 45 28.301943 28.301943 28.301943
101 46 18.601075 18.601075 18.601075
101 47 18.384776 18.384776 18.384776
101 48 36.400549 36.400549 36.400549
101 49 62.968246 62.968246 62.968246
101 50 56.089215 56.089215 56.089215
101 51 52.009614 52.009614 52.009614
101 52 52.773099 52.773099 52.773099
101 53 37.483330 37.483330 37.483330
101 54 20.248457 20.248457 20.248457
101 55 25.000000 25.000000 25.000000
101 56 7.071068 7.071068 7.071068
101 57 37.215588 37.215588 37.215588
101 58 42.011903 42.011903 42.011903
101 59 59.203040 59.203040 59.203040
101 60 51.478151 51.478151 51.478151
101 61 17.464249 17.464249 17.464249
101 62 14.142136 14.142136 14.142136
101 63 46.690470 46.690470 46.690470
101 64 58.008620 58.008620 58.008620
101 65 39.560081 39.560081 39.560081
101 66 27.294688 27.294688 27.294688
101 67 31.622777 31.622777 31.622777
101 68 41.400483 41.400483 41.400483
101 69 11.401754 11.401754 11.401754
101 70 15.000000 15.000000 15.000000
101 71 4.472136 4.472136 4.472136
101 72 36.055513 36.055513 36.055513
101 73 32.062439 32.062439 32.062439
101 74 29.832868 29.832868 29.832868
101 75 48.270074 48.270074 48.270074
101 76 67.230945 67.230945 67.230945
101 77 62.177166 62.177166 62.177166
101 78 64.498062 64.498062 64.498062
101 79 25.495098 25.495098 25.495098
101 80 25.019992 25.019992 25.019992
101 81 25.612497 25.612497 25.612497
101 82 20.124612 20.124612 20.124612
101 83 24.331050 24.331050 24.331050
101 84 36.496575 36.496575 36.496575
101 85 46.043458 46.043458 46.043458
101 86 54.405882 54.405882 54.405882
101 87 44.147480 44.147480 44.147480
101 88 47.010637 47.010637 47.010637
101 89 11.401754 11.401754 11.401754
101 90 71.693793 71.693793 71.693793
101 91 20.880613 20.880613 20.880613
101 92 30.805844 30.805844 30.805844
101 93 32.557641 32.557641 32.557641
101 94 33.541020 33.541020 33.541020
101 95 32.202484 32.202484 32.202484
101 96 39.051248 39.051248 39.051248
101 97 27.294688 27.294688 27.294688
101 98 55.946403 55.946403 55.946403
101 99 15.811388 15.811388 15.811388
101 100 32.388269 32.388269 32.388269
\.
```

See Also

- **https://en.wikipedia.org/wiki/Vehicle_routing_problem**

Indices and tables

- **Index**
- **Search Page**

Previous versions of this page

- **Supported versions:** current(**3.0**)

Introduction

Vehicle Routing Problems *VRP* are **NP-hard** optimization problem, it generalises the travelling salesman problem (TSP).

- The objective of the VRP is to minimize the total route cost.
- There are several variants of the VRP problem,

pgRouting does not try to implement all variants.

Characteristics

- Capacitated Vehicle Routing Problem *CVRP* where The vehicles have limited carrying capacity of the goods.
- Vehicle Routing Problem with Time Windows *VRPTW* where the locations have time windows within which the vehicle's visits must be made.
- Vehicle Routing Problem with Pickup and Delivery *VRPPD* where a number of goods need to be moved from certain pickup locations to other delivery locations.

Limitations

- No multiple time windows for a location.
- Less vehicle used is considered better.
- Less total duration is better.
- Less wait time is better.

Pick & Delivery

Problem: *CVRPPDTW* Capacitated Pick and Delivery Vehicle Routing problem with Time Windows

- Times are relative to *0*
- The vehicles
  - have start and ending service duration times.
  - have opening and closing times for the start and ending locations.
  - have a capacity.
- The orders
  - Have pick up and delivery locations.
  - Have opening and closing times for the pickup and delivery locations.
  - Have pickup and delivery duration service times.
  - have a demand request for moving goods from the pickup location to the delivery location.
- Time based calculations:
  - Travel time between customers is distance / speed
  - Pickup and delivery order pair is done by the same vehicle.
  - A pickup is done before the delivery.

Parameters

Pick & deliver

Both implementations use the following same parameters:

Column	Type	Default	Description
orders_sql	TEXT		**Pick & Deliver Orders SQL** query containing the orders to be processed.
vehicles_sql	TEXT		**Pick & Deliver Vehicles SQL** query containing the vehicles to be used.
factor	NUMERIC	1	(Optional) Travel time multiplier. See **Factor Handling**
max_cycles	INTEGER	10	(Optional) Maximum number of cycles to perform on the optimization.

Column	Type	Default	Description
initial_sol	INTEGER	4	(Optional) Initial solution to be used. • 1 One order per truck • 2 Push front order. • 3 Push back order. • 4 Optimize insert. • 5 Push back order that allows more orders to be inserted at the back • 6 Push front order that allows more orders to be inserted at the front

The non euclidean implementation, additionally has:

Column	Type	Description
matrix_sql	TEXT	**Pick & Deliver Matrix SQL** query containing the distance or travel times.

Inner Queries

- **Pick & Deliver Orders SQL**
- **Pick & Deliver Vehicles SQL**
- **Pick & Deliver Matrix SQL**

return columns

- **Description of return columns**
- **Description of the return columns for Euclidean version**

Pick & Deliver Orders SQL

In general, the columns for the orders SQL is the same in both implementation of pick and delivery:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the pick-delivery order pair.
demand	ANY-NUMERICAL		Number of units in the order
p_open	ANY-NUMERICAL		The time, relative to 0, when the pickup location opens.
p_close	ANY-NUMERICAL		The time, relative to 0, when the pickup location closes.
d_service	ANY-NUMERICAL	0	The duration of the loading at the pickup location.
d_open	ANY-NUMERICAL		The time, relative to 0, when the delivery location opens.
d_close	ANY-NUMERICAL		The time, relative to 0, when the delivery location closes.
d_service	ANY-NUMERICAL	0	The duration of the loading at the delivery location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Description
p_node_id	ANY-INTEGER	The node identifier of the pickup, must match a node identifier in the matrix table.
d_node_id	ANY-INTEGER	The node identifier of the delivery, must match a node identifier in the matrix table.

For the euclidean implementation, pick up and delivery (x,y) locations are needed:

Column	Type	Description
p_x	ANY-NUMERICAL	x value of the pick up location
p_y	ANY-NUMERICAL	y value of the pick up location
d_x	ANY-NUMERICAL	x value of the delivery location
d_y	ANY-NUMERICAL	y value of the delivery location

Pick & Deliver Vehicles SQL

In general, the columns for the vehicles_sql is the same in both implementation of pick and delivery:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the pick-delivery order pair.
capacity	ANY-NUMERICAL		Number of units in the order
speed	ANY-NUMERICAL	*1*	Average speed of the vehicle.
start_open	ANY-NUMERICAL		The time, relative to 0, when the starting location opens.
start_close	ANY-NUMERICAL		The time, relative to 0, when the starting location closes.

Column	Type	Default	Description
start_service	ANY-NUMERICAL	*0*	The duration of the loading at the starting location.
end_open	ANY-NUMERICAL	*start_open*	The time, relative to 0, when the ending location opens.
end_close	ANY-NUMERICAL	*start_close*	The time, relative to 0, when the ending location closes.
end_service	ANY-NUMERICAL	*start_service*	The duration of the loading at the ending location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Default	Description
start_node_id	ANY-INTEGER		The node identifier of the starting location, must match a node identifier in the matrix table.
end_node_id	ANY-INTEGER	*start_node_id*	The node identifier of the ending location, must match a node identifier in the matrix table.

For the euclidean implementation, starting and ending(x,y) locations are needed:

Column	Type	Default	Description
start_x	ANY-NUMERICAL		x value of the coordinate of the starting location.
start_y	ANY-NUMERICAL		y value of the coordinate of the starting location.
end_x	ANY-NUMERICAL	*start_x*	x value of the coordinate of the ending location.
end_y	ANY-NUMERICAL	*start_y*	y value of the coordinate of the ending location.

Pick & Deliver Matrix SQL

Warning

TODO

Results

Description of the result (TODO Disussion: Euclidean & Matrix)

```
RETURNS SET OF
   (seq, vehicle_seq, vehicle_id, stop_seq, stop_type,
      travel_time, arrival_time, wait_time, service_time,  departure_time)
   UNION
   (summary row)
```

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
vehicle_seq	INTEGER	Sequential value starting from **1** for current vehicles. The n_{th} vehicle in the solution.
vehicle_id	BIGINT	Current vehicle identifier.
stop_seq	INTEGER	Sequential value starting from **1** for the stops made by the current vehicle. Them_{th} stop of the current vehicle.
stop_type	INTEGER	Kind of stop location the vehicle is at: • `1`: Starting location • `2`: Pickup location • `3`: Delivery location • `6`: Ending location
order_id	BIGINT	Pickup-Delivery order pair identifier. • `-1`: When no order is involved on the current stop location.
cargo	FLOAT	Cargo units of the vehicle when leaving the stop.
travel_time	FLOAT	Travel time from previous `stop_seq` to current `stop_seq`. • `0` When `stop_type = 1`
arrival_time	FLOAT	Previous `departure_time` plus current `travel_time`.
wait_time	FLOAT	Time spent waiting for current *location* to open.
service_time	FLOAT	Service time at current *location*.
departure_time	FLOAT	arrival_time + wait_time + service_time. • When `stop_type = 6` has the *total_time* used for the current vehicle.

Summary Row

Warning

TODO: Review the summary

Column	Type	Description
seq	INTEGER	Continues the Sequential value
vehicle_seq	INTEGER	`-2` to indicate is a summary row
vehicle_id	BIGINT	*Total Capacity Violations* in the solution.
stop_seq	INTEGER	*Total Time Window Violations* in the solution.
stop_type	INTEGER	`-1`
order_id	BIGINT	`-1`
cargo	FLOAT	`-1`
travel_time	FLOAT	*total_travel_time* The sum of all the*travel_time*
arrival_time	FLOAT	`-1`
wait_time	FLOAT	*total_waiting_time* The sum of all the*wait_time*
service_time	FLOAT	*total_service_time* The sum of all the*service_time*
departure_time	FLOAT	*total_solution_time* = total_travel_time + total_wait_time + total_service_time.

Description of the result (TODO Disussion: Euclidean & Matrix)

```
RETURNS SET OF
   (seq, vehicle_seq, vehicle_id, stop_seq, stop_type,
      travel_time, arrival_time, wait_time, service_time,  departure_time)
   UNION
   (summary row)
```

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
vehicle_seq	INTEGER	Sequential value starting from **1** for current vehicles. The n_{th} vehicle in the solution.
vehicle_id	BIGINT	Current vehicle identifier.
stop_seq	INTEGER	Sequential value starting from **1** for the stops made by the current vehicle. Them_{th} stop of the current vehicle.
stop_type	INTEGER	Kind of stop location the vehicle is at: • `1`: Starting location • `2`: Pickup location • `3`: Delivery location • `6`: Ending location
order_id	BIGINT	Pickup-Delivery order pair identifier. • `-1`: When no order is involved on the current stop location.
cargo	FLOAT	Cargo units of the vehicle when leaving the stop.
travel_time	FLOAT	Travel time from previous `stop_seq` to current `stop_seq`. • `0` When `stop_type = 1`
arrival_time	FLOAT	Previous `departure_time` plus current `travel_time`.
wait_time	FLOAT	Time spent waiting for current *location* to open.
service_time	FLOAT	Service time at current *location*.
departure_time	FLOAT	arrival_time + wait_time + service_time. • When `stop_type = 6` has the *total_time* used for the current vehicle.

Summary Row

Warning

TODO: Review the summary

Column	Type	Description
seq	INTEGER	Continues the Sequential value
vehicle_seq	INTEGER	`-2` to indicate is a summary row
vehicle_id	BIGINT	*Total Capacity Violations* in the solution.
stop_seq	INTEGER	*Total Time Window Violations* in the solution.
stop_type	INTEGER	`-1`

Column	Type	Description
order_id	BIGINT	-1
cargo	FLOAT	-1
travel_time	FLOAT	*total_travel_time* The sum of all the *travel_time*
arrival_time	FLOAT	-1
wait_time	FLOAT	*total_waiting_time* The sum of all the *wait_time*
service_time	FLOAT	*total_service_time* The sum of all the *service_time*
departure_time	FLOAT	*total_solution_time* = total_travel_time + total_wait_time + total_service_time.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Handling Parameters

To define a problem, several considerations have to be done, to get consistent results. This section gives an insight of how parameters are to be considered.

- **Capacity and Demand Units Handling**
- **Locations**
- **Time Handling**
- **Factor Handling**

Capacity and Demand Units Handling

The *capacity* of a vehicle, can be measured in:

- Volume units like m^3.
- Area units like m^2 (when no stacking is allowed).
- Weight units like kg.
- Number of boxes that fit in the vehicle.
- Number of seats in the vehicle

The *demand* request of the pickup-deliver orders must use the same units as the units used in the vehicle's *capacity*.

To handle problems like: 10 (equal dimension) boxes of apples and 5 kg of feathers that are to be transported (not packed in boxes).

If the vehicle's *capacity* is measured by *boxes*, a conversion of *kg of feathers* to *equivalent number of boxes* is needed. If the vehicle's *capacity* is measured by *kg*, a conversion of *box of apples* to *equivalent number of kg* is needed.

Showing how the 2 possible conversions can be done

Let: - f_boxes: number of boxes that would be used for *1* kg of feathers. - a_weight: weight of *1* box of apples.

Capacity Units	apples	feathers
boxes	10	5 * f_boxes
kg	10 * a_weight	5

Locations

- When using the Euclidean signatures:
  - The vehicles have (x, y) pairs for start and ending locations.
  - The orders Have (x, y) pairs for pickup and delivery locations.
- When using a matrix:
  - The vehicles have identifiers for the start and ending locations.
  - The orders have identifiers for the pickup and delivery locations.
  - All the identifiers are indices to the given matrix.

Time Handling

The times are relative to 0

Suppose that a vehicle's driver starts the shift at 9:00 am and ends the shift at 4:30 pm and the service time duration is 10 minutes with 30 seconds.

All time units have to be converted

Meaning of 0	time units	9:00 am	4:30 pm	10 min 30 secs
0:00 am	hours	9	16.5	10.5 / 60 = 0.175
9:00 am	hours	0	7.5	10.5 / 60 = 0.175
0:00 am	minutes	9*60 = 54	16.5*60 = 990	10.5
9:00 am	minutes	0	7.5*60 = 540	10.5

Factor Handling

Warning

TODO

See Also

- **https://en.wikipedia.org/wiki/Vehicle_routing_problem**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

Not classified

- **pgr_bellmanFord - Experimental**
- **pgr_binaryBreadthFirstSearch - Experimental**
- **pgr_breadthFirstSearch - Experimental**
- **pgr_dagShortestPath - Experimental**
- **pgr_edwardMoore - Experimental**
- **pgr_stoerWagner - Experimental**
- **pgr_topologicalSort - Experimental**
- **pgr_transitiveClosure - Experimental**
- **pgr_turnRestrictedPath - Experimental**

pgr_bellmanFord - Experimental

`pgr_bellmanFord` — Returns the shortest path(s) using Bellman-Ford algorithm. In particular, the Bellman-Ford algorithm implemented by Boost.Graph.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.

- Documentation examples might need to be automatically generated.
- Might need a lot of feedback from the comunity.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

Bellman-Ford's algorithm, is named after Richard Bellman and Lester Ford, who first published it in 1958 and 1956, respectively. It is a graph search algorithm that computes shortest paths from a starting vertex (`start_vid`) to an ending vertex (`end_vid`) in a graph where some of the edge weights may be negative number. Though it is more versatile, it is slower than Dijkstra's algorithm/ This implementation can be used with a directed graph and an undirected graph.

The main characteristics are:

- Process is valid for edges with both positive and negative edge weights.
- Values are returned when there is a path.
  - When the start vertex and the end vertex are the same, there is no path. The agg_cost would be 0.
  - When the start vertex and the end vertex are different, and there exists a path between them without having a *negative cycle*. The agg_cost would be some finite value denoting the shortest distance between them.
  - When the start vertex and the end vertex are different, and there exists a path between them, but it contains a *negative cycle*. In such case, agg_cost for those vertices keep on decreasing furthermore, Hence agg_cost can't be defined for them.
  - When the start vertex and the end vertex are different, and there is no path. The agg_cost is ∞.
- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: $O(|\ start\_vids\ | * (\ V * E))$

Signatures

Summary

```
pgr_bellmanFord(edges_sql, from_vid,  to_vid  [, directed])
pgr_bellmanFord(edges_sql, from_vid,  to_vids [, directed])
pgr_bellmanFord(edges_sql, from_vids, to_vid  [, directed])
pgr_bellmanFord(edges_sql, from_vids, to_vids [, directed])

RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Using defaults

```
pgr_bellmanFord(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_bellmanFord(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)
```

One to One

```
pgr_bellmanFord(edges_sql, from_vid,  to_vid  [, directed])
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **undirected** graph

```
SELECT * FROM pgr_bellmanFord(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)
```

One to many

```
pgr_bellmanFord(edges_sql, from_vid,  to_vids [, directed])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{ 3, 5\} on an **undirected** graph

```
SELECT * FROM pgr_bellmanFord(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    2 |    1 |        0
   2 |        2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |       5 |    2 |    4 |    1 |        0
   4 |        2 |       5 |    5 |   -1 |    0 |        1
(4 rows)
```

Many to One

```
pgr_bellmanFord(edges_sql, from_vids, to_vid  [, directed])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertex 5 on a **directed** graph

```
SELECT * FROM pgr_bellmanFord(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   13 |    1 |        0
   4 |        2 |        11 |   12 |   15 |    1 |        1
   5 |        3 |        11 |    9 |    9 |    1 |        2
   6 |        4 |        11 |    6 |    8 |    1 |        3
   7 |        5 |        11 |    5 |   -1 |    0 |        4
(7 rows)
```

Many to Many

```
pgr_bellmanFord(edges_sql, from_vids, to_vids [, directed])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertices \{3, 5\} on an **undirected** graph

```
SELECT * FROM pgr_bellmanFord(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], ARRAY[3,5]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   8 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   9 |        1 |        11 |       3 |   11 |   13 |    1 |        0
  10 |        2 |        11 |       3 |   12 |   15 |    1 |        1
  11 |        3 |        11 |       3 |    9 |   16 |    1 |        2
  12 |        4 |        11 |       3 |    4 |    3 |    1 |        3
  13 |        5 |        11 |       3 |    3 |   -1 |    0 |        4
  14 |        1 |        11 |       5 |   11 |   13 |    1 |        0
  15 |        2 |        11 |       5 |   12 |   15 |    1 |        1
  16 |        3 |        11 |       5 |    9 |    9 |    1 |        2
  17 |        4 |        11 |       5 |    6 |    8 |    1 |        3
  18 |        5 |        11 |       5 |    5 |   -1 |    0 |        4
(18 rows)
```

Parameters

Description of the parameters of the signatures

Parameter	Type	Default	Description
edges_sql	TEXT		SQL query as described above.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner Query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Results Columns

Returns set of (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**

Column	Type	Description
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from `start_vid` to `end_vid`.
edge	BIGINT	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the path.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

See Also

- **https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_binaryBreadthFirstSearch - Experimental

`pgr_binaryBreadthFirstSearch` — Returns the shortest path(s) in a binary graph. Any graph whose edge-weights belongs to the set {0,X}, where 'X' is any non-negative real integer, is termed as a 'binary graph'.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- To-be experimental on v3.0.0

Description

It is well-known that the shortest paths between a single source and all other vertices can be found using Breadth First Search in O(|E|) in an unweighted graph, i.e. the distance is the minimal number of edges that you need to traverse from the source to another vertex. We can interpret such a graph also as a weighted graph, where every edge has the weight 1. If not all edges in

graph have the same weight, that we need a more general algorithm, like Dijkstra's Algorithm which runs in O(|E|log|V|) time.

However if the weights are more constrained, we can use a faster algorithm. This algorithm, termed as 'Binary Breadth First Search' as well as '0-1 BFS', is a variation of the standard Breadth First Search problem to solve the SSSP (single-source shortest path) problem in O(|E|), if the weights of each edge belongs to the set {0,X}, where 'X' is any non-negative real integer.

The main Characteristics are:

- Process is done only on 'binary graphs'. ('Binary Graph': Any graph whose edge-weights belongs to the set {0,X}, where 'X' is any non-negative real integer.)
- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: O(| start_vids | * |E|)

Signatures

```
pgr_binaryBreadthFirstSearch(edges_sql, start_vid,  end_vid  [, directed])
pgr_binaryBreadthFirstSearch(edges_sql, start_vid,  end_vids [, directed])
pgr_binaryBreadthFirstSearch(edges_sql, start_vids, end_vid  [, directed])
pgr_binaryBreadthFirstSearch(edges_sql, start_vids, end_vids [, directed])
RETURNS SET OF (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)
OR EMPTY SET
```

```
pgr_binaryBreadthFirstSearch(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Example:

From vertex 2 to vertex 3 on a **directed** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
    'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    0 |        0
   2 |        2 |    5 |    8 |    1 |        0
   3 |        3 |    6 |    9 |    1 |        1
   4 |        4 |    9 |   16 |    0 |        2
   5 |        5 |    4 |    3 |    0 |        2
   6 |        6 |    3 |   -1 |    0 |        2
(6 rows)
```

One to One

```
pgr_binaryBreadthFirstSearch(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid,
BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **undirected** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
    'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)
```

One to many

```
pgr_binaryBreadthFirstSearch(TEXT edges_sql, BIGINT start_vid, ARRAY[ANY_INTEGER] end_vids,
BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{3, 5\} on an **undirected** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
    'SELECT id, source, target, road_work as cost FROM roadworks',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    0 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        0
   3 |        3 |       3 |    6 |    5 |    1 |        1
   4 |        4 |       3 |    3 |   -1 |    0 |        2
   5 |        1 |       5 |    2 |    4 |    0 |        0
   6 |        2 |       5 |    5 |   -1 |    0 |        0
(6 rows)
```

Many to One

```
pgr_binaryBreadthFirstSearch(TEXT edges_sql, ARRAY[ANY_INTEGER] start_vids, BIGINT end_vid,
    BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertex 5 on a **directed** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
    'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    0 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        0
   3 |        1 |        11 |   11 |   13 |    1 |        0
   4 |        2 |        11 |   12 |   15 |    0 |        1
   5 |        3 |        11 |    9 |   16 |    0 |        1
   6 |        4 |        11 |    4 |    3 |    0 |        1
   7 |        5 |        11 |    3 |    2 |    1 |        1
   8 |        6 |        11 |    2 |    4 |    0 |        2
   9 |        7 |        11 |    5 |   -1 |    0 |        2
(9 rows)
```

Many to Many

```
pgr_binaryBreadthFirstSearch(TEXT edges_sql, ARRAY[ANY_INTEGER] start_vids, ARRAY[ANY_INTEGER] end_vids,
    BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertices \{3, 5\} on an **undirected** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
    'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
    ARRAY[2,11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    2 |    1 |        0
   2 |        2 |         2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |         2 |       5 |    2 |    4 |    0 |        0
   4 |        2 |         2 |       5 |    5 |   -1 |    0 |        0
   5 |        1 |        11 |       3 |   11 |   13 |    1 |        0
   6 |        2 |        11 |       3 |   12 |   15 |    0 |        1
   7 |        3 |        11 |       3 |    9 |   16 |    0 |        1
   8 |        4 |        11 |       3 |    4 |    3 |    0 |        1
   9 |        5 |        11 |       3 |    3 |   -1 |    0 |        1
  10 |        1 |        11 |       5 |   11 |   12 |    0 |        0
  11 |        2 |        11 |       5 |   10 |   10 |    1 |        0
  12 |        3 |        11 |       5 |    5 |   -1 |    0 |        1
(12 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.

Parameter	Type	Default	Description
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same start_vid to end_vid combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from start_v to node.

Example Data

This type of data is used on the examples of this page.

Edwards-Moore Algorithm is best applied when trying to answer queries such as the following:**"Find the path with the minimum number from Source to Destination"** Here: * *Source* = Source Vertex, *Destination* = Any arbitrary destination vertex * *X* is an event/property * Each edge in the graph is either "*X*" or "*Not X*" .

Example: "Find the path with the minimum number of road works from Source to Destination"

Here, a road under work(aka **road works**) means that part of the road is occupied for construction work/maintenance.

Here: * Edge (u , v) has weight = 0 if no road work is ongoing on the road from u to v. * Edge (u, v) has weight = 1 if road work is ongoing on the road from u to v.

Then, upon running the algorithm, we obtain the path with the minimum number of road works from the given source and destination.

Thus, the queries used in the previous section can be interpreted in this manner.

Table Data

The queries in the previous sections use the table ‘roadworks’. The data of the table:

```
DROP TABLE IF EXISTS roadworks CASCADE;
NOTICE:  table "roadworks" does not exist, skipping
DROP TABLE
CREATE table roadworks (
    id BIGINT not null primary key,
    source BIGINT,
    target BIGINT,
    road_work FLOAT,
    reverse_road_work FLOAT
);
CREATE TABLE
INSERT INTO roadworks(
  id, source, target, road_work, reverse_road_work) VALUES
  (1,  1,  2,  0,  0),
  (2,  2,  3,  -1,  1),
  (3,  3,  4,  -1,  0),
  (4,  2,  5,  0,  0),
  (5,  3,  6,  1,  -1),
  (6,  7,  8,  1,  1),
  (7,  8,  5,  0,  0),
  (8,  5,  6,  1,  1),
  (9,  6,  9,  1,  1),
  (10,  5,  10,  1,  1),
  (11,  6,  11,  1,  -1),
  (12,  10,  11,  0,  -1),
  (13,  11,  12,  1,  -1),
  (14,  10,  13,  1,  1),
  (15,  9,  12,  0,  0),
  (16,  4,  9,  0,  0),
  (17,  14,  15,  0,  0),
  (18,  16,  17,  0,  0);
INSERT 0 18
```

See Also

- **https://cp-algorithms.com/graph/01_bfs.html**
- **https://codeforces.com/blog/entry/22276**
- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Specialized_variants**

Indices and tables

- **Index**
- **Search Page**

pgr_breadthFirstSearch - Experimental

`pgr_breadthFirstSearch` — Returns the traversal order(s) using Breadth First Search algorithm.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.

- pgTap tests might be missing.
- Might need c/c++ coding.
- May lack documentation.
- Documentation if any might need to be rewritten.
- Documentation examples might need to be automatically generated.
- Might need a lot of feedback from the comunity.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

Availability

Description

Provides the Breadth First Search traversal order from a root vertex to a particular depth.

The main Characteristics are:

- The implementation will work on any type of graph.
- Provides the Breadth First Search traversal order from a source node to a target depth level
- Breath First Search Running time: O(E + V)

Signatures

```
pgr_breadthFirstSearch(Edges SQL, Root vid [, max_depth] [, directed])
pgr_breadthFirstSearch(Edges SQL, Root vids [, max_depth] [, directed])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Single Vertex

```
pgr_breadthFirstSearch(Edges SQL, Root vid [, max_depth] [, directed])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Breadth First Search traversal with root vertex 2

```
SELECT * FROM pgr_breadthFirstSearch(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    2
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         2 |    2 |   -1 |    0 |        0
   2 |     1 |         2 |    1 |    1 |    1 |        1
   3 |     1 |         2 |    5 |    4 |    1 |        1
   4 |     2 |         2 |    8 |    7 |    1 |        2
   5 |     2 |         2 |    6 |    8 |    1 |        2
   6 |     2 |         2 |   10 |   10 |    1 |        2
   7 |     3 |         2 |    7 |    6 |    1 |        3
   8 |     3 |         2 |    9 |    9 |    1 |        3
   9 |     3 |         2 |   11 |   11 |    1 |        3
  10 |     3 |         2 |   13 |   14 |    1 |        3
  11 |     4 |         2 |   12 |   15 |    1 |        4
  12 |     4 |         2 |    4 |   16 |    1 |        4
  13 |     5 |         2 |    3 |    3 |    1 |        5
(13 rows)
```

Multiple Vertices

```
pgr_breadthFirstSearch(Edges SQL, Root vids [, max_depth] [, directed])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

Example:
The Breadth First Search traverls starting on vertices \{11, 12\} with depth <= 2

```
SELECT * FROM pgr_breadthFirstSearch(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[11,12], max_depth := 2
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |        11 |   11 |   -1 |    0 |        0
   2 |     1 |        11 |   12 |   13 |    1 |        1
   3 |     2 |        11 |    9 |   15 |    1 |        2
   4 |     0 |        12 |   12 |   -1 |    0 |        0
   5 |     1 |        12 |    9 |   15 |    1 |        1
   6 |     2 |        12 |    6 |    9 |    1 |        2
   7 |     2 |        12 |    4 |   16 |    1 |        2
(7 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	SQL query described in **Inner query**.
Root vid	BIGINT	Identifier of the root vertex of the tree. • Used on **Single Vertex**.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • Used on **Multiple Vertices**. • For optimization purposes, any duplicated value is ignored.

Optional Parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit for depth of node in the tree • When value is Negative then **throws error**
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
depth	BIGINT	Depth of the node. • 0 when node = start_vid.
start_vid	BIGINT	Identifier of the root vertex. • In *Multiple Vertices* results are in ascending order.

Column	Type	Description
node	BIGINT	Identifier of `node` reached using `edge`.
edge	BIGINT	Identifier of the `edge` used to arrive to `node`. • `-1` when `node` = `start_vid`.
cost	FLOAT	Cost to traverse `edge`.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `node`.

Additional Examples

Undirected Graph

Example:

The Breadth First Search traverls starting on vertices `\{11, 12\}` with `depth <= 2` as well as considering the graph to be undirected.

```
SELECT * FROM pgr_breadthFirstSearch(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    ARRAY[11,12], max_depth := 2, directed := false
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |        11 |   11 |   -1 |    0 |        0
   2 |     1 |        11 |    6 |   11 |    1 |        1
   3 |     1 |        11 |   10 |   12 |    1 |        1
   4 |     1 |        11 |   12 |   13 |    1 |        1
   5 |     2 |        11 |    3 |    5 |    1 |        2
   6 |     2 |        11 |    5 |    8 |    1 |        2
   7 |     2 |        11 |    9 |    9 |    1 |        2
   8 |     2 |        11 |   13 |   14 |    1 |        2
   9 |     0 |        12 |   12 |   -1 |    0 |        0
  10 |     1 |        12 |   11 |   13 |    1 |        1
  11 |     1 |        12 |    9 |   15 |    1 |        1
  12 |     2 |        12 |    6 |   11 |    1 |        2
  13 |     2 |        12 |   10 |   12 |    1 |        2
  14 |     2 |        12 |    4 |   16 |    1 |        2
(14 rows)
```

Vertex Out Of Graph

Example:

The output of the function when a vertex not present in the graph is passed as a parameter.

```
SELECT * FROM pgr_breadthFirstSearch(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    -10
);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
(0 rows)
```

See Also

- The queries use the **Sample Data** network.
- **Boost: Breadth First Search algorithm documentation**
- **Wikipedia: Breadth First Search algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_dagShortestPath - Experimental

`pgr_dagShortestPath` — Returns the shortest path(s) for weighted directed acyclic graphs(DAG). In particular, the DAG shortest paths algorithm implemented by Boost.Graph.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

Shortest Path for Directed Acyclic Graph(DAG) is a graph search algorithm that solves the shortest path problem for weighted directed acyclic graph, producing a shortest path from a starting vertex (`start_vid`) to an ending vertex (`end_vid`).

This implementation can only be used with a **directed** graph with no cycles i.e. directed acyclic graph.

The algorithm relies on topological sorting the dag to impose a linear ordering on the vertices, and thus is more efficient for DAG's than either the Dijkstra or Bellman-Ford algorithm.

The main characteristics are:

- Process is valid for weighted directed acyclic graphs only. otherwise it will throw warnings.
- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* the non included values *(v, v)* is *0*
  - When the starting vertex and ending vertex are the different and there is no path:
    - The *agg_cost* the non included values *(u, v)* is \infty
- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: O(| start_vids | * (V + E))

Signatures

Summary

```
pgr_dagShortestPath(edges_sql, from_vid,  to_vid)
pgr_dagShortestPath(edges_sql, from_vid,  to_vids)
pgr_dagShortestPath(edges_sql, from_vids, to_vid)
pgr_dagShortestPath(edges_sql, from_vids, to_vids)

RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

One to One

```
pgr_dagShortestPath(edges_sql, from_vid,  to_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 1 to vertex 6

```
SELECT * FROM pgr_dagShortestPath(
    'SELECT id, source, target, cost FROM edge_table',
    1, 6
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    1 |    1 |    1 |        0
   2 |        2 |    2 |    4 |    1 |        1
   3 |        3 |    5 |    8 |    1 |        2
   4 |        4 |    6 |   -1 |    0 |        3
(4 rows)
```

One to Many

```
pgr_dagShortestPath(edges_sql, from_vid,  to_vids)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 1 to vertices \{ 5, 6\}

```
SELECT * FROM pgr_dagShortestPath(
    'SELECT id, source, target, cost FROM edge_table',
    1, ARRAY[5,6]
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    1 |    1 |    1 |        0
   2 |        2 |    2 |    4 |    1 |        1
   3 |        3 |    5 |   -1 |    0 |        2
   4 |        1 |    1 |    1 |    1 |        0
   5 |        2 |    2 |    4 |    1 |        1
   6 |        3 |    5 |    8 |    1 |        2
   7 |        4 |    6 |   -1 |    0 |        3
(7 rows)
```

Many to One

```
pgr_dagShortestPath(edges_sql, from_vids, to_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{1, 3\} to vertex 6

```
SELECT * FROM pgr_dagShortestPath(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[1,3], 6
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    1 |    1 |    1 |        0
   2 |        2 |    2 |    4 |    1 |        1
   3 |        3 |    5 |    8 |    1 |        2
   4 |        4 |    6 |   -1 |    0 |        3
   5 |        1 |    3 |    5 |    1 |        0
   6 |        2 |    6 |   -1 |    0 |        1
(6 rows)
```

Many to Many

```
pgr_dagShortestPath(edges_sql, from_vids, to_vids)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{1, 4\} to vertices \{12, 6\}

```
SELECT * FROM pgr_dagShortestPath(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[1, 4],ARRAY[12,6]
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    1 |    1 |    1 |        0
   2 |        2 |    2 |    4 |    1 |        1
   3 |        3 |    5 |    8 |    1 |        2
   4 |        4 |    6 |   -1 |    0 |        3
   5 |        1 |    1 |    1 |    1 |        0
   6 |        2 |    2 |    4 |    1 |        1
   7 |        3 |    5 |   10 |    1 |        2
   8 |        4 |   10 |   12 |    1 |        3
   9 |        5 |   11 |   13 |    1 |        4
  10 |        6 |   12 |   -1 |    0 |        5
  11 |        1 |    4 |   16 |    1 |        0
  12 |        2 |    9 |   15 |    1 |        1
  13 |        3 |   12 |   -1 |    0 |        2
(13 rows)
```

Parameters

Description of the parameters of the signatures

Parameter	Type	Default	Description
edges_sql	TEXT		SQL query as described above.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.

Inner Query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Results Columns

Returns set of (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.

Column	Type	Description
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from `start_v` to `node`.

See Also

- **https://en.wikipedia.org/wiki/Topological_sorting**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_edwardMoore - Experimental

`pgr_edwardMoore` — Returns the shortest path(s) using Edward-Moore algorithm. Edward Moore's Algorithm is an improvement of the Bellman-Ford Algorithm.

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Description

Edward Moore's Algorithm is an improvement of the Bellman-Ford Algorithm. It can compute the shortest paths from a single source vertex to all other vertices in a weighted directed graph. The main difference between Edward Moore's Algorithm and Bellman Ford's Algorithm lies in the run time.

The worst-case running time of the algorithm is$O(| V | * | E |)$ similar to the time complexity of Bellman-Ford algorithm. However, experiments suggest that this algorithm has an average running time complexity of $O(| E |)$ for random graphs. This is significantly faster in terms of computation speed.

Thus, the algorithm is at-best, significantly faster than Bellman-Ford algorithm and is at-worst, as good as Bellman-Ford algorithm

The main characteristics are:

- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* the non included values *(v, v)* is *0*
  - When the starting vertex and ending vertex are the different and there is no path:
    - The *agg_cost* the non included values *(u, v)* is \infty

- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: - Worst case: O(| V | * | E |) - Average case: O(| E |)

Signatures

```
pgr_edwardMoore(edges_sql, start_vid,  end_vid  [, directed])
pgr_edwardMoore(edges_sql, start_vid,  end_vids [, directed])
pgr_edwardMoore(edges_sql, start_vids, end_vid  [, directed])
pgr_edwardMoore(edges_sql, start_vids, end_vids [, directed])
RETURNS SET OF (seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)
OR EMPTY SET
```

```
pgr_edwardMoore(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Example:

From vertex 2 to vertex 3 on a **directed** graph

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
   5 |        5 |    4 |    3 |    1 |        4
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)
```

One to One

```
pgr_edwardMoore(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid,
BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertex 3 on an **undirected** graph

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)
```

One to many

```
pgr_edwardMoore(TEXT edges_sql, BIGINT start_vid, ARRAY[ANY_INTEGER] end_vids,
BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex 2 to vertices \{3, 5\} on an **undirected** graph

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    5 |    1 |        2
   4 |        4 |       3 |    3 |   -1 |    0 |        3
   5 |        1 |       5 |    2 |    4 |    1 |        0
   6 |        2 |       5 |    5 |   -1 |    0 |        1
(6 rows)
```

Many to One

```
pgr_edwardMoore(TEXT edges_sql, ARRAY[ANY_INTEGER] start_vids, BIGINT end_vid,
    BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertex 5 on a **directed** graph

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   13 |    1 |        0
   4 |        2 |        11 |   12 |   15 |    1 |        1
   5 |        3 |        11 |    9 |    9 |    1 |        2
   6 |        4 |        11 |    6 |    8 |    1 |        3
   7 |        5 |        11 |    5 |   -1 |    0 |        4
(7 rows)
```

Many to Many

```
pgr_edwardMoore(TEXT edges_sql, ARRAY[ANY_INTEGER] start_vids, ARRAY[ANY_INTEGER] end_vids,
    BOOLEAN directed:=true);
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertices \{2, 11\} to vertices \{3, 5\} on an **undirected** graph

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    2 |    1 |        0
   2 |        2 |         2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   4 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   5 |        1 |        11 |       3 |   11 |   11 |    1 |        0
   6 |        2 |        11 |       3 |    6 |    5 |    1 |        1
   7 |        3 |        11 |       3 |    3 |   -1 |    0 |        2
   8 |        1 |        11 |       5 |   11 |   11 |    1 |        0
   9 |        2 |        11 |       5 |    6 |    8 |    1 |        1
  10 |        3 |        11 |       5 |    5 |   -1 |    0 |        2
(10 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		Inner SQL query as described below.
start_vid	BIGINT		Identifier of the starting vertex of the path.
start_vids	ARRAY[BIGINT]		Array of identifiers of starting vertices.
end_vid	BIGINT		Identifier of the ending vertex of the path.
end_vids	ARRAY[BIGINT]		Array of identifiers of ending vertices.

Parameter	Type	Default	Description
directed	BOOLEAN	true	• When true Graph is considered *Directed* • When false the graph is considered as *Undirected*.

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (seq, path_id, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_id	INT	Path identifier. Has value **1** for the first of a path. Used when there are multiple paths for the same start_vid to end_vid combination.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Returned when multiple starting vetrices are in the query. • **Many to One** • **Many to Many**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many**
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from start_v to node.

Example Application

The examples of this section are based on the **Sample Data** network.

The examples include combinations from starting vertices 2 and 11 to ending vertices 3 and 5 in a directed and undirected graph with and with out reverse_cost.

Examples:
For queries marked as directed with cost and reverse_cost columns

The examples in this section use the following **Network for queries marked as directed and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    9 |    1 |        2
   4 |        4 |    9 |   16 |    1 |        3
```

```
  4 |    4 |  9 | 16 |  1 |    3
  5 |    5 |  4 |  3 |  1 |    4
  6 |    6 |  3 | -1 |  0 |    5
(6 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3,5]
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    9 |    1 |        2
   4 |        4 |       3 |    9 |   16 |    1 |        3
   5 |        5 |       3 |    4 |    3 |    1 |        4
   6 |        6 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |       5 |    2 |    4 |    1 |        0
   8 |        2 |       5 |    5 |   -1 |    0 |        1
(8 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   13 |    1 |        0
   2 |        2 |   12 |   15 |    1 |        1
   3 |        3 |    9 |   16 |    1 |        2
   4 |        4 |    4 |    3 |    1 |        3
   5 |        5 |    3 |   -1 |    0 |        4
(5 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   13 |    1 |        0
   2 |        2 |   12 |   15 |    1 |        1
   3 |        3 |    9 |    9 |    1 |        2
   4 |        4 |    6 |    8 |    1 |        3
   5 |        5 |    5 |   -1 |    0 |        4
(5 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   13 |    1 |        0
   4 |        2 |        11 |   12 |   15 |    1 |        1
   5 |        3 |        11 |    9 |    9 |    1 |        2
   6 |        4 |        11 |    6 |    8 |    1 |        3
   7 |        5 |        11 |    5 |   -1 |    0 |        4
(7 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    9 |    1 |        2
   4 |        4 |         2 |       3 |    9 |   16 |    1 |        3
   5 |        5 |         2 |       3 |    4 |    3 |    1 |        4
   6 |        6 |         2 |       3 |    3 |   -1 |    0 |        5
   7 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   8 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   9 |        1 |        11 |       3 |   11 |   13 |    1 |        0
  10 |        2 |        11 |       3 |   12 |   15 |    1 |        1
  11 |        3 |        11 |       3 |    9 |   16 |    1 |        2
  12 |        4 |        11 |       3 |    4 |    3 |    1 |        3
  13 |        5 |        11 |       3 |    3 |   -1 |    0 |        4
  14 |        1 |        11 |       5 |   11 |   13 |    1 |        0
  15 |        2 |        11 |       5 |   12 |   15 |    1 |        1
  16 |        3 |        11 |       5 |    9 |    9 |    1 |        2
  17 |        4 |        11 |       5 |    6 |    8 |    1 |        3
  18 |        5 |        11 |       5 |    5 |   -1 |    0 |        4
```

```
 18 |     5 |     11 |     5 |  5 | -1 |  6 |    4
(18 rows)
```

Examples:

For queries marked as `undirected` with `cost` and `reverse_cost` columns

The examples in this section use the following **Network for queries marked as undirected and cost and reverse_cost columns are used**

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    2 |    1 |        0
   2 |        2 |    3 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, 5,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    5 |    1 |        1
   3 |        3 |    3 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    11, 5,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    8 |    1 |        1
   3 |        3 |    5 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2,11], 5,
    FALSE
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   11 |    1 |        0
   4 |        2 |        11 |    6 |    8 |    1 |        1
   5 |        3 |        11 |    5 |   -1 |    0 |        2
(5 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    2 |    1 |        0
   2 |        2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |       5 |    2 |    4 |    1 |        0
   4 |        2 |       5 |    5 |   -1 |    0 |        1
(4 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    2 |    1 |        0
   2 |        2 |         2 |       3 |    3 |   -1 |    0 |        1
   3 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   4 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   5 |        1 |        11 |       3 |   11 |   11 |    1 |        0
   6 |        2 |        11 |       3 |    6 |    5 |    1 |        1
   7 |        3 |        11 |       3 |    3 |   -1 |    0 |        2
   8 |        1 |        11 |       5 |   11 |   11 |    1 |        0
   9 |        2 |        11 |       5 |    6 |    8 |    1 |        1
  10 |        3 |        11 |       5 |    5 |   -1 |    0 |        2
(10 rows)
```

Examples:

For queries marked as `directed` with `cost` column

The examples in this section use the following **Network for queries marked as directed and only cost column is used**

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    11, 3
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    11, 5
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2,11], 5
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, ARRAY[3,5]
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       5 |    2 |    4 |    1 |        0
   2 |        2 |       5 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
(2 rows)
```

Examples:

For queries marked as `undirected` with `cost` column

The examples in this section use the following **Network for queries marked as undirected and only cost column is used**

```
SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |    8 |    1 |        1
   3 |        3 |    6 |    5 |    1 |        2
   4 |        4 |    3 |   -1 |    0 |        3
(4 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, 5,
```

```
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    2 |    4 |    1 |        0
   2 |        2 |    5 |   -1 |    0 |        1
(2 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    11, 3,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    5 |    1 |        1
   3 |        3 |    3 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    11, 5,
    FALSE
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |   11 |    1 |        0
   2 |        2 |    6 |    8 |    1 |        1
   3 |        3 |    5 |   -1 |    0 |        2
(3 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2,11], 5,
    FALSE
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         2 |    2 |    4 |    1 |        0
   2 |        2 |         2 |    5 |   -1 |    0 |        1
   3 |        1 |        11 |   11 |   11 |    1 |        0
   4 |        2 |        11 |    6 |    8 |    1 |        1
   5 |        3 |        11 |    5 |   -1 |    0 |        2
(5 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    2, ARRAY[3,5],
    FALSE
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       3 |    2 |    4 |    1 |        0
   2 |        2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |       3 |    6 |    5 |    1 |        2
   4 |        4 |       3 |    3 |   -1 |    0 |        3
   5 |        1 |       5 |    2 |    4 |    1 |        0
   6 |        2 |       5 |    5 |   -1 |    0 |        1
(6 rows)

SELECT * FROM pgr_edwardMoore(
    'SELECT id, source, target, cost FROM edge_table',
    ARRAY[2, 11], ARRAY[3,5],
    FALSE
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         2 |       3 |    2 |    4 |    1 |        0
   2 |        2 |         2 |       3 |    5 |    8 |    1 |        1
   3 |        3 |         2 |       3 |    6 |    5 |    1 |        2
   4 |        4 |         2 |       3 |    3 |   -1 |    0 |        3
   5 |        1 |         2 |       5 |    2 |    4 |    1 |        0
   6 |        2 |         2 |       5 |    5 |   -1 |    0 |        1
   7 |        1 |        11 |       3 |   11 |   11 |    1 |        0
   8 |        2 |        11 |       3 |    6 |    5 |    1 |        1
   9 |        3 |        11 |       3 |    3 |   -1 |    0 |        2
  10 |        1 |        11 |       5 |   11 |   11 |    1 |        0
  11 |        2 |        11 |       5 |    6 |    8 |    1 |        1
  12 |        3 |        11 |       5 |    5 |   -1 |    0 |        2
(12 rows)
```

See Also

- **https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm**

Indices and tables

- **Index**
- **Search Page**

pgr_stoerWagner - Experimental

`pgr_stoerWagner` — Returns the weight of the min-cut of graph using stoerWagner algorithm. Function determines a min-cut and the min-cut weight of a connected, undirected graph implemented by Boost.Graph.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 2.3.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

In graph theory, the Stoer–Wagner algorithm is a recursive algorithm to solve the minimum cut problem in undirected weighted graphs with non-negative weights. The essential idea of this algorithm is to shrink the graph by merging the most intensive vertices, until the graph only contains two combined vertex sets. At each phase, the algorithm finds the minimum s-t cut for two vertices s and t chosen as its will. Then the algorithm shrinks the edge between s and t to search for non s-t cuts. The minimum cut found in all phases will be the minimum weighted cut of the graph.

A cut is a partition of the vertices of a graph into two disjoint subsets. A minimum cut is a cut for which the size or weight of the cut is not larger than the size of any other cut. For an unweighted graph, the minimum cut would simply be the cut with the least edges. For a weighted graph, the sum of all edges' weight on the cut determines whether it is a minimum cut.

The main characteristics are:

- Process is done only on edges with positive costs.
- It's implementation is only on **undirected** graph.
- Sum of the weights of all edges between the two sets is mincut.
  - A **mincut** is a cut having the least weight.
- Values are returned when graph is connected.
  - When there is no edge in graph then EMPTY SET is return.
  - When the graph is unconnected then EMPTY SET is return.
- Sometimes a graph has multiple min-cuts, but all have the same weight. The this function determines exactly one of the min-cuts as well as its weight.

- Running time: O(V*E + V^2*log V).

Signatures

```
pgr_stoerWagner(edges_sql)

RETURNS SET OF (seq, edge, cost, mincut)
OR EMPTY SET
```

Example:

- **TBD**

```
pgr_stoerWagner(TEXT edges_sql);
RETURNS SET OF (seq, edge, cost, mincut)
OR EMPTY SET
```

```
SELECT * FROM pgr_stoerWagner(
   'SELECT id, source, target, cost, reverse_cost
      FROM edge_table
    WHERE id < 17'
);
 seq | edge | cost | mincut
-----+------+------+--------
   1 |    1 |    1 |      1
(1 row)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		SQL query as described above.

Inner query

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* - When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, - When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, edge, cost, mincut)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Edges which divides the set of vertices into two.
cost	FLOAT	Cost to traverse of edge.
mincut	FLOAT	Min-cut weight of a undirected graph.

Additional Example:

```
SELECT * FROM pgr_stoerWagner(
   'SELECT id, source, target, cost, reverse_cost
      FROM edge_table
    WHERE id = 18'
);
 seq | edge | cost | mincut
-----+------+------+--------
   1 |   18 |    1 |      1
(1 row)
```

Use pgr_connectedComponents( ) function in query:

```
SELECT * FROM pgr_stoerWagner(
$$
 SELECT id, source, target, cost, reverse_cost FROM edge_table
   where source = any (ARRAY(SELECT node FROM pgr_connectedComponents(
                        'SELECT id, source, target, cost, reverse_cost FROM edge_table ')
                      WHERE component = 14)
                     )
                OR
         target = any (ARRAY(SELECT node FROM pgr_connectedComponents(
                        'SELECT id, source, target, cost, reverse_cost FROM edge_table ')
                      WHERE component = 14)
                     )
$$
 );
 seq | edge | cost | mincut
-----+------+------+--------
   1 |   17 |    1 |      1
(1 row)
```

See Also

- **https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner_algorithm**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_topologicalSort - Experimental

`pgr_topologicalSort` — Returns the linear ordering of the vertices(s) for weighted directed acyclic graphs(DAG). In particular, the topological sort algorithm implemented by Boost.Graph.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.

- Documentation examples might need to be automatically generated.
- Might need a lot of feedback from the comunity.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)
- **TBD**

Description

The topological sort algorithm creates a linear ordering of the vertices such that if edge (u,v) appears in the graph, then v comes before u in the ordering.

This implementation can only be used with a **directed** graph with no cycles i.e. directed acyclic graph.

The main characteristics are:

- Process is valid for directed acyclic graphs only. otherwise it will throw warnings.
- For optimization purposes, if there are more than one answer, the function will return one of them.
- The returned values are ordered in topological order:
- Running time: O((V + E))

Signatures

Summary

```
pgr_topologicalSort(edges_sql)

RETURNS SET OF (seq, sorted_v)
OR EMPTY SET
```

Example:
For a **directed** graph

```
SELECT * FROM pgr_topologicalsort(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table1'
);
 seq | sorted_v
-----+----------
   1 |        0
   2 |        1
   3 |        3
   4 |        2
(4 rows)
```

Parameters

Parameter	Type	Default	Description
edges_sql	TEXT		SQL query as described above.

Inner query

edges_sql:
an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* - When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.

Column	Type	Default	Description
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, sorted_v)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
sorted_v	BIGINT	Linear ordering of the vertices(ordered in topological order)

See Also

- **https://en.wikipedia.org/wiki/Topological_sorting**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_transitiveClosure - Experimental

pgr_transitiveClosure — Returns the transitive closure graph of the input graph. In particular, the transitive closure algorithm implemented by Boost.Graph.

Boost Graph Inside

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

The transitive_closure() function transforms the input graph g into the transitive closure graph tc.

This implementation can only be used with a **directed** graph with no cycles i.e. directed acyclic graph.

The main characteristics are:

- Process is valid for directed acyclic graphs only. otherwise it will throw warnings.
- The returned values are not ordered:
- Running time: O(|V||E|)

Signatures

Summary

The pgr_transitiveClosure function has the following signature:

```
pgr_transitiveClosure(Edges SQL)
RETURNS SETOF (id, vid, target_array)
```

Example:
Complete Graph of 3 vertexs

```
SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table1'
);
 seq | vid | target_array
-----+-----+--------------
   1 |   0 | {1,3,2}
   2 |   1 | {3,2}
   3 |   3 | {2}
   4 |   2 | {}
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	SQL query as described in **Inner query**

Inner query

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge *(source, target)* - When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge *(target, source)*, - When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL:
SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

RETURNS SETOF (seq, vid, target_array)

The function returns a single row. The columns of the row are:

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
vid	BIGINT	Identifier of the vertex.
target_array	ARRAY[BIGINT]	Array of identifiers of the vertices that are reachable from vertex v.

Additional Examples

Example:
Some sub graphs of the sample data

```
SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=2'
);
 seq | vid | target_array
-----+-----+--------------
   1 |   2 | {}
   2 |   3 | {2}
(2 rows)

SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=3'
);
 seq | vid | target_array
-----+-----+--------------
   1 |   3 | {}
   2 |   4 | {3}
(2 rows)

SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=2 or id=3'
);
 seq | vid | target_array
-----+-----+--------------
   1 |   2 | {}
   2 |   3 | {2}
   3 |   4 | {3,2}
(3 rows)

SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=11'
);
 seq | vid | target_array
-----+-----+--------------
   1 |   6 | {11}
   2 |  11 | {}
(2 rows)

-- q3
SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where cost=-1 or reverse_cost=-1'
);
 seq | vid | target_array
-----+-----+--------------
   1 |   2 | {}
   2 |   3 | {11,12,6,2}
   3 |   4 | {11,12,3,6,2}
   4 |   6 | {11,12}
   5 |  11 | {12}
   6 |  10 | {11,12}
   7 |  12 | {}
(7 rows)
```

See Also

- **https://en.wikipedia.org/wiki/Transitive_closure**
- The queries use the **Sample Data** network.

Indices and tables

- **Index**
- **Search Page**

pgr_turnRestrictedPath - Experimental

pgr_turnRestrictedPath

Warning

Possible server crash

- These functions might create a server crash

Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Availability

- Version 3.0.0
  - New **Experimental** function

Support

- **Supported versions:** current(**3.0**)

Description

- TBD

Signatures

- TBD

Parameters

- TBD

Inner query

- TBD

Result Columns

- TBD

Additional Examples

Example:

See Also

Indices and tables

- **Index**
- **Search Page**

See Also

Indices and tables

- **Index**

Release Notes

Release Notes

To see the full list of changes check the list of**Git commits** on Github.

Table of contents

pgRouting 3.0.0 Release Notes

Fixed Issues

- **#1153**: Renamed pgr_eucledianTSP to pgr_TSPeuclidean
- **#1188**: Removed CGAL dependency
- **#1002**: Fixed contraction issues:
  - **#1004**: Contracts when forbidden vertices do not belong to graph
  - **#1005**: Intermideate results eliminated
  - **#1006**: No loss of information

New functions

- Kruskal family
  - pgr_kruskal
  - pgr_kruskalBFS
  - pgr_kruskalDD
  - pgr_kruskalDFS
- Prim family
  - pgr_prim
  - pgr_primDD
  - pgr_primDFS
  - pgr_primBFS

Proposed moved to official on pgRouting

- aStar Family
  - pgr_aStar(one to many)
  - pgr_aStar(many to one)
  - pgr_aStar(many to many)
  - pgr_aStarCost(one to one)
  - pgr_aStarCost(one to many)
  - pgr_aStarCost(many to one)
  - pgr_aStarCost(many to many)
  - pgr_aStarCostMatrix(one to one)
  - pgr_aStarCostMatrix(one to many)
  - pgr_aStarCostMatrix(many to one)
  - pgr_aStarCostMatrix(many to many)
- bdAstar Family
  - pgr_bdAstar(one to many)
  - pgr_bdAstar(many to one)
  - pgr_bdAstar(many to many)
  - pgr_bdAstarCost(one to one)
  - pgr_bdAstarCost(one to many)
  - pgr_bdAstarCost(many to one)
  - pgr_bdAstarCost(many to many)
  - pgr_bdAstarCostMatrix(one to one)
  - pgr_bdAstarCostMatrix(one to many)
  - pgr_bdAstarCostMatrix(many to one)
  - pgr_bdAstarCostMatrix(many to many)
- bdDijkstra Family
  - pgr_bdDijkstra(one to many)
  - pgr_bdDijkstra(many to one)
  - pgr_bdDijkstra(many to many)
  - pgr_bdDijkstraCost(one to one)
  - pgr_bdDijkstraCost(one to many)
  - pgr_bdDijkstraCost(many to one)
  - pgr_bdDijkstraCost(many to many)
  - pgr_bdDijkstraCostMatrix(one to one)
  - pgr_bdDijkstraCostMatrix(one to many)
  - pgr_bdDijkstraCostMatrix(many to one)
  - pgr_bdDijkstraCostMatrix(many to many)
- Flow Family
  - pgr_pushRelabel(one to one)
  - pgr_pushRelabel(one to many)
  - pgr_pushRelabel(many to one)
  - pgr_pushRelabel(many to many)
  - pgr_edmondsKarp(one to one)
  - pgr_edmondsKarp(one to many)
  - pgr_edmondsKarp(many to one)

    - pgr_edmondsKarp(many to many)
    - pgr_boykovKolmogorov (one to one)
    - pgr_boykovKolmogorov (one to many)
    - pgr_boykovKolmogorov (many to one)
    - pgr_boykovKolmogorov (many to many)
    - pgr_maxCardinalityMatching
    - pgr_maxFlow
    - pgr_edgeDisjointPaths(one to one)
    - pgr_edgeDisjointPaths(one to many)
    - pgr_edgeDisjointPaths(many to one)
    - pgr_edgeDisjointPaths(many to many)
- Components family
    - pgr_connectedComponents
    - pgr_strongComponents
    - pgr_biconnectedComponents
    - pgr_articulationPoints
    - pgr_bridges
- Contraction:
    - Removed unnecessary column seq
    - Bug Fixes

New Experimental functions

- pgr_maxFlowMinCost
- pgr_maxFlowMinCost_Cost
- pgr_extractVertices
- pgr_turnRestrictedPath
- pgr_stoerWagner
- pgr_dagShortestpath
- pgr_topologicalSort
- pgr_transitiveClosure
- VRP category
    - pgr_pickDeliverEuclidean
    - pgr_pickDeliver
- Chinese Postman family
    - pgr_chinesePostman
    - pgr_chinesePostmanCost
- Breadth First Search family
    - pgr_breadthFirstSearch
    - pgr_binaryBreadthFirstSearch
- Bellman Ford family
    - pgr_bellmanFord
    - pgr_edwardMoore

Moved to legacy

- Experimental functions
    - pgr_labelGraph - Use the components family of functions instead.
    - Max flow - functions were renamed on v2.5.0
        - pgr_maxFlowPushRelabel
        - pgr_maxFlowBoykovKolmogorov
        - pgr_maxFlowEdmondsKarp
        - pgr_maximumcardinalitymatching
    - VRP
        - pgr_gsoc_vrppdtw
- TSP old signatures
- pgr_pointsAsPolygon
- pgr_alphaShape old signature

pgRouting 2.6.3 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.6.3** on Github.

Bug fixes

- **#1219** Implicit cast for via_path integer to text
- **#1193** Fixed pgr_pointsAsPolygon breaking when comparing strings in WHERE clause

- **#1185** Improve FindPostgreSQL.cmake

pgRouting 2.6.2 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.6.2** on Github.

Bug fixes

- **#1152** Fixes driving distance when vertex is not part of the graph
- **#1098** Fixes windows test
- **#1165** Fixes build for python3 and perl5

pgRouting 2.6.1 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.6.1** on Github.

- Fixes server crash on several functions.
  - pgr_floydWarshall
  - pgr_johnson
  - pgr_astar
  - pgr_bdAstar
  - pgr_bdDijstra
  - pgr_alphashape
  - pgr_dijkstraCostMatrix
  - pgr_dijkstra
  - pgr_dijkstraCost
  - pgr_drivingDistance
  - pgr_KSP
  - pgr_dijkstraVia (proposed)
  - pgr_boykovKolmogorov (proposed)
  - pgr_edgeDisjointPaths (proposed)
  - pgr_edmondsKarp (proposed)
  - pgr_maxCardinalityMatch (proposed)
  - pgr_maxFlow (proposed)
  - pgr_withPoints (proposed)
  - pgr_withPointsCost (proposed)
  - pgr_withPointsKSP (proposed)
  - pgr_withPointsDD (proposed)
  - pgr_withPointsCostMatrix (proposed)
  - pgr_contractGraph (experimental)
  - pgr_pushRelabel (experimental)
  - pgr_vrpOneDepot (experimental)
  - pgr_gsoc_vrppdtw (experimental)
  - Fixes for deprecated functions where also applied but not tested
- Removed compilation warning for g++8
- Fixed a fallthrugh on Astar and bdAstar.

pgRouting 2.6.0 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.6.0** on Github.

New fexperimental functions

- pgr_lineGraphFull

Bug fixes

- Fix pgr_trsp(text,integer,double precision,integer,double precision,boolean,boolean[,text])
  - without restrictions
    - calls pgr_dijkstra when both end points have a fraction IN (0,1)
    - calls pgr_withPoints when at least one fraction NOT IN (0,1)
  - with restrictions
    - calls original trsp code

Internal code

- Cleaned the internal code of trsp(text,integer,integer,boolean,boolean [, text])

    - Removed the use of pointers
    - Internal code can accept BIGINT
- Cleaned the internal code of withPoints

pgRouting 2.5.5 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.5.5** on Github.

Bug fixes

- Fixes driving distance when vertex is not part of the graph
- Fixes windows test
- Fixes build for python3 and perl5

pgRouting 2.5.4 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.5.4** on Github.

- Fixes server crash on several functions.
    - pgr_floydWarshall
    - pgr_johnson
    - pgr_astar
    - pgr_bdAstar
    - pgr_bdDijstra
    - pgr_alphashape
    - pgr_dijkstraCostMatrix
    - pgr_dijkstra
    - pgr_dijkstraCost
    - pgr_drivingDistance
    - pgr_KSP
    - pgr_dijkstraVia (proposed)
    - pgr_boykovKolmogorov (proposed)
    - pgr_edgeDisjointPaths (proposed)
    - pgr_edmondsKarp (proposed)
    - pgr_maxCardinalityMatch (proposed)
    - pgr_maxFlow (proposed)
    - pgr_withPoints (proposed)
    - pgr_withPointsCost (proposed)
    - pgr_withPointsKSP (proposed)
    - pgr_withPointsDD (proposed)
    - pgr_withPointsCostMatrix (proposed)
    - pgr_contractGraph (experimental)
    - pgr_pushRelabel (experimental)
    - pgr_vrpOneDepot (experimental)
    - pgr_gsoc_vrppdtw (experimental)
    - Fixes for deprecated functions where also applied but not tested
- Removed compilation warning for g++8
- Fixed a fallthrugh on Astar and bdAstar.

pgRouting 2.5.3 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.5.3** on Github.

Bug fixes

- Fix for postgresql 11: Removed a compilation error when compiling with postgreSQL

pgRouting 2.5.2 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.5.2** on Github.

Bug fixes

- Fix for postgresql 10.1: Removed a compiler condition

pgRouting 2.5.1 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.5.1** on Github.

Bug fixes

- Fixed prerequisite minimum version of: cmake

pgRouting 2.5.0 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.5.0** on Github.

enhancement:

- pgr_version is now on SQL language

Breaking change on:

- pgr_edgeDisjointPaths:
  - Added path_id, cost and agg_cost columns on the result
  - Parameter names changed
  - The many version results are the union of the one to one version

New Signatures:

- pgr_bdAstar(one to one)

New Proposed functions

- pgr_bdAstar(one to many)
- pgr_bdAstar(many to one)
- pgr_bdAstar(many to many)
- pgr_bdAstarCost(one to one)
- pgr_bdAstarCost(one to many)
- pgr_bdAstarCost(many to one)
- pgr_bdAstarCost(many to many)
- pgr_bdAstarCostMatrix
- pgr_bdDijkstra(one to many)
- pgr_bdDijkstra(many to one)
- pgr_bdDijkstra(many to many)
- pgr_bdDijkstraCost(one to one)
- pgr_bdDijkstraCost(one to many)
- pgr_bdDijkstraCost(many to one)
- pgr_bdDijkstraCost(many to many)
- pgr_bdDijkstraCostMatrix
- pgr_lineGraph
- pgr_lineGraphFull
- pgr_connectedComponents
- pgr_strongComponents
- pgr_biconnectedComponents
- pgr_articulationPoints
- pgr_bridges

Deprecated Signatures

- pgr_bdastar - use pgr_bdAstar instead

Renamed Functions

- pgr_maxFlowPushRelabel - use pgr_pushRelabel instead
- pgr_maxFlowEdmondsKarp -use pgr_edmondsKarp instead
- pgr_maxFlowBoykovKolmogorov - use pgr_boykovKolmogorov instead
- pgr_maximumCardinalityMatching - use pgr_maxCardinalityMatch instead

Deprecated function

- pgr_pointToEdgeNode

pgRouting 2.4.2 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.4.2** on Github.

Improvement

- Works for postgreSQL 10

Bug fixes

- Fixed: Unexpected error column “cname”
- Replace __linux__ with __GLIBC__ for glibc-specific headers and functions

pgRouting 2.4.1 Release Notes

To see the issues closed by this release see the **Git closed milestone for 2.4.1** on Github.

Bug fixes

- Fixed compiling error on macOS
- Condition error on pgr_withPoints

pgRouting 2.4.0 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.4.0** on Github.

New Signatures

- pgr_bdDijkstra

New Proposed Signatures

- pgr_maxFlow
- pgr_astar(one to many)
- pgr_astar(many to one)
- pgr_astar(many to many)
- pgr_astarCost(one to one)
- pgr_astarCost(one to many)
- pgr_astarCost(many to one)
- pgr_astarCost(many to many)
- pgr_astarCostMatrix

Deprecated Signatures

- pgr_bddijkstra - use pgr_bdDijkstra instead

Deprecated Functions

- pgr_pointsToVids

Bug fixes

- Bug fixes on proposed functions
  - pgr_withPointsKSP: fixed ordering
- TRSP original code is used with no changes on the compilation warnings

pgRouting 2.3.2 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.3.2** on Github.

Bug Fixes

- Fixed pgr_gsoc_vrppdtw crash when all orders fit on one truck.
- Fixed pgr_trsp:
  - Alternate code is not executed when the point is in reality a vertex
  - Fixed ambiguity on seq

pgRouting 2.3.1 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.3.1** on Github.

Bug Fixes

- Leaks on proposed max_flow functions
- Regression error on pgr_trsp
- Types discrepancy on pgr_createVerticesTable

pgRouting 2.3.0 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.3.0** on Github.

New Signatures

- pgr_TSP
- pgr_aStar

New Functions

- pgr_eucledianTSP

New Proposed functions

- pgr_dijkstraCostMatrix
- pgr_withPointsCostMatrix
- pgr_maxFlowPushRelabel(one to one)
- pgr_maxFlowPushRelabel(one to many)
- pgr_maxFlowPushRelabel(many to one)
- pgr_maxFlowPushRelabel(many to many)
- pgr_maxFlowEdmondsKarp(one to one)
- pgr_maxFlowEdmondsKarp(one to many)
- pgr_maxFlowEdmondsKarp(many to one)
- pgr_maxFlowEdmondsKarp(many to many)
- pgr_maxFlowBoykovKolmogorov (one to one)
- pgr_maxFlowBoykovKolmogorov (one to many)
- pgr_maxFlowBoykovKolmogorov (many to one)
- pgr_maxFlowBoykovKolmogorov (many to many)
- pgr_maximumCardinalityMatching
- pgr_edgeDisjointPaths(one to one)
- pgr_edgeDisjointPaths(one to many)
- pgr_edgeDisjointPaths(many to one)
- pgr_edgeDisjointPaths(many to many)
- pgr_contractGraph

Deprecated Signatures

- pgr_tsp - use pgr_TSP or pgr_eucledianTSP instead
- pgr_astar - use pgr_aStar instead

Deprecated Functions

- pgr_flip_edges
- pgr_vidsToDmatrix
- pgr_pointsToDMatrix
- pgr_textToPoints

pgRouting 2.2.4 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.2.4** on Github.

Bug Fixes

- Bogus uses of extern “C”
- Build error on Fedora 24 + GCC 6.0
- Regression error pgr_nodeNetwork

pgRouting 2.2.3 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.2.3** on Github.

Bug Fixes

- Fixed compatibility issues with PostgreSQL 9.6.

pgRouting 2.2.2 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.2.2** on Github.

Bug Fixes

- Fixed regression error on pgr_drivingDistance

pgRouting 2.2.1 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.2.1** on Github.

Bug Fixes

- Server crash fix on pgr_alphaShape
- Bug fix on With Points family of functions

pgRouting 2.2.0 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.2.0** on Github.

Improvements

- pgr_nodeNetwork
  - Adding a row_where and outall optional parameters
- Signature fix
  - pgr_dijkstra – to match what is documented

New Functions

- pgr_floydWarshall
- pgr_Johnson
- pgr_dijkstraCost(one to one)
- pgr_dijkstraCost(one to many)
- pgr_dijkstraCost(many to one)
- pgr_dijkstraCost(many to many)

Proposed functionality

- pgr_withPoints(one to one)
- pgr_withPoints(one to many)
- pgr_withPoints(many to one)
- pgr_withPoints(many to many)
- pgr_withPointsCost(one to one)
- pgr_withPointsCost(one to many)
- pgr_withPointsCost(many to one)
- pgr_withPointsCost(many to many)
- pgr_withPointsDD(single vertex)
- pgr_withPointsDD(multiple vertices)
- pgr_withPointsKSP
- pgr_dijkstraVia

Deprecated functions:

- pgr_apspWarshall use pgr_floydWarshall instead
- pgr_apspJohnson use pgr_Johnson instead
- pgr_kDijkstraCost use pgr_dijkstraCost instead
- pgr_kDijkstraPath use pgr_dijkstra instead

Renamed and deprecated function

- pgr_makeDistanceMatrix renamed to _pgr_makeDistanceMatrix

pgRouting 2.1.0 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.1.0** on Github.

New Signatures

- pgr_dijkstra(one to many)
- pgr_dijkstra(many to one)
- pgr_dijkstra(many to many)
- pgr_drivingDistance(multiple vertices)

Refactored

- pgr_dijkstra(one to one)
- pgr_ksp
- pgr_drivingDistance(single vertex)

Improvements

- pgr_alphaShape function now can generate better (multi)polygon with holes and alpha parameter.

Proposed functionality

- Proposed functions from Steve Woodbridge, (Classified as Convenience by the author.)
  - pgr_pointToEdgeNode - convert a point geometry to a vertex_id based on closest edge.
  - pgr_flipEdges - flip the edges in an array of geometries so the connect end to end.
  - pgr_textToPoints - convert a string of x,y;x,y;... locations into point geometries.
  - pgr_pointsToVids - convert an array of point geometries into vertex ids.
  - pgr_pointsToDMatrix - Create a distance matrix from an array of points.
  - pgr_vidsToDMatrix - Create a distance matrix from an array of vertix_id.
  - pgr_vidsToDMatrix - Create a distance matrix from an array of vertix_id.
- Added proposed functions from GSoc Projects:
  - pgr_vrppdtw
  - pgr_vrponedepot

Deprecated functions

- pgr_getColumnName
- pgr_getTableName
- pgr_isColumnCndexed
- pgr_isColumnInTable
- pgr_quote_ident
- pgr_versionless
- pgr_startPoint
- pgr_endPoint
- pgr_pointToId

No longer supported

- Removed the 1.x legacy functions

Bug Fixes

- Some bug fixes in other functions

Refactoring Internal Code

- A C and C++ library for developer was created
  - encapsulates postgreSQL related functions
  - encapsulates Boost.Graph graphs
    - Directed Boost.Graph
    - Undirected Boost.graph.
  - allow any-integer in the id's
  - allow any-numerical on the cost/reverse_cost columns
- Instead of generating many libraries: - All functions are encapsulated in one library - The library has the prefix 2-1-0

pgRouting 2.0.1 Release Notes

Minor bug fixes.

Bug Fixes

- No track of the bug fixes were kept.

pgRouting 2.0.0 Release Notes

To see the issues closed by this release see the **Git closed issues for 2.0.0** on Github.

With the release of pgRouting 2.0.0 the library has abandoned backwards compatibility to **pgRouting 1.x** releases. The main Goals for this release are:

- Major restructuring of pgRouting.
- Standardization of the function naming
- Preparation of the project for future development.

As a result of this effort:

- pgRouting has a simplified structure
- Significant new functionality has being added
- Documentation has being integrated
- Testing has being integrated
- And made it easier for multiple developers to make contributions.

Important Changes

- Graph Analytics - tools for detecting and fixing connection some problems in a graph
- A collection of useful utility functions
- Two new All Pairs Short Path algorithms (pgr_apspJohnson, pgr_apspWarshall)
- Bi-directional Dijkstra and A-star search algorithms (pgr_bdAstar, pgr_bdDijkstra)
- One to many nodes search (pgr_kDijkstra)
- K alternate paths shortest path (pgr_ksp)
- New TSP solver that simplifies the code and the build process (pgr_tsp), dropped “Gaul Library” dependency
- Turn Restricted shortest path (pgr_trsp) that replaces Shooting Star
- Dropped support for Shooting Star
- Built a test infrastructure that is run before major code changes are checked in
- Tested and fixed most all of the outstanding bugs reported against 1.x that existing in the 2.0-dev code base.
- Improved build process for Windows
- Automated testing on Linux and Windows platforms trigger by every commit
- Modular library design
- Compatibility with PostgreSQL 9.1 or newer
- Compatibility with PostGIS 2.0 or newer
- Installs as PostgreSQL EXTENSION
- Return types re factored and unified
- Support for table SCHEMA in function parameters
- Support for `st_` PostGIS function prefix
- Added `pgr_` prefix to functions and types
- Better documentation: **https://docs.pgrouting.org**
- shooting_star is discontinued

pgRouting 1.x Release Notes

To see the issues closed by this release see the **Git closed issues for 1.x** on Github. The following release notes have been copied from the previous `RELEASE_NOTES` file and are kept as a reference.

Changes for release 1.05

- Bug fixes

Changes for release 1.03

- Much faster topology creation
- Bug fixes

Changes for release 1.02

- Shooting* bug fixes