- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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 v3.3.4.

General

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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**, **Paragon Corporation** and a broad user community.

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

Licensing

The following licenses can be found in pgRouting:

License	
GNU General Public License v2.0 or later	Most features of pgRouting are available under **GNU General Public License v2.0 or later**.
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 this release (in alphabetical order)

Ashish Kumar, Cayetano Benavent, Daniel Kastl, Nitish Chauhan, Rajat Shinde, Regina Obe, Shobhit Chaurasia, Swapnil Joshi, 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, workshop.

Corporate Sponsors in this release (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**
- **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, Ashish Kumar, Cayetano Benavent, Christian Gonzalez, Daniel Kastl, Dave Potts, David Techer, Denis Rykov, Ema Miyawaki, Esteban Zimanyi, Florian Thurkow, Frederic Junod, Gerald Fenoy, Gudesa Venkata Sai Akhil, Hang Wu, Himanshu Raj, Imre Samu, Jay Mahadeokar, Jinfu Leng, Kai Behncke, Kishore Kumar, Ko Nagase, Mahmoud Sakr, Manikata Kondeti, Mario Basa, Martin Wiesenhaan, Maxim Dubinin, Maoguang Wang, Mohamed Bakli, Mohamed Zia, Mukul Priya, Nitish Chauhan, Rajat Shinde, Razequl Islam, Regina Obe, Rohith Reddy, Sarthak Agarwal, Shobhit Chaurasia, Sourabh Garg, Stephen Woodbridge, Swapnil Joshi, Sylvain Housseman, Sylvain Pasche, Veenit Kumar, 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
- Leopark
- 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 from 2.6 can be found at **https://github.com/pgRouting/pgrouting/wiki/Migration-Guide**.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

Installation

Table of Contents

Instructions for downloading and installing binaries for different operating systems, 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-3.3.4.tar.gz
cd pgrouting-3.3.4
```

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-3.3.4.tar.gz    https://github.com/pgRouting/pgrouting/archive/v3.3.4.tar.gz
```

Go to **Short Version** for more instructions on extracting tar ball and compiling pgRouting.

git

To download the repository

```
git clone git://github.com/pgRouting/pgrouting.git
cd pgrouting
git checkout v3.3.4
```

Go to **Short Version** for more instructions on compiling pgRouting (there is no tar ball involved while downloading pgRouting repository from GitHub).

Enabling and upgrading in the database

Enabling the database

pgRouting is a PostgreSQL extension and depends on PostGIS to provide functionalities to end user. Below given code demonstrates enabling PostGIS and pgRouting in the database.

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

Checking PostGIS and pgRouting version after enabling them in the database.

```
SELECT PostGIS_full_version();
SELECT * FROM pgr_version();
```

Upgrading the database

To upgrade pgRouting in the database to version 3.3.4 use the following command:

```
ALTER EXTENSION pgrouting UPDATE TO "3.3.4";
```

More information can be found in **https://www.postgresql.org/docs/current/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
    - Compiling with Boost 1.56 up to Boost 1.74 requires C++ Compiler with C++03 or C++11 standard support
    - Compiling with Boost 1.75 requires C++ Compiler with C++14 standard support

- Postgresql version = Supported versions by PostgreSQL
- The Boost Graph Library (BGL). Version >= 1.56
- 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 install postgresql-14
sudo apt install postgresql-server-dev-14
sudo apt install postgresql-14-postgis
```

Configuring PostgreSQL

Entering psql console

```
sudo systemctl start postgresql.service
sudo -i -u postgres
psql
```

To exit psql console

```
q
```

Entering psql console directly without switching roles can be done by the following commands

```
sudo -u postgres psql
```

Then use the above given method to exit out of the psql console

Checking PostgreSQL version

```
psql --version
```

or

Enter the psql console using above given method and then enter

```
SELECT VERSION();
```

Creating PostgreSQL role

```
sudo -i -u postgres
createuser --interactive
```

or

```
sudo -u postgres createuser --interactive
```

Default role provided by PostgreSQL is postgres. To create new roles you can use the above provided commands. The prompt will ask the user to type name of the role and then provide affirmation. Proceed with the steps and you will succeed in creating PostgreSQL role successfully.

To add password to the role or change previously created password of the role use the following commands

```
ALTER USER <role name> PASSWORD <password>
```

To get additional details on the flags associated with `createuser` below given command can be used

```
man createuser
```

Creating Database in PostgreSQL

```
sudo -i -u postgres
createdb <database name>
```

or

```
sudo -u postgres createdb <database name>
```

Connecting to a PostgreSQL Database

Enter the psql console and type the following commands

```
connect <database name>
```

Build dependencies

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

Optional dependencies

For documentation and testing

```
pip install sphinx
pip install sphinx-bootstrap-theme
sudo apt install texlive
sudo apt install doxygen
sudo apt install libtap-parser-sourcehandler-pgtap-perl
sudo apt install postgresql-14-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 been on the HTML files. Some variables for building 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

Variable	Default	Comment
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 cmake to create documentation before building pgRouting

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

Note

Most of the effort of the documentation has been 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 user's documentation
$ make all doc  # build both the code and the user's documentation
$ make doxy     # build only the developer's 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
```

To remove the build when the configuration changes, use the following code:

```
rm -rf build
```

and start the build process as mentioned previously.

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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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 **https://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**
Paragon Corporation	United States	**https://www.paragoncorporation.com**
Netlab	Capranica, Italy	**https://www.osgeo.org/service-providers/netlab/**

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

Sample Data

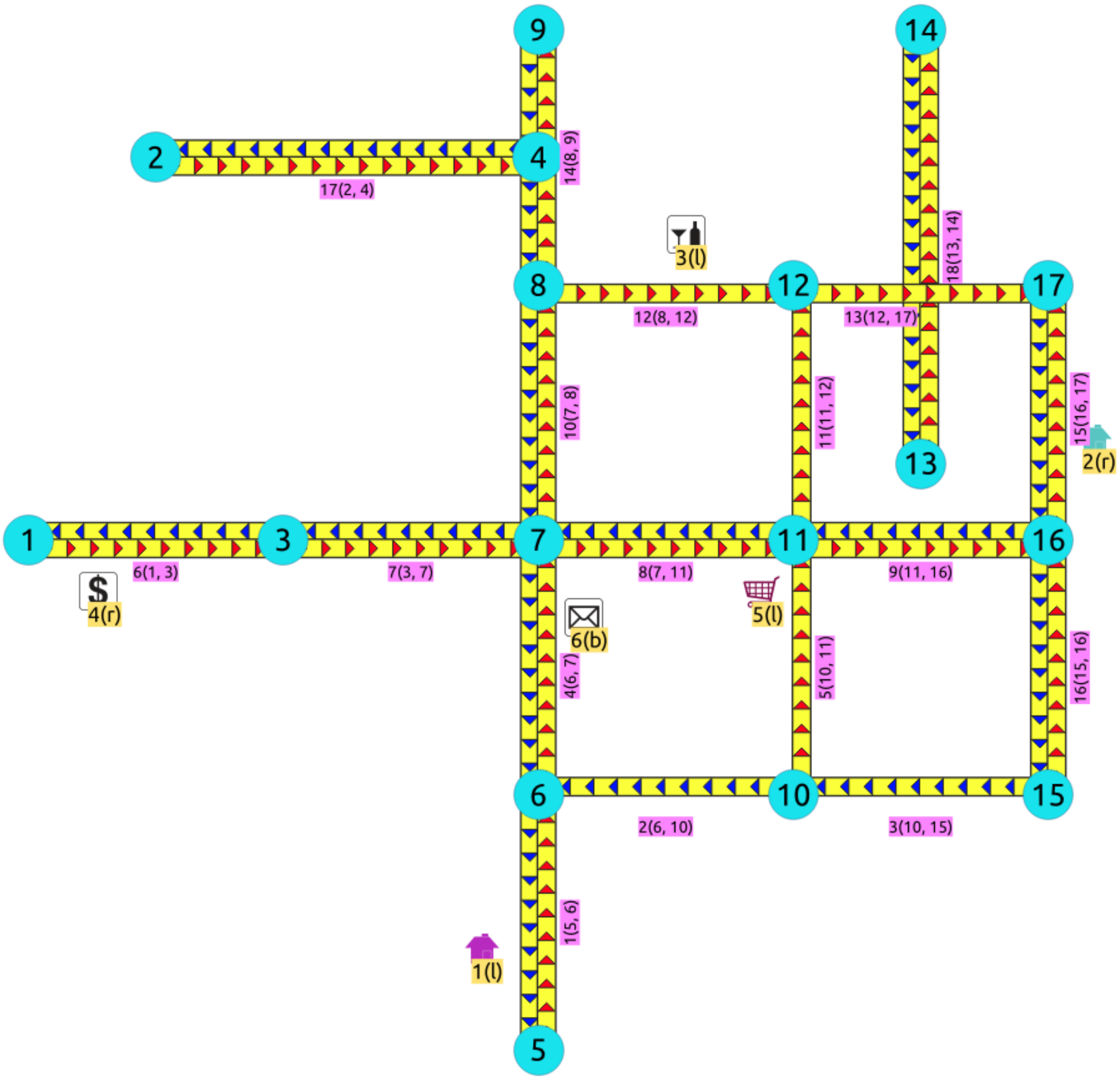
The documentation provides very simple example queries based on a small sample network that resembles a city. To be able to execute the mayority of the examples queries, follow the instructions bellow.

- **Main graph**

Main graph

A graph consists of a set of edges and a set of vertices.

The following city is to be inserted into the database:



Information known at this point is the geometry of the edges, cost values, cpacity values, category values and some locations that are not in the graph.

The process to have working topology starts by inserting the edges. After that everything else is calculated.

Edges

The database design for the documentation of pgRouting, keeps in the same row 2 segments, one in the direction of the geometry and the second in the oposite direction. Therfore some information has the `reverse_` prefix which corresponds to the segment on the oposite direction of the geometry.

Column	Description
`id`	A unique identifier.
`source`	Identifier of the starting vertex of the geometry`geom`.
`target`	Identifier of the ending vertex of the geometry`geom`
`cost`	Cost to traverse from *source* to `target`.
`reverse_cost`	Cost to traverse from *target* to `source`.
`capacity`	Flow capacity from *source* to `target`.
`reverse_capacity`	Flow capacity from *target* to `source`.
`category`	Flow capacity from *target* to `source`.
`reverse_category`	Flow capacity from *target* to `source`.
`x1`	\(x\) coordinate of the starting vertex of the geometry. • For convinience it is saved on the table but can be calculated as `ST_X(ST_StartPoint(geom))`.

Column	Description
y2	\(y\) coordinate of the ending vertex of the geometry. • For convinience it is saved on the table but can be calculated as ST_Y(ST_EndPoint(geom)).
geom	The geometry of the segments.

```
CREATE TABLE edges (
    id BIGSERIAL PRIMARY KEY,
    source BIGINT,
    target BIGINT,
    cost FLOAT,
    reverse_cost FLOAT,
    capacity BIGINT,
    reverse_capacity BIGINT,
    x1 FLOAT,
    y1 FLOAT,
    x2 FLOAT,
    y2 FLOAT,
    geom geometry
);
CREATE TABLE
```

Starting on PostgreSQL 12:

```
...
x1 FLOAT GENERATED ALWAYS AS (ST_X(ST_StartPoint(geom))) STORED,
y1 FLOAT GENERATED ALWAYS AS (ST_Y(ST_StartPoint(geom))) STORED,
x1 FLOAT GENERATED ALWAYS AS (ST_X(ST_EndPoint(geom))) STORED,
y1 FLOAT GENERATED ALWAYS AS (ST_Y(ST_EndPoint(geom))) STORED,
...
```

Optionally indexes on different columns can be created. The recomendation is to have

- id indexed.
- source and target columns indexed to speed up pgRouting queries.
- geom indexed to speed up gemetry processes that might be needed in the front end.

For this small example the indexes are skipped, except forid

Edges data

Inserting into the database the information of the edges:

```
INSERT INTO edges (
    cost, reverse_cost,
    capacity, reverse_capacity, geom) VALUES
( 1,  1,  80, 130,   ST_MakeLine(ST_POINT(2, 0), ST_POINT(2, 1))),
(-1,  1,  -1, 100,   ST_MakeLine(ST_POINT(2, 1), ST_POINT(3, 1))),
(-1,  1,  -1, 130,   ST_MakeLine(ST_POINT(3, 1), ST_POINT(4, 1))),
( 1,  1, 100,  50,   ST_MakeLine(ST_POINT(2, 1), ST_POINT(2, 2))),
( 1, -1, 130,  -1,   ST_MakeLine(ST_POINT(3, 1), ST_POINT(3, 2))),
( 1,  1,  50, 100,   ST_MakeLine(ST_POINT(0, 2), ST_POINT(1, 2))),
( 1,  1,  50, 130,   ST_MakeLine(ST_POINT(1, 2), ST_POINT(2, 2))),
( 1,  1, 100, 130,   ST_MakeLine(ST_POINT(2, 2), ST_POINT(3, 2))),
( 1,  1, 130,  80,   ST_MakeLine(ST_POINT(3, 2), ST_POINT(4, 2))),
( 1,  1, 130,  50,   ST_MakeLine(ST_POINT(2, 2), ST_POINT(2, 3))),
( 1, -1, 130,  -1,   ST_MakeLine(ST_POINT(3, 2), ST_POINT(3, 3))),
( 1, -1, 100,  -1,   ST_MakeLine(ST_POINT(2, 3), ST_POINT(3, 3))),
( 1, -1, 100,  -1,   ST_MakeLine(ST_POINT(3, 3), ST_POINT(4, 3))),
( 1,  1,  80, 130,   ST_MakeLine(ST_POINT(2, 3), ST_POINT(2, 4))),
( 1,  1,  80,  50,   ST_MakeLine(ST_POINT(4, 2), ST_POINT(4, 3))),
( 1,  1,  80,  80,   ST_MakeLine(ST_POINT(4, 1), ST_POINT(4, 2))),
( 1,  1, 130, 100,   ST_MakeLine(ST_POINT(0.5, 3.5), ST_POINT(1.999999999999, 3.5))),
( 1,  1,  50, 130,   ST_MakeLine(ST_POINT(3.5, 2.3), ST_POINT(3.5, 4)));
INSERT 0 18
```

Negative values on the cost, capacity and category means that the edge do not exist.

Vertices

The vertex information is calculated based on the identifier of the edge and the geometry and saved on a table. Saving all the information provided by **pgr_extractVertices – Proposed**:

```
SELECT * INTO vertices
FROM pgr_extractVertices('SELECT id, geom FROM edges ORDER BY id');
SELECT 17
```

In this case the because the CREATE statement was not used, the definition of an index on the table is needed.

```
CREATE SEQUENCE vertices_id_seq;
CREATE SEQUENCE
ALTER TABLE vertices ALTER COLUMN id SET DEFAULT nextval('vertices_id_seq');
ALTER TABLE
ALTER SEQUENCE vertices_id_seq OWNED BY vertices.id;
ALTER SEQUENCE
SELECT setval('vertices_id_seq', (SELECT coalesce(max(id)) FROM vertices));
 setval
--------
     17
(1 row)
```

The structure of the table is:

```
                         Table "public.vertices"
  Column   |       Type       | Collation | Nullable |               Default
-----------+------------------+-----------+----------+-------------------------------------
 id        | bigint           |           |          | nextval('vertices_id_seq'::regclass)
 in_edges  | bigint[]         |           |          |
 out_edges | bigint[]         |           |          |
 x         | double precision |           |          |
 y         | double precision |           |          |
 geom      | geometry         |           |          |
```

Vertices data

The saved information of the vertices is:

```
SELECT * FROM vertices;
 id | in_edges | out_edges |       x        |  y  |                    geom
----+----------+-----------+----------------+-----+--------------------------------------------
  1 |          | {6}       |              0 |   2 | 010100000000000000000000000000000000000040
  2 |          | {17}      |            0.5 | 3.5 | 0101000000000000000000E03F0000000000000C40
  3 | {6}      | {7}       |              1 |   2 | 0101000000000000000000F03F0000000000000040
  4 | {17}     |           | 1.999999999999 | 3.5 | 010100000068EEFFFFFFFFFF3F0000000000000C40
  5 |          | {1}       |              2 |   0 | 010100000000000000000000400000000000000000
  6 | {1}      | {2,4}     |              2 |   1 | 01010000000000000000000040000000000000F03F
  7 | {4,7}    | {8,10}    |              2 |   2 | 010100000000000000000000400000000000000040
  8 | {10}     | {12,14}   |              2 |   3 | 010100000000000000000000400000000000000840
  9 | {14}     |           |              2 |   4 | 010100000000000000000000400000000000001040
 10 | {2}      | {3,5}     |              3 |   1 | 01010000000000000000000840000000000000F03F
 11 | {5,8}    | {9,11}    |              3 |   2 | 010100000000000000000008400000000000000040
 12 | {11,12}  | {13}      |              3 |   3 | 010100000000000000000008400000000000000840
 13 |          | {18}      |            3.5 | 2.3 | 01010000000000000000000C406666666666660240
 14 | {18}     |           |            3.5 |   4 | 01010000000000000000000C400000000000001040
 15 | {3}      | {16}      |              4 |   1 | 01010000000000000000001040000000000000F03F
 16 | {9,16}   | {15}      |              4 |   2 | 010100000000000000000010400000000000000040
 17 | {13,15}  |           |              4 |   3 | 010100000000000000000010400000000000000840
(17 rows)
```

Here is where adding more columns to the vertices table can be done. Additional columns names and types will depend on the application.

The topology

This queries based on the vertices data create a topology by filling the`source` and `target` columns in the edges table.

```
/* -- set the source information */
UPDATE edges AS e
SET source = v.id, x1 = x, y1 = y
FROM vertices AS v
WHERE ST_StartPoint(e.geom) = v.geom;
UPDATE 18
/* -- set the target information */
UPDATE edges AS e
SET target = v.id, x2 = x, y2 = y
FROM vertices AS v
WHERE ST_EndPoint(e.geom) = v.geom;
UPDATE 18
```

Topology data

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

Points outside the graph

Points of interest

Some times the applications work "on the fly" starting from a location that is not a vertex in the graph. Those locations, in pgRrouting are called points of interest.

The information needed in the points of interest is `pid`, `edge_id`, `side`, `fraction`.

On this documentation there will be some 6 fixed points of interest and they will be stored on a table.

Column	Description
`pid`	A unique identifier.
`edge_id`	Identifier of the edge nearest edge that allows an arrival to the point.
`side`	Is it on the left, right or both sides of the segment `edge_id`
`fraction`	Where in the segment is the point located.
`geom`	The geometry of the points.
`newPoint`	The geometry of the points moved on top of the segment.

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

Points of interest fillup

Inserting the data of the points of interest:

```
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);
INSERT 0 6
```

Points of interest geometry

Calculating for visual purposes the points over the graph.

```
UPDATE pointsOfInterest SET geom = st_makePoint(x,y);
UPDATE 6
UPDATE pointsOfInterest
   SET newPoint = ST_LineInterpolatePoint(e.geom, fraction)
   FROM edges AS e WHERE edge_id = id;
UPDATE 6
```

Points of interest data

```
SELECT pid, edge_id, side, fraction,
      ST_AsText(geom), ST_AsText(newPoint)
FROM pointsOfInterest
ORDER BY pid;
 pid | edge_id | side | fraction |   st_astext    |  st_astext
-----+---------+------+----------+----------------+--------------
   1 |       1 | l    |      0.4 | POINT(1.8 0.4) | POINT(2 0.4)
   2 |      15 | r    |      0.4 | POINT(4.2 2.4) | POINT(4 2.4)
   3 |      12 | l    |      0.6 | POINT(2.6 3.2) | POINT(2.6 3)
   4 |       6 | r    |      0.3 | POINT(0.3 1.8) | POINT(0.3 2)
   5 |       5 | l    |      0.8 | POINT(2.9 1.8) | POINT(3 1.8)
   6 |       4 | b    |      0.7 | POINT(2.2 1.7) | POINT(2 1.7)
(6 rows)
```

Support tables

Combinations

Many functions can be used with a combinations of `(source, target)` pairs when wanting a route from `source` to `target`.

For convinence of this documentations, some combinations will be stored on a table:

```
CREATE TABLE combinations (
   source BIGINT,
   target BIGINT
);
CREATE TABLE
```

Inserting the data:

```
INSERT INTO combinations (
   source, target) VALUES
(5, 6),
(5, 10),
(6, 5),
(6, 15),
(6, 14);
INSERT 0 5
```

Combinations data

```
SELECT * FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

Restrictions

Some functions accept soft restrictions about the segments.

The creation of the restrictions table

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

Adding the restrictions

```
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);
INSERT 0 5
```

Restrictions used on **pgr_turnRestrictedPath - Experimental**

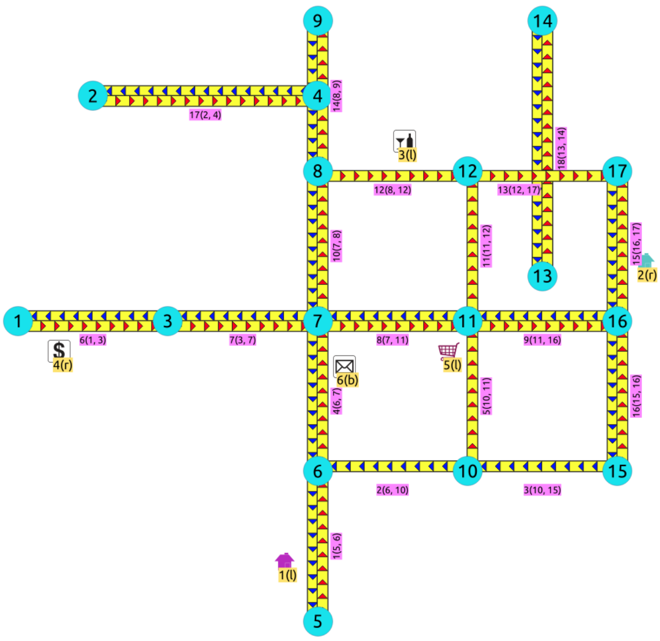
```
CREATE TABLE new_restrictions (
    id SERIAL PRIMARY KEY,
    path BIGINT[],
    cost float
);
CREATE TABLE
INSERT INTO new_restrictions (path, cost) VALUES
(ARRAY[4, 7], 100),
(ARRAY[8, 11], 100),
(ARRAY[7, 10], 100),
(ARRAY[3, 5, 9], 4),
(ARRAY[9, 16], 100);
INSERT 0 5
```

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.

Directed graph with `cost` and `reverse_cost`

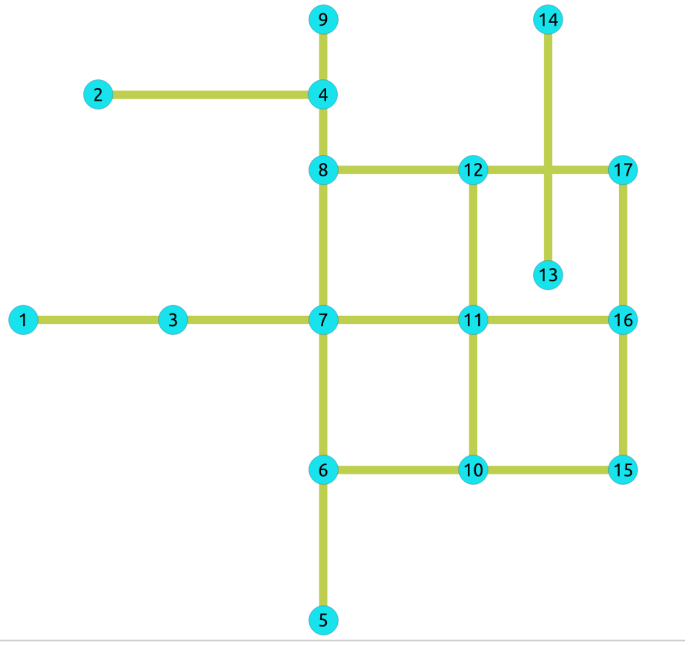
When working with city networks, this is recommended for point of view of vehicles.



Directed, with cost and reverse_cost

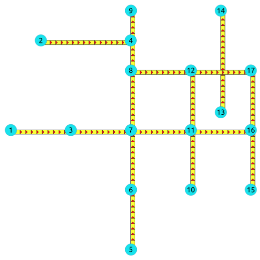
Undirected graph with `cost` and `reverse_cost`

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



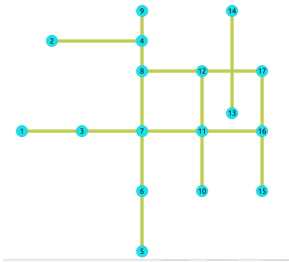
Undirected, with cost and reverse cost

Directed graph with `cost`



Directed, with cost

Undirected graph with `cost`



Undirected, with cost

Pick & Deliver Data

This data example **lc101** is from data published at **https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/**

The vehicles

There are 25 vehciles in the problem all with the same characteristics.

```
CREATE TABLE v_lc101(
  id BIGINT NOT NULL primary key,
  capacity BIGINT DEFAULT 200,
  start_x FLOAT DEFAULT 30,
  start_y FLOAT DEFAULT 50,
  start_open INTEGER DEFAULT 0,
  start_close INTEGER DEFAULT 1236);
CREATE TABLE
/* create 25 vehciles */
INSERT INTO v_lc101 (id)
(SELECT * FROM generate_series(1, 25));
INSERT 0 25
```

The original orders

The data comes in different rows for the pickup and the delivery of the same order.

```
CREATE table lc101_c(
  id BIGINT not null primary key,
  x DOUBLE PRECISION,
  y DOUBLE PRECISION,
  demand INTEGER,
  open INTEGER,
  close INTEGER,
  service INTEGER,
  pindex BIGINT,
  dindex BIGINT
);
CREATE TABLE
/* the original data */
INSERT INTO lc101_c(
  id,    x,    y, demand, open, close, service, pindex, dindex) VALUES
(  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,   -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);
INSERT 0 106
```

The orders

The original data needs to be converted to an appropiate table:

```
WITH deliveries AS (SELECT * FROM lc101_c WHERE dindex = 0)
SELECT
  row_number() over() AS id, p.demand,
  p.id as p_node_id, p.x AS p_x, p.y AS p_y, p.open AS p_open, p.close as p_close, p.service as p_service,
  d.id as d_node_id, d.x AS d_x, d.y AS d_y, d.open AS d_open, d.close as d_close, d.service as d_service
INTO c_lc101
FROM deliveries as d JOIN lc101_c as p ON (d.pindex = p.id);
SELECT 53
SELECT * FROM c_lc101 LIMIT 1;
 id | demand | p_node_id | p_x | p_y | p_open | p_close | p_service | d_node_id | d_x | d_y | d_open | d_close | d_service
----+--------+-----------+-----+-----+--------+---------+-----------+-----------+-----+-----+--------+---------+-----------
  1 |     10 |         3 |  42 |  66 |     65 |     146 |        90 |        75 |  45 |  65 |    997 |    1068 |        90
(1 row)
```

Pgrouting Concepts

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgRouting Concepts

This is a simple guide that go through some of the steps for getting started with pgRouting. This guide covers:

Graphs

Graph definition

A graph is an ordered pair\(G = (V ,E)\) where:

- \(V\) is a set of vertices, also called nodes.
- \(E \subseteq \{(u, v) \mid u , v \in V \}\)

There are different kinds of graphs:

- Undirected graph
  - \(E \subseteq \{(u, v) \mid u , v \in V\}\)
- Undirected simple graph
  - \(E \subseteq \{(u, v) \mid u , v \in V, u \neq v\}\)
- Directed graph
  - \(E \subseteq \{(u, v) \mid (u , v) \in (V X V) \}\)
- Directed simple graph
  - \(E \subseteq \{(u, v) \mid (u , v) \in (V X V), u \neq v\}\)

Graphs:

- Do not have geometries.
- Some graph theory problems require graphs to have weights, called**cost** in pgRouting.

In pgRouting there are several ways to represent a graph on the database:

- With `cost`
  - (`id`, `source`, `target`, `cost`)
- With `cost` and `reverse_cost`
  - (`id`, `source`, `target`, `cost`, `reverse_cost`)

Where:

Column	Description
`id`	Identifier of the edge. Requirement to use the database in a consistent. manner.
`source`	Identifier of a vertex.
`target`	Identifier of a vertex.
`cost`	Weight of the edge (`source`, `target`): • When negative the edge (`source`, `target`) do not exist on the graph. • `cost` must exist in the query.
`reverse_cost`	Weight of the edge (`target`, `source`) • When negative the edge (`target`, `source`) do not exist on the graph.

The decision of the graph to be**directed** or **undirected** is done when executing a pgRouting algorithm.

Graph with `cost`

The weighted directed graph, \(G_d(V,E)\):

- Graph data is obtained with a query

  `SELECT id, source, target, cost FROM edges`

- the set of edges\(E\)
  - \(E = \{(source_{id}, target_{id}, cost_{id}) \text{ when } cost_{id} \ge 0 \}\)
  - Edges where `cost` is non negative are part of the graph.

- the set of vertices V
    - $V = \{source_{id} \cup target_{id}\}$
    - All vertices in `source` and `target` are part of the graph.

Directed graph

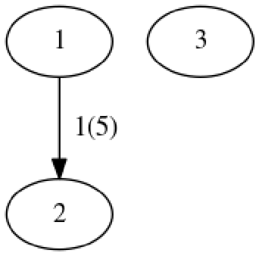
In a directed graph the edge $(source_{id}, target_{id}, cost_{id})$ has directionality: $source_{id} \rightarrow target_{id}$

For the following data:

```
SELECT *
FROM (VALUES (1, 1, 2, 5), (2, 1, 3, -3))
    AS t(id, source, target, cost);
 id | source | target | cost
----+--------+--------+------
  1 |      1 |      2 |    5
  2 |      1 |      3 |   -3
(2 rows)
```

Edge 2 ($1 \rightarrow 3$) is not part of the graph.

The data is representing the following graph:



Undirected graph

In an undirected graph the edge $(source_{id}, target_{id}, cost_{id})$ does not have directionality: $source_{id} \frac{\;\;\;\;\;}{} target_{id}$

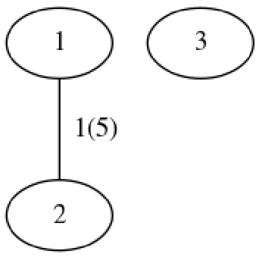
- In terms of a directed graph is like having two edges: $source_{id} \leftrightarrow target_{id}$

For the following data:

```
SELECT *
FROM (VALUES (1, 1, 2, 5), (2, 1, 3, -3))
    AS t(id, source, target, cost);
 id | source | target | cost
----+--------+--------+------
  1 |      1 |      2 |    5
  2 |      1 |      3 |   -3
(2 rows)
```

Edge 2 ($1 \frac{\;\;\;\;\;}{} 3$) is not part of the graph.

The data is representing the following graph:



Graph with `cost` and `reverse_cost`

The weighted directed graph, $G_d(V,E)$, is defined by:

- Graph data is obtained with a query

  `SELECT id, source, target, cost, reverse_cost FROM edges`

- The set of edges E:

- \(E = \begin{split} \begin{align} & {\{(source_{id}, target_{id}, cost_{id}) \text{ when } cost_{id} >=0 \}} \\ & \cup \\ & {\{(target_{id}, source_{id}, reverse_cost_{id}) \text{ when } reverse_cost_{id} >=0 \}} \end{align} \end{split}\)
    - Edges \((source \rightarrow target)\) where `cost` is non negative are part of the graph.
    - Edges \((target \rightarrow source)\) where `reverse_cost` is non negative are part of the graph.
- The set of vertices \(V\):
    - \(V = \{source_{id} \cup target_{id}\}\)
    - All vertices in `source` and `target` are part of the graph.

Directed graph

In a directed graph both edges have directionality

- edge \((source_{id}, target_{id}, cost_{id})\) has directionality: \(source_{id} \rightarrow target_{id}\)
- edge \((target_{id}, source_{id}, reverse_cost_{id})\) has directionality: \(target_{id} \rightarrow source_{id}\)

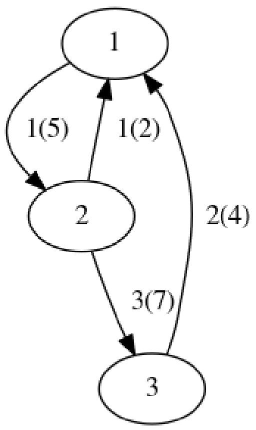
For the following data:

```
SELECT *
FROM (VALUES (1, 1, 2, 5, 2), (2, 1, 3, -3, 4), (3, 2, 3, 7, -1))
     AS t(id, source, target, cost, reverse_cost);
 id | source | target | cost | reverse_cost
----+--------+--------+------+-------------
  1 |      1 |      2 |    5 |            2
  2 |      1 |      3 |   -3 |            4
  3 |      2 |      3 |    7 |           -1
(3 rows)
```

Edges not part of the graph:

- \(2\) (\(1 \rightarrow 3\))
- \(3\) (\(3 \rightarrow 2\))

The data is representing the following graph:



Undirected graph

In a directed graph both edges do not have directionality

- Edge \((source_{id}, target_{id}, cost_{id})\) is \(source_{id} \frac{\;\;\;\;\;}{} target_{id}\)
- Edge \((target_{id}, source_{id}, reverse_cost_{id})\) is \(target_{id} \frac{\;\;\;\;\;}{} source_{id}\)
- In terms of a directed graph is like having four edges:
    - \(source_i \leftrightarrow target_i\)
    - \(target_i \leftrightarrow source_i\)

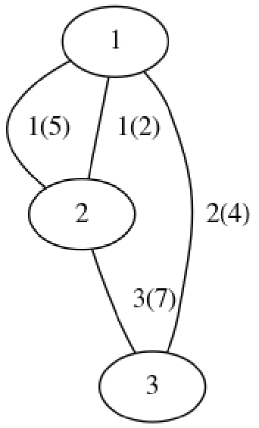
For the following data:

```
SELECT *
FROM (VALUES (1, 1, 2, 5, 2), (2, 1, 3, -3, 4), (3, 2, 3, 7, -1))
     AS t(id, source, target, cost, reverse_cost);
 id | source | target | cost | reverse_cost
----+--------+--------+------+-------------
  1 |      1 |      2 |    5 |            2
  2 |      1 |      3 |   -3 |            4
  3 |      2 |      3 |    7 |           -1
(3 rows)
```

Edges not part of the graph:

- \(2\) (\(1 \frac{\;\;\;\;\;}{} 3\))
- \(3\) (\(3 \frac{\;\;\;\;\;}{} 2\))

The data is representing the following graph:

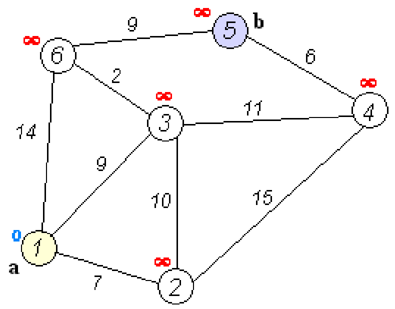


Graphs without geometries

Personal relationships, genealogy, file dependency problems can be solved using pgRouting. Those problems, normally, do not come with geometries associated with the graph.

Wiki example

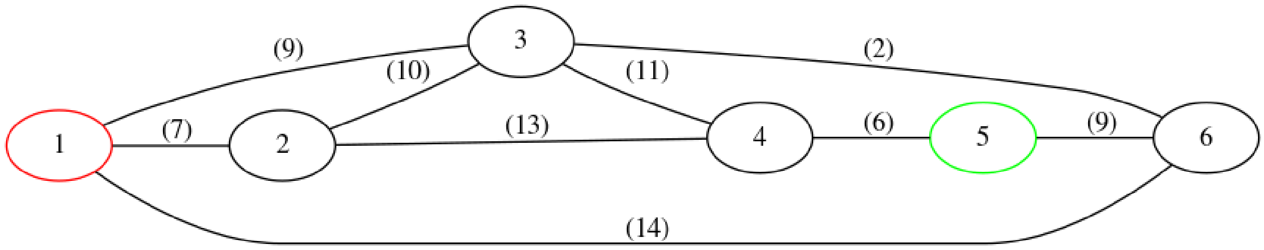
Solve the example problem taken from **wikipedia**):



Where:

- Problem is to find the shortest path from \(1\) to \(5\).
- Is an undirected graph.
- Although visually looks like to have geometries, the drawing is not to scale.
  - No geometries associated to the vertices or edges
- Has 6 vertices \(\{1,2,3,4,5,6\}\)
- Has 9 edges:

  \(\begin{split} \begin{align} E = & \{(1,2,7), (1,3,9), (1,6,14), \\ & (2,3,10), (2,4,13), \\ & (3,4,11), (3,6,2), \\ & (4,5,6), \\ & (5,6,9) \} \end{align} \end{split}\)
- The graph can be represented in many ways for example:



Prepare the database

Create a database for the example, access the database and install pgRouting:

```
$ createdb wiki
$ psql wiki
wiki =# CREATE EXTENSION pgRouting CASCADE;
```

Create a table

The basic elements needed to perform basic routing on an undirected graph are:

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.
cost	**ANY-NUMERICAL**	Weight of the edge (source, target)

Where:

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

Using this table design for this example:

```
CREATE TABLE wiki (
  id SERIAL,
  source INTEGER,
  target INTEGER,
  cost INTEGER);
CREATE TABLE
```

Insert the data

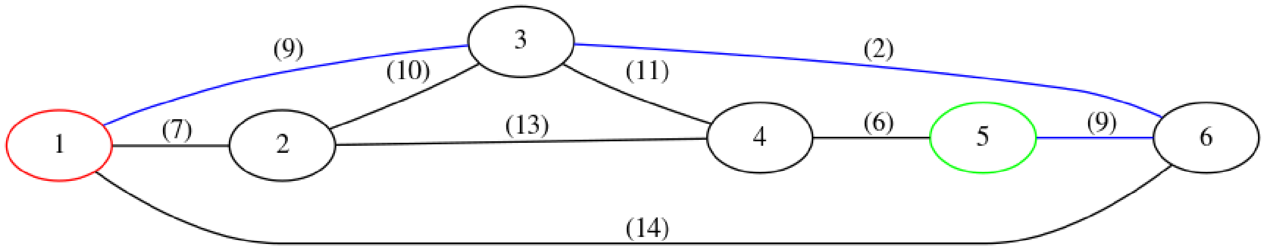
```
INSERT INTO wiki (source, target, cost) VALUES
(1, 2, 7),  (1, 3, 9), (1, 6, 14),
(2, 3, 10), (2, 4, 15),
(3, 6, 2),  (3, 4, 11),
(4, 5, 6),
(5, 6, 9);
INSERT 0 9
```

Find the shortest path

To solve this example **pgr_dijkstra** is used:

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM wiki',
  1, 5, false);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    1 |    2 |    9 |        0
   2 |        2 |    3 |    6 |    2 |        9
   3 |        3 |    6 |    9 |    9 |       11
   4 |        4 |    5 |   -1 |    0 |       20
(4 rows)
```

To go from \(1\) to \(5\) the path goes thru the following vertices:\(1 \rightarrow 3 \rightarrow 6 \rightarrow 5\)



Vertex information

To obtain the vertices information, use **pgr_extractVertices – Proposed**

```
SELECT id, in_edges, out_edges
FROM pgr_extractVertices('SELECT id, source, target FROM wiki');
 id | in_edges | out_edges
----+----------+-----------
  3 | {2,4}    | {6,7}
  5 | {8}      | {9}
  4 | {5,7}    | {8}
  2 | {1}      | {4,5}
  1 |          | {1,2,3}
  6 | {3,6,9}  |
(6 rows)
```

Graphs with geometries

- **Create a routing Database**
- **Load Data**
- **Build a routing topology**
- **Adjust costs**
  - **Update costs to length of geometry**
  - **Update costs based on codes**

Create a routing Database

The first step is to create a database and load pgRouting in the database.

Typically create a database for each project.

Once having the database to work in, load your data and build the routing application in that database.

```
createdb sampledata
psql sampledata -c "CREATE EXTENSION pgrouting CASCADE"
```

Load Data

There are several ways to load your data into pgRouting.

- Manually creating a database.
  - **Graphs without geometries**
  - **Sample Data**: a small graph used on the documentation examples
- Using **osm2pgrouting**

There are various open source tools that can help, like:

shp2pgsql:
- postgresql shapefile loader

ogr2ogr:
- vector data conversion utility

osm2pgsql:
- load OSM data into postgresql

Please note that these tools will **not** import the data in a structure compatible with pgRouting and when this happens the topology needs to be adjusted.

- Breakup a segments on each segment-segment intersection
- When missing, add columns and assign values to `source`, `target`, `cost`, `reverse_cost`.
- Connect a disconnected graph.
- Create the complete graph topology
- Create one or more graphs based on the application to be developed.

- Create a contracted graph for the high speed roads
- Create graphs per state/country

In few words:

> Prepare the graph

What and how to prepare the graph, will depend on the application and/or on the quality of the data and/or on how close the information is to have a topology usable by pgRouting and/or some other factors not mentioned.

The steps to prepare the graph involve geometry operations using **PostGIS** and some others involve graph operations like **pgr_contraction** to contract a graph.

The **workshop** has a step by step on how to prepare a graph using Open Street Map data, for a small application.

The use of indexes on the database design in general:

- Have the geometries indexed.
- Have the identifiers columns indexed.

Please consult the **PostgreSQL** documentation and the **PostGIS** documentation.

Build a routing topology

The basic information to use the majority of the pgRouting functions `id, source, target, cost, [reverse_cost]` is what in pgRouting is called the routing topology.

`reverse_cost` is optional but strongly recommended to have in order to reduce the size of the database due to the size of the geometry columns. Having said that, in this documentation `reverse_cost` is used in this documentation.

When the data comes with geometries and there is no routing topology, then this step is needed.

All the start and end vertices of the geometries need an identifier that is to be stored in a `source` and `target` columns of the table of the data. Likewise, `cost` and `reverse_cost` need to have the value of traversing the edge in both directions.

If the columns do not exist they need to be added to the table in question. (see **ALTER TABLE**)

The function **pgr_extractVertices – Proposed** is used to create a vertices table based on the edge identifier and the geometry of the edge of the graph.

Finally using the data stored on the vertices tables the `source` and `target` are filled up.

See **Sample Data** for an example for building a topology.

Data coming from OSM and using **osm2pgrouting** as an import tool, comes with the routing topology. See an example of using `osm2pgrouting` on the **workshop**.

Adjust costs

For this example the `cost` and `reverse_cost` values are going to be the double of the length of the geometry.

Update costs to length of geometry

Suppose that `cost` and `reverse_cost` columns in the sample data represent:

- \(1\) when the edge exists in the graph
- \(-1\) when the edge does not exist in the graph

Using that information updating to the length of the geometries:

```
UPDATE edges SET
cost = sign(cost) * ST_length(geom) * 2,
reverse_cost = sign(reverse_cost) * ST_length(geom) * 2;
UPDATE 18
```

Which gives the following results:

```
SELECT id, cost, reverse_cost FROM edges;
 id |        cost        |    reverse_cost
----+--------------------+--------------------
  6 |                  2 |                  2
  7 |                  2 |                  2
  4 |                  2 |                  2
  5 |                  2 |                 -2
  8 |                  2 |                  2
 12 |                  2 |                 -2
 11 |                  2 |                 -2
 10 |                  2 |                  2
 17 |     2.999999999998 |     2.999999999998
 14 |                  2 |                  2
 18 | 3.4000000000000004 | 3.4000000000000004
 13 |                  2 |                 -2
 15 |                  2 |                  2
 16 |                  2 |                  2
  9 |                  2 |                  2
  3 |                 -2 |                  2
  1 |                  2 |                  2
  2 |                 -2 |                  2
(18 rows)
```

Note that to be able to follow the documentation examples, everything is based on the original graph.

Returning to the original data:

```
UPDATE edges SET
cost = sign(cost),
reverse_cost = sign(reverse_cost);
UPDATE 18
```

Update costs based on codes

Other datasets, can have a column with values like

- FT vehicle flow on the direction of the geometry
- TF vehicle flow opposite of the direction of the geometry
- B vehicle flow on both directions

Preparing a code column for the example:

```
ALTER TABLE edges ADD COLUMN direction TEXT;
ALTER TABLE
UPDATE edges SET
direction = CASE WHEN (cost>0 AND reverse_cost>0) THEN 'B'
          WHEN (cost>0 AND reverse_cost<0) THEN 'FT'
          WHEN (cost<0 AND reverse_cost>0) THEN 'TF'
          ELSE '' END;
UPDATE 18
```

Adjusting the costs based on the codes:

```
UPDATE edges SET
cost = CASE WHEN (direction = 'B' OR direction = 'FT')
       THEN ST_length(geom) * 2
       ELSE -1 END,
reverse_cost = CASE WHEN (direction = 'B' OR direction = 'TF')
       THEN ST_length(geom) * 2
       ELSE -1 END;
UPDATE 18
```

Which gives the following results:

```
SELECT id, cost, reverse_cost FROM edges;
 id |        cost        |    reverse_cost
----+--------------------+--------------------
  6 |                  2 |                  2
  7 |                  2 |                  2
  4 |                  2 |                  2
  5 |                  2 |                 -1
  8 |                  2 |                  2
 12 |                  2 |                 -1
 11 |                  2 |                 -1
 10 |                  2 |                  2
 17 |     2.999999999998 |     2.999999999998
 14 |                  2 |                  2
 18 | 3.4000000000000004 | 3.4000000000000004
 13 |                  2 |                 -1
 15 |                  2 |                  2
 16 |                  2 |                  2
  9 |                  2 |                  2
  3 |                 -1 |                  2
  1 |                  2 |                  2
  2 |                 -1 |                  2
(18 rows)
```

Returning to the original data:

```
UPDATE edges SET
cost = sign(cost),
reverse_cost = sign(reverse_cost);
UPDATE 18
ALTER TABLE edges DROP COLUMN direction;
ALTER TABLE
```

Check the Routing Topology

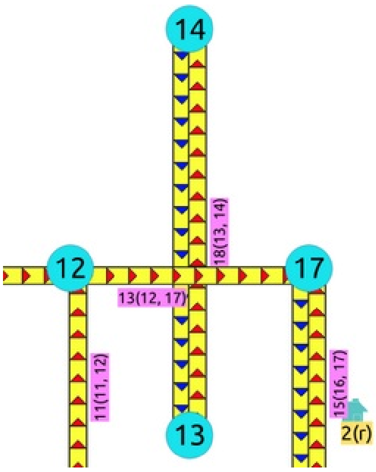
There are lots of possible problems in a graph.

- The data used may not have been designed with routing in mind.
- A graph has some very specific requirements.
- The graph is disconnected.
- There are unwanted intersections.
- The graph is too large and needs to be contracted.
- A sub graph is needed for the application.
- and many other problems that the pgRouting user, that is the application developer might encounter.

Crossing edges

To get the crossing edges:

```
SELECT a.id, b.id
FROM edges AS a, edges AS b
WHERE a.id < b.id AND st_crosses(a.geom, b.geom);
 id | id
----+----
 13 | 18
(1 row)
```



That information is correct, for example, when in terms of vehicles, is it a tunnel or bride crossing over another road.

It might be incorrect, for example:

1. When it is actually an intersection of roads, where vehicles can make turns.
2. When in terms of electrical lines, the electrical line is able to switch roads even on a tunnel or bridge.

When it is incorrect, it needs fixing:

1. For vehicles and pedestrians
   - If the data comes from OSM and was imported to the database using `osm2pgrouting`, the fix needs to be done in the **OSM portal** and the data imported again.
   - In general when the data comes from a supplier that has the data prepared for routing vehicles, and there is a problem, the data is to be fixed from the supplier
2. For very specific applications
   - The data is correct when from the point of view of routing vehicles or pedestrians.
   - The data needs a local fix for the specific application.

Once analyzed one by one the crossings, for the ones that need a local fix, the edges need to be **split**.

```
SELECT ST_AsText((ST_Dump(ST_Split(a.geom, b.geom))).geom)
FROM edges AS a, edges AS b
WHERE a.id = 13 AND b.id = 18
UNION
SELECT ST_AsText((ST_Dump(ST_Split(b.geom, a.geom))).geom)
FROM edges AS a, edges AS b
WHERE a.id = 13 AND b.id = 18;
        st_astext
---------------------------
 LINESTRING(3.5 2.3,3.5 3)
 LINESTRING(3 3,3.5 3)
 LINESTRING(3.5 3,4 3)
 LINESTRING(3.5 3,3.5 4)
(4 rows)
```

The new edges need to be added to the edges table, the rest of the attributes need to be updated in the new edges, the old edges need to be removed and the routing topology needs to be updated.

Adding split edges

For each pair of crossing edges a process similar to this one must be performed.

The columns inserted and the way are calculated are based on the application. For example, if the edges have a trait **name**, then that column is to be copied.

For pgRouting calculations

- **factor** based on the position of the intersection of the edges can be used to adjust the `cost` and `reverse_cost` columns.
- Capacity information, used on the **Flow - Family of functions** functions does not need to change when splitting edges.

```
WITH
first_edge AS (
  SELECT (ST_Dump(ST_Split(a.geom, b.geom))).path[1],
    (ST_Dump(ST_Split(a.geom, b.geom))).geom,
    ST_LineLocatePoint(a.geom,ST_Intersection(a.geom,b.geom)) AS factor
  FROM edges AS a, edges AS b
  WHERE a.id = 13 AND b.id = 18),
first_segments AS (
  SELECT path, first_edge.geom,
    capacity, reverse_capacity,
    CASE WHEN path=1 THEN factor * cost
         ELSE (1 - factor) * cost END AS cost,
    CASE WHEN path=1 THEN factor * reverse_cost
         ELSE (1 - factor) * reverse_cost END AS reverse_cost
  FROM first_edge , edges WHERE id = 13),
second_edge AS (
  SELECT (ST_Dump(ST_Split(b.geom, a.geom))).path[1],
    (ST_Dump(ST_Split(b.geom, a.geom))).geom,
    ST_LineLocatePoint(b.geom,ST_Intersection(a.geom,b.geom)) AS factor
  FROM edges AS a, edges AS b
  WHERE a.id = 13 AND b.id = 18),
second_segments AS (
  SELECT path, second_edge.geom,
    capacity, reverse_capacity,
    CASE WHEN path=1 THEN factor * cost
         ELSE (1 - factor) * cost END AS cost,
    CASE WHEN path=1 THEN factor * reverse_cost
         ELSE (1 - factor) * reverse_cost END AS reverse_cost
  FROM second_edge , edges WHERE id = 18),
all_segments AS (
  SELECT * FROM first_segments
  UNION
  SELECT * FROM second_segments)
INSERT INTO edges
  (capacity, reverse_capacity,
    cost, reverse_cost,
    x1, y1, x2, y2,
    geom)
(SELECT capacity, reverse_capacity, cost, reverse_cost,
  ST_X(ST_StartPoint(geom)), ST_Y(ST_StartPoint(geom)),
  ST_X(ST_EndPoint(geom)), ST_Y(ST_EndPoint(geom)),
  geom
  FROM all_segments);
INSERT 0 4
```

Adding new vertices

After adding all the split edges required by the application, the newly created vertices need to be added to the vertices table.

```
INSERT INTO vertices (in_edges, out_edges, x, y, geom)
(SELECT nv.in_edges, nv.out_edges, nv.x, nv.y, nv.geom
FROM pgr_extractVertices('SELECT id, geom FROM edges') AS nv
LEFT JOIN vertices AS v USING(geom) WHERE v.geom IS NULL);
INSERT 0 1
```

Updating edges topology

```
/* -- set the source information */
UPDATE edges AS e
SET source = v.id
FROM vertices AS v
WHERE source IS NULL AND ST_StartPoint(e.geom) = v.geom;
UPDATE 4
/* -- set the target information */
UPDATE edges AS e
SET target = v.id
FROM vertices AS v
WHERE target IS NULL AND ST_EndPoint(e.geom) = v.geom;
UPDATE 4
```

Removing the surplus edges

Once all significant information needed by the application has been transported to the new edges, then the crossing edges can be deleted.

```
DELETE FROM edges WHERE id IN (13, 18);
DELETE 2
```

There are other options to do this task, like creating a view, or a materialized view.

Updating vertices topology

To keep the graph consistent, the vertices topology needs to be updated

```
UPDATE vertices AS v SET
in_edges = nv.in_edges, out_edges = nv.out_edges
FROM (SELECT * FROM pgr_extractVertices('SELECT id, geom FROM edges')) AS nv
WHERE v.geom = nv.geom;
UPDATE 18
```

Checking for crossing edges

There are no crossing edges on the graph.

```
SELECT a.id, b.id
FROM edges AS a, edges AS b
WHERE a.id < b.id AND st_crosses(a.geom, b.geom);
 id | id
----+----
(0 rows)
```

Disconnected graphs

To get the graph connectivity:

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

In this example, the component \(2\) consists of vertices \(\{2, 4\}\) and both vertices are also part of the dead end result set.

This graph needs to be connected.

Note

With the original graph of this documentation, there would be 3 components as the crossing edge in this graph is a different component.

Prepare storage for connection information

```
ALTER TABLE vertices ADD COLUMN component BIGINT;
ALTER TABLE
ALTER TABLE edges ADD COLUMN component BIGINT;
ALTER TABLE
```

Save the vertices connection information

```
UPDATE vertices SET component = c.component
FROM (SELECT * FROM pgr_connectedComponents(
  'SELECT id, source, target, cost, reverse_cost FROM edges'
)) AS c
WHERE id = node;
UPDATE 18
```

Save the edges connection information

```
UPDATE edges SET component = v.component
FROM (SELECT id, component FROM vertices) AS v
WHERE source = v.id;
UPDATE 20
```

Get the closest vertex

The closest vertex to component \(1\) is vertex \(4\). And the closest edge to vertex\(4\) is edge \(14\).

```
WITH
edges_sql AS (SELECT id, geom FROM edges WHERE component = 1),
point_sql AS (SELECT geom AS point FROM vertices WHERE component = 2),
results AS (
  SELECT
    id::BIGINT AS edge_id,
    ST_LineLocatePoint(geom, point) AS fraction,
    CASE WHEN ST_Intersects(ST_Buffer(geom, 2, 'side=right endcap=flat'), point)
         THEN 'r'
         ELSE 'l' END::CHAR AS side,
    geom <-> point AS distance,
    point,
    ST_MakeLine(point, ST_ClosestPoint(geom, point)) AS new_line
  FROM  edges_sql, point_sql
  WHERE ST_DWithin(geom, point, 2)
  ORDER BY geom <-> point),
prepare_cap AS (
  SELECT row_number() OVER (PARTITION BY point ORDER BY point, distance) AS rn, *
  FROM results),
cap AS (
  SELECT edge_id, fraction, side, distance, point, new_line
  FROM prepare_cap
  WHERE rn <= 1
)
SELECT edge_id, fraction, side, distance, point AS geom, new_line AS edge, id AS closest_vertex
INTO closest
FROM cap JOIN vertices ON (point = geom) ORDER BY distance LIMIT 1;
SELECT 1
```

The `edge` can be used to connect the components, using the `fraction` information about the edge \(14\) to split the connecting edge.

Connecting components

There are three basic ways to connect the components

- From the vertex to the starting point of the edge
- From the vertex to the ending point of the edge
- From the vertex to the closest vertex on the edge
  - This solution requires the edge to be split.

The following query shows the three ways to connect the components:

```
WITH
info AS (
  SELECT
    edge_id, fraction, side, distance, ce.geom, edge, v.id AS closest,
    source, target, capacity, reverse_capacity, e.geom AS e_geom
  FROM closest AS ce
  JOIN vertices AS v USING (geom)
  JOIN edges AS e ON (edge_id = e.id)
  ORDER BY distance LIMIT 1),
three_options AS (
  SELECT
    closest AS source, target, 0 AS cost, 0 AS reverse_cost,
    capacity, reverse_capacity,
    ST_X(geom) AS x1, ST_Y(geom) AS y1,
    ST_X(ST_EndPoint(e_geom)) AS x2, ST_Y(ST_EndPoint(e_geom)) AS y2,
    ST_MakeLine(geom, ST_EndPoint(e_geom)) AS geom
  FROM info

  UNION

  SELECT closest, source, 0, 0, capacity, reverse_capacity,
    ST_X(geom) AS x1, ST_Y(geom) AS y1,
    ST_X(ST_StartPoint(e_geom)) AS x2, ST_Y(ST_StartPoint(e_geom)) AS y2,
    ST_MakeLine(info.geom, ST_StartPoint(e_geom))
  FROM info
  /*
  UNION
  -- This option requires splitting the edge
  SELECT closest, NULL, 0, 0, capacity, reverse_capacity,
    ST_X(geom) AS x1, ST_Y(geom) AS y1,
    ST_X(ST_EndPoint(edge)) AS x2, ST_Y(ST_EndPoint(edge)) AS y2,
    edge
  FROM info */
  )

INSERT INTO edges
  (source, target,
    cost, reverse_cost,
    capacity, reverse_capacity,
    x1, y1, x2, y2,
    geom)
(SELECT
    source, target, cost, reverse_cost, capacity, reverse_capacity,
    x1, y1, x2, y2, geom
  FROM three_options);
INSERT 0 2
```

Checking components

Ignoring the edge that requires further work. The graph is now fully connected as there is only one component.

```
SELECT * FROM pgr_connectedComponents(
  'SELECT id, source, target, cost, reverse_cost FROM edges'
);
 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 |         1 |   14
  15 |         1 |   15
  16 |         1 |   16
  17 |         1 |   17
  18 |         1 |   18
(18 rows)
```

Contraction of a graph

The graph can be reduced in size using **Contraction - Family of functions**

When to contract will depend on the size of the graph, processing times, correctness of the data, on the final application, or any other factor not mentioned.

A fairly good method of finding out if contraction can be useful is because of the number of dead ends and/or the number of linear edges.

A complete method on how to contract and how to use the contracted graph is described on **Contraction - Family of**

functions

Dead ends

To get the dead ends:

```
SELECT id FROM vertices
WHERE array_length(in_edges || out_edges, 1) = 1;
 id
----
  1
  5
  9
 13
 14
  2
  4
(7 rows)
```

That information is correct, for example, when the dead end is on the limit of the imported graph.

Visually node \(4\) looks to be as start/ending of 3 edges, but it is not.

Is that correct?

- Is there such a small curb:
  - That does not allow a vehicle to use that visual intersection?
  - Is the application for pedestrians and therefore the pedestrian can easily walk on the small curb?
  - Is the application for the electricity and the electrical lines than can easily be extended on top of the small curb?
- Is there a big cliff and from eagles view look like the dead end is close to the segment?

When there are many dead ends, to speed up, the **Contraction - Family of functions** functions can be used to divide the problem.

Linear edges

To get the linear edges:

```
SELECT id FROM vertices
WHERE array_length(in_edges || out_edges, 1) = 2;
 id
----
  3
 15
 17
(3 rows)
```

This information is correct, for example, when the application is taking into account speed bumps, stop signals.

When there are many linear edges, to speed up, the **Contraction - Family of functions** functions can be used to divide the problem.

Function’s structure

Once the graph preparation work has been done above, it is time to use a

The general form of a pgRouting function call is:

> pgr_<name>(**Inner queries**, **parameters**, [`Optional parameters`)

Where:

- **Inner queries**: Are compulsory parameters that are `TEXT` strings containing SQL queries.
- **parameters**: Additional compulsory parameters needed by the function.
- `Optional parameters`: Are non compulsory **named** parameters that have a default value when omitted.

The compulsory parameters are positional parameters, the optional parameters are named parameters.

For example, for this **pgr_dijkstra** signature:

> pgr_dijkstra(**Edges SQL**, **start vid**, **end vid** [, `directed`])

- **Edges SQL**:
  - Is the first parameter.
  - It is compulsory.
  - It is an inner query.
  - It has no name, so **Edges SQL** gives an idea of what kind of inner query needs to be used
- **start vid**:
  - Is the second parameter.
  - It is compulsory.
  - It has no name, so **start vid** gives an idea of what the second parameter's value should contain.
- **end vid**
  - Is the third parameter.
  - It is compulsory.
  - It has no name, so **end vid** gives an idea of what the third parameter's value should contain
- `directed`
  - Is the fourth parameter.
  - It is optional.
  - It has a name.

The full description of the parameters are found on the **Parameters** section of each function.

Function's overloads

A function might have different overloads. The most common are called:

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

Depending on the overload the parameters types change.

- **One**: **ANY-INTEGER**
- **Many**: `ARRAY` [**ANY-INTEGER**]

Depending of the function the overloads may vary. But the concept of parameter type change remains the same.

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

Combinations

When routing from:

- From **many** different starting vertices
- to **many** different ending vertices
- Every tuple specifies a pair of a start vertex and an end vertex
- Users can define the combinations as desired.

- Needs a **Combinations SQL**

Inner Queries

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

Edges SQL

General

Edges SQL for

- **Dijkstra - Family of functions**
- **withPoints - Family of functions**
- **Bidirectional Dijkstra - Family of functions**
- **Components - Family of functions**
- **Kruskal - Family of functions**
- **Prim - Family of functions**
- Some uncategorised 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`)
`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`

General without `id`

Edges SQL for

- **All Pairs - Family of Functions**

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`)
`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

General with (X,Y)

Edges SQL for

- A* - Family of functions
- Bidirectional A* - Family of functions

Parameter	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

Flow

Edges SQL for Flow - Family of functions

Edges SQL for

- pgr_pushRelabel
- pgr_edmondsKarp
- pgr_boykovKolmogorov

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

Edges SQL for the following functions of Flow - Family of functions

- pgr_maxFlowMinCost - Experimental

- **pgr_maxFlowMinCost_Cost - Experimental**

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 (target, source) 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 exist
reverse_cost	**ANY-NUMERICAL**	\(-1\)	Weight of the edge (target, source) if it exist

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Combinations SQL

Used on combination signatures

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Restrictions SQL

Column	Type	Description
path	ARRAY [**ANY-INTEGER**]	Sequence of edge identifiers that form a path that is not allowed to be taken. - Empty arrays or NULL arrays are ignored. - Arrays that have a NULL element will raise an exception.
Cost	**ANY-NUMERICAL**	Cost of taking the forbidden path.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Points SQL

Points SQL for

- **withPoints - Family of functions**

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative **value** will be given automatically.
edge_id	**ANY-INTEGER**		Identifier of the "closest" edge to the point.

Parameter	Type	Default	Description
fraction	**ANY-NUMERICAL**		Value in <0,1> that indicates the relative postition from the first end point of the edge.
side	CHAR	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Parameters

The main parameter of the majority of the pgRouting functions is a query that selects the edges of the graph.

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Depending on the family or category of a function it will have additional parameters, some of them are compulsory and some are optional.

The compulsory parameters are nameless and must be given in the required order. The optional parameters are named parameters and will have a default value.

Parameters for the Via functions

- **pgr_dijkstraVia - Proposed**

Parameter	Type	Default	Description
Edges SQL	TEXT		SQL query as described.
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 true if a path is missing stops and returns **EMPTY SET** • When false ignores missing paths returning all paths found
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 identifier 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 identifier is used when no other path is found.

For the TRSP functions

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

Column	Type	Description
Edges SQL	TEXT	SQL query as described.
Restrictions SQL	TEXT	SQL query as described.
Combinations SQL	TEXT	**Combinations SQL** as described below
start vid	**ANY-INTEGER**	Identifier of the departure vertex.
start vids	ARRAY [**ANY-INTEGER**]	Array of identifiers of destination vertices.
end vid	**ANY-INTEGER**	Identifier of the departure vertex.
end vids	ARRAY [**ANY-INTEGER**]	Array of identifiers of destination vertices.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Return columns

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

Return columns for a path

Used on functions that return one path solution

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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_vid` to `node`.

Used on functions the following:

- **pgr_withPoints - Proposed**

Returns set of `(seq, path_seq [, start_pid] [, end_pid], node, edge, cost, agg_cost)`

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	Relative position in the path. - **1** For the first row of the path.
start_pid	BIGINT	Identifier of a starting vertex/point of the path. - When positive is the identifier of the starting vertex. - When negative is the identifier of the starting point. - Returned on **Many to One** and **Many to Many**
end_pid	BIGINT	Identifier of an ending vertex/point of the path. - When positive is the identifier of the ending vertex. - When negative is the identifier of the ending point. - Returned on **One to Many** and **Many to Many**
node	BIGINT	Identifier of the node in the path from`start_pid` to `end_pid`. - When positive is the identifier of the a vertex. - When negative is the identifier of the 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 of the path.

Column	Type	Description
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence. - **0** For the first row of the path.
agg_cost	FLOAT	Aggregate cost from start_vid to node. - **0** For the first row of the path.

Used on functions the following:

- **pgr_dijkstraNear - Proposed**

Returns (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	Relative position in the path. Has value**1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex of the current path.
end_vid	BIGINT	Identifier of the ending vertex of the current path.
node	BIGINT	Identifier of the node in the path fromstart_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go fromnode 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_vid to node.

Multiple paths

Selective for multiple paths.

The columns depend on the function call.

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Path identifier. - Has value **1** for the first of a path fromstart_vid to end_vid.
path_seq	INTEGER	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** - **Combinations**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. - **One to Many** - **Many to Many** - **Combinations**
node	BIGINT	Identifier of the node in the path fromstart_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go fromnode 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_vid to node.

Non selective for multiple paths

Regardless of the call, al the columns are returned.

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

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.

Column	Type	Description
path_id	INTEGER	Path identifier. • Has value **1** for the first of a path from start_vid to end_vid.
path_seq	INTEGER	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex.
end_vid	BIGINT	Identifier of the ending vertex.
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_vid to node.

Return columns for cost functions

Used in the following

- Cost - Category
- Cost Matrix - Category
- All Pairs - Family of Functions

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	Aggregate cost from start_vid to end_vid.

Note

When start_vid or end_vid columns have negative values, the identifier is for a Point.

Return columns for flow functions

Edges SQL for the following

- Flow - Family of functions

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).

Edges SQL for the following functions of Flow - Family of functions

- 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.

Return columns for spanning tree functions

Edges SQL for the following

- **pgr_prim**
- **pgr_kruskal**

Returns SET OF `(edge, cost)`

Column	Type	Description
`edge`	`BIGINT`	Identifier of the edge.
`cost`	`FLOAT`	Cost to traverse the edge.

Performance Tips

- **For the Routing functions**

For the Routing functions

To get faster results bound the queries to an area of interest of routing.

In this example Use an inner query SQL that does not include some edges in the routing function and is within the area of the results.

```
SELECT * FROM pgr_dijkstra($$
  SELECT id, source, target, cost, reverse_cost from edges
  WHERE geom && (SELECT st_buffer(geom, 1) AS myarea
    FROM edges WHERE id = 2)$$,
  1, 2);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

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**

Function Families

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

Function Families

All Pairs - Family of Functions

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

A* - 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**

Reference

- **pgr_version**
- **pgr_full_version**

Topology - Family of Functions

The following functions modify the database directly therefore the user must have special permissions given by the administrators to use them.

- **pgr_createTopology** - create a topology based on the geometry.
- **pgr_createVerticesTable** - 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_bdAstarCost**
- **pgr_dijkstraCost**
- **pgr_bdDijkstraCost**
- **pgr_dijkstraNearCost - Proposed**

Cost Matrix - Category

- **pgr_aStarCostMatrix**
- **pgr_bdAstarCostMatrix**
- **pgr_bdDijkstraCostMatrix**
- **pgr_dijkstraCostMatrix**
- **pgr_bdDijkstraCostMatrix**

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**

BFS - Category

- **pgr_kruskalBFS**
- **pgr_primBFS**

DFS - Category

- **pgr_kruskalDFS**
- **pgr_primDFS**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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

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

Example:
For a directed subgraph with edges $\{1, 2, 3, 4\}$.

```
SELECT * FROM pgr_floydWarshall(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges where id < 5'
) ORDER BY start_vid, end_vid;
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |       7 |        2
         6 |       5 |        1
         6 |       7 |        1
         7 |       5 |        2
         7 |       6 |        1
        10 |       5 |        2
        10 |       6 |        1
        10 |       7 |        2
        15 |       5 |        3
        15 |       6 |        2
        15 |       7 |        3
        15 |      10 |        1
(13 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		**Edges SQL** as described below.

Optional parameters

Column	Type	Default	Description

Column	Type	Default	Description
directed	BOOLEAN	true	- When true the graph is considered *Directed* - When false the graph is considered as *Undirected*.

Inner Queries

Edges SQL

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)
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

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	Aggregate cost from start_vid to end_vid.

See Also

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

Indices and tables

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

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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

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 \(O(V E \log V)\) 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.
- **Recommended, use a bounding box of no more than 3500 edges.**

Signatures

Summary

pgr johnson(**Edges SQL**, [directed])

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

Example:
For a directed subgraph with edges \(\{1, 2, 3, 4\}\).

```
SELECT * FROM pgr_johnson(
  'SELECT source, target, cost FROM edges
  WHERE id < 5'
) ORDER BY start_vid, end_vid;
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |       7 |        2
         6 |       7 |        1
(3 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		**Edges SQL** as described below.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

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	Aggregate cost from start_vid to end_vid.

See Also

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

Indices and tables

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

Introduction

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.**

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		**Edges SQL** as described below.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

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	Aggregate cost from start_vid to end_vid.

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\(\dfrac{E}{V \times (V-1)}\).

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

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
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\(\dfrac{E}{V \times (V-1)}\).
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
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

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
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**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4**

A* - 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.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_aStar

`pgr_aStar` — Shortest path using the A* algorithm.

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** signature:
    - `pgr_aStar` (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **Proposed** signatures:
    - `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**)

Description

The main characteristics are:

- Process works for directed and undirected graphs.
- 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 \(0\)
- 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**, **start vid**, **end vid**, [**options**])
pgr_aStar(**Edges SQL**, **start vid**, **end vids**, [**options**])
pgr_aStar(**Edges SQL**, **start vids**, **end vid**, [**options**])
pgr_aStar(**Edges SQL**, **start vids**, **end vids**, [**options**])
pgr_aStar(**Edges SQL**, **Combinations SQL**, [**options**])
options: [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.

One to One

pgr_aStar(**Edges SQL**, **start vid**, **end vid**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertex \(12\) on a **directed** graph with heuristic \(2\)

```
SELECT * FROM pgr_aStar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, 12,
  directed => true, heuristic => 2);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   10 |    1 |        1
   3 |        3 |    8 |   12 |    1 |        2
   4 |        4 |   12 |   -1 |    0 |        3
(4 rows)
```

One to Many

pgr_aStar(**Edges SQL**, **start vid**, **end vids**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:

From vertex \(6\) to vertices \(\{10, 12\}\) on a **directed** graph with heuristic \(3\) and factor \(3.5\)

```
SELECT * FROM pgr_aStar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, ARRAY[10, 12],
  heuristic => 3, factor := 3.5);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |      10 |    6 |    4 |    1 |        0
   2 |        2 |      10 |    7 |    8 |    1 |        1
   3 |        3 |      10 |   11 |    9 |    1 |        2
   4 |        4 |      10 |   16 |   16 |    1 |        3
   5 |        5 |      10 |   15 |    3 |    1 |        4
   6 |        6 |      10 |   10 |   -1 |    0 |        5
   7 |        1 |      12 |    6 |    4 |    1 |        0
   8 |        2 |      12 |    7 |    8 |    1 |        1
   9 |        3 |      12 |   11 |   11 |    1 |        2
  10 |        4 |      12 |   12 |   -1 |    0 |        3
(10 rows)
```

Many to One

pgr_aStar(**Edges SQL**, **start vids**, **end vid**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:

From vertices \(\{6, 8\}\) to vertex \(10\) on an **undirected** graph with heuristic \(4\)

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

Many to Many

pgr_aStar(**Edges SQL**, **start vids**, **end vids**, [**options**])
options: [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 \(\{6, 8\}\) to vertices \(\{10, 12\}\) on a **directed** graph with factor \(0.5\)

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

Combinations

pgr_aStar(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
Using a combinations table on a **directed** graph with factor \(0.5\).

The combinations table:

```
SELECT * FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_aStar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM combinations',
  factor => 0.5);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    4 |    1 |        1
   5 |        3 |         5 |      10 |    7 |    8 |    1 |        2
   6 |        4 |         5 |      10 |   11 |    9 |    1 |        3
   7 |        5 |         5 |      10 |   16 |   16 |    1 |        4
   8 |        6 |         5 |      10 |   15 |    3 |    1 |        5
   9 |        7 |         5 |      10 |   10 |   -1 |    0 |        6
  10 |        1 |         6 |       5 |    6 |    1 |    1 |        0
  11 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
  12 |        1 |         6 |      15 |    6 |    4 |    1 |        0
  13 |        2 |         6 |      15 |    7 |    8 |    1 |        1
  14 |        3 |         6 |      15 |   11 |    9 |    1 |        2
  15 |        4 |         6 |      15 |   16 |   16 |    1 |        3
  16 |        5 |         6 |      15 |   15 |   -1 |    0 |        4
(16 rows)
```

Parameters

Column	Type	Description

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

aStar optional Parameters

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. • 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: \(h(v) = abs(\Delta x) + abs(\Delta y)\)
factor	FLOAT	1	For units manipulation. \(factor > 0\).
epsilon	FLOAT	1	For less restricted results. \(epsilon >= 1\).

See **heuristics** available and **factor** handling.

Inner Queries

Edges SQL

Parameter	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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.

Parameter	Type	Description
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

Result Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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_vid to node.

Additional Examples

Example 1:
Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:
Making **start vids** the same as **end vids**.

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

Example 3:

Manually assigned vertex combinations.

```
SELECT * FROM pgr_aStar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

See Also

- **A* - Family of functions**
- **Bidirectional A* - 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` - Total cost of the shortest path(s) using the A* algorithm.

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** signature:
    - pgr_aStarCost (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **proposed** function

Description

The pgr_aStarCost function sumarizes of the cost of the shortest path(s) using the A* algorithm.

The main characteristics are:

- Process works for directed and undirected graphs.
- 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 \(0\)
- When \((x,y)\) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's \((x,y)\) coordinates is used.
- Running time: \(O((E + V) * \log V)\)

- It does not return a path.
- Returns the sum of the costs of the shortest path of each pair combination of nodes requested.
- 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)*.
- The returned values are ordered in ascending order:
  - *start_vid* ascending
  - *end_vid* ascending

Signatures

Summary

```
pgr_aStarCost(Edges SQL, start vid, end vid, [options])
pgr_aStarCost(Edges SQL, start vid, end vids, [options])
pgr_aStarCost(Edges SQL, start vids, end vid, [options])
pgr_aStarCost(Edges SQL, start vids, end vids, [options])
pgr_aStarCost(Edges SQL, Combinations SQL, [options])
options: [directed, heuristic, factor, epsilon]

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

One to One

```
pgr_aStarCost(Edges SQL, start vid, end vid, [options])
    options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertex \(12\) on a **directed** graph with heuristic \(2\)

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, 12,
  directed => true, heuristic => 2);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      12 |        3
(1 row)
```

One to Many

pgr_aStarCost(**Edges SQL**, **start vid**, **end vids**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertices \(\{10, 12\}\) on a **directed** graph with heuristic \(3\) and factor \(3.5\)

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, ARRAY[10, 12],
  heuristic => 3, factor => 3.5);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
         6 |      12 |        3
(2 rows)
```

Many to One

pgr_aStarCost(**Edges SQL**, **start vids**, **end vid**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertices \(\{6, 8\}\) to vertex \(10\) on an **undirected** graph with heuristic \(4\)

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[6, 8], 10,
  false, heuristic => 4);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        1
         8 |      10 |        3
(2 rows)
```

Many to Many

pgr_aStarCost(**Edges SQL**, **start vids**, **end vids**, [options])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertices \(\{6, 8\}\) to vertices \(\{10, 12\}\) on a **directed** graph with factor \(0.5\)

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[6, 8], ARRAY[10, 12],
  factor => 0.5);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
         6 |      12 |        3
         8 |      10 |        5
         8 |      12 |        1
(4 rows)
```

Combinations

pgr_aStarCost(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
Using a combinations table on a **directed** graph with factor \(0.5\).

The combinations table:

```
SELECT * FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM combinations',
  factor => 0.5);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        6
         6 |       5 |        1
         6 |      15 |        4
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

aStar optional Parameters

Parameter	Type	Default	Description

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. • 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: \(h(v) = abs(\Delta x) + abs(\Delta y)\)
factor	FLOAT	1	For units manipulation. \(factor > 0\).
epsilon	FLOAT	1	For less restricted results. \(epsilon >= 1\).

See **heuristics** available and **factor** handling.

Inner Queries

Edges SQL

Parameter	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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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	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_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[7, 10, 15, 10, 10, 15], ARRAY[10, 7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 2:

Making **start vids** the same as **end vids**.

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[7, 10, 15], ARRAY[7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 3:

Manually assigned vertex combinations.

```
SELECT * FROM pgr_aStarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |       7 |        1
         6 |      10 |        5
        12 |      10 |        4
(3 rows)
```

See Also

- **A* - Family of functions**
- **Cost - Category**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4**

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

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, start vids, [options])
options: [directed, heuristic, factor, epsilon]

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

Example:
Symmetric cost matrix for vertices \(\{5, 6, 10, 15\}\) on an **undirected** graph using heuristic \(2\)

```
SELECT * FROM pgr_aStarCostMatrix(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edges',
  (SELECT array_agg(id) FROM vertices WHERE id IN (5, 6, 10, 15)),
  directed => false, heuristic => 2);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        2
         5 |      15 |        3
         6 |       5 |        1
         6 |      10 |        1
         6 |      15 |        2
        10 |       5 |        2
        10 |       6 |        1
        10 |      15 |        1
        15 |       5 |        3
        15 |       6 |        2
        15 |      10 |        1
(12 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Optional parameters

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

aStar optional Parameters

Parameter	Type	Default	Description

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. • 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: \(h(v) = abs(\Delta x) + abs(\Delta y)\)
factor	FLOAT	1	For units manipulation. \(factor > 0\).
epsilon	FLOAT	1	For less restricted results. \(epsilon >= 1\).

See **heuristics** available and **factor** handling.

Inner Queries

Edges SQL

Parameter	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

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	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with **pgr_TSP**

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_aStarCostMatrix(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edges',
    (SELECT array_agg(id) FROM vertices WHERE id IN (5, 6, 10, 15)),
    directed=> false, heuristic => 2)
  $$,
  randomize => false
);
NOTICE:  pgr_TSP no longer solving with simulated annaeling
HINT:  Ignoring annaeling parameters
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    5 |    0 |        0
   2 |    6 |    1 |        1
   3 |   10 |    1 |        2
   4 |   15 |    1 |        3
   5 |    5 |    3 |        6
(5 rows)
```

See Also

- **A* - Family of functions**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- **Sample Data**

Indices and tables

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

Description

The main Characteristics are:

- Process works for directed and undirected graphs.
- 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
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.
- Running time: $O((E + V) * \log V)$

aStar optional Parameters

Parameter	Type	Default	Description
`heuristic`	`INTEGER`	5	Heuristic number. Current valid values 0~5. • 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: $h(v) = abs(\Delta x) + abs(\Delta y)$
`factor`	`FLOAT`	`1`	For units manipulation. $factor > 0$.
`epsilon`	`FLOAT`	`1`	For less restricted results. $epsilon >= 1$.

See **heuristics** available and **factor** handling.

Advanced documentation

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

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

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.5 2.6**

Bidirectional A* - Family of functions

The bidirectional A* (pronounced “A Star”) algorithm is based on the A* algorithm.

- **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.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_bdAstar

pgr_bdAstar — Shortest path using the bidirectional A* algorithm.

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** signature:
    - pgr_bdAstar (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **Proposed** signatures:
    - pgr_bdAstar (**One to Many**)
    - pgr_bdAstar (**Many to One**)
    - pgr_bdAstar (**Many to Many**)
  - Signature change on pgr_bdAstar (**One to One**)
    - Old signature no longer supported
- Version 2.0.0
  - **Official** pgr_bdAstar (**One to One**)

Description

The main characteristics are:

- Process works for directed and undirected graphs.
- 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 \(0\)
- 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.

Signatures

Summary

pgr_bdAstar(**Edges SQL**, **start vid**, **end vid**, [**options**])
pgr_bdAstar(**Edges SQL**, **start vid**, **end vids**, [**options**])
pgr_bdAstar(**Edges SQL**, **start vids**, **end vid**, [**options**])
pgr_bdAstar(**Edges SQL**, **start vids**, **end vids**, [**options**])
pgr_bdAstar(**Edges SQL**, **Combinations SQL**, [**options**])
options: [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.

One to One

```
pgr_bdAstar(Edges SQL, start vid, end vid, [options])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertex \(12\) on a **directed** graph with heuristic \(2\)

```
SELECT * FROM pgr_bdAstar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, 12,
  directed => true, heuristic => 2
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   10 |    1 |        1
   3 |        3 |    8 |   12 |    1 |        2
   4 |        4 |   12 |   -1 |    0 |        3
(4 rows)
```

One to Many

```
pgr_bdAstar(Edges SQL, start vid, end vids, [options])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertices \(\{10, 12\}\) on a **directed** graph with heuristic \(3\) and factor \(3.5\)

```
SELECT * FROM pgr_bdAstar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, ARRAY[10, 12],
  heuristic => 3, factor := 3.5
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |      10 |    6 |    4 |    1 |        0
   2 |        2 |      10 |    7 |    8 |    1 |        1
   3 |        3 |      10 |   11 |    9 |    1 |        2
   4 |        4 |      10 |   16 |   16 |    1 |        3
   5 |        5 |      10 |   15 |    3 |    1 |        4
   6 |        6 |      10 |   10 |   -1 |    0 |        5
   7 |        1 |      12 |    6 |    4 |    1 |        0
   8 |        2 |      12 |    7 |    8 |    1 |        1
   9 |        3 |      12 |   11 |   11 |    1 |        2
  10 |        4 |      12 |   12 |   -1 |    0 |        3
(10 rows)
```

Many to One

```
pgr_bdAstar(Edges SQL, start vids, end vid, [options])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertices \(\{6, 8\}\) to vertex \(10\) on an **undirected** graph with heuristic \(4\)

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

Many to Many

pgr_bdAstar(**Edges SQL**, **start vids**, **end vids**, [**options**])
options: [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 \(\{6, 8\}\) to vertices \(\{10, 12\}\) on a **directed** graph with factor \(0.5\)

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

Combinations

pgr_bdAstar(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
Using a combinations table on a **directed** graph with factor \(0.5\).

The combinations table:

```
SELECT * FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_bdAstar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM combinations',
  factor => 0.5
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    4 |    1 |        1
   5 |        3 |         5 |      10 |    7 |    8 |    1 |        2
   6 |        4 |         5 |      10 |   11 |    9 |    1 |        3
   7 |        5 |         5 |      10 |   16 |   16 |    1 |        4
   8 |        6 |         5 |      10 |   15 |    3 |    1 |        5
   9 |        7 |         5 |      10 |   10 |   -1 |    0 |        6
  10 |        1 |         6 |       5 |    6 |    1 |    1 |        0
  11 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
  12 |        1 |         6 |      15 |    6 |    4 |    1 |        0
  13 |        2 |         6 |      15 |    7 |    8 |    1 |        1
  14 |        3 |         6 |      15 |   11 |    9 |    1 |        2
  15 |        4 |         6 |      15 |   16 |   16 |    1 |        3
  16 |        5 |         6 |      15 |   15 |   -1 |    0 |        4
(16 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

aStar optional Parameters

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. • 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: \(h(v) = abs(\Delta x) + abs(\Delta y)\)
factor	FLOAT	1	For units manipulation. \(factor > 0\).
epsilon	FLOAT	1	For less restricted results. \(epsilon >= 1\).

See **heuristics** available and **factor** handling.

Inner Queries

Edges SQL

Parameter	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.

Parameter	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.
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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_vid to node.

Additional Examples

Example 1:
Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:

Making **start vids** the same as **end vids**.

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

Example 3:

Manually assigned vertex combinations.

```
SELECT * FROM pgr_bdAstar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

See Also

- A* - Family of functions
- Bidirectional A* - 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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

pgr_bdAstarCost

`pgr_bdAstarCost` - Total cost of the shortest path(s) using the bidirectional A* algorithm.

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** signature:
    - `pgr_bdAstarCost` (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **proposed** function

Description

The `pgr_bdAstarCost` function sumarizes of the cost of the shortest path(s) using the bidirectional A* algorithm.

The main characteristics are:

- Process works for directed and undirected graphs.
- 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
- When (x,y) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's (x,y) coordinates is used.
- Running time: $O((E + V) * \log V)$

- It does not return a path.
- Returns the sum of the costs of the shortest path of each pair combination of nodes requested.
- 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)*.
- The returned values are ordered in ascending order:
  - *start_vid* ascending
  - *end_vid* ascending

Signatures

Summary

pgr_bdAstarCost(**Edges SQL**, **start vid**, **end vid**, [**options**])
pgr_bdAstarCost(**Edges SQL**, **start vid**, **end vids**, [**options**])
pgr_bdAstarCost(**Edges SQL**, **start vids**, **end vid**, [**options**])
pgr_bdAstarCost(**Edges SQL**, **start vids**, **end vids**, [**options**])
pgr_bdAstarCost(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

One to One

pgr_bdAstarCost(**Edges SQL**, **start vid**, **end vid**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertex \(12\) on a **directed** graph with heuristic \(2\)

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, 12,
  directed => true, heuristic => 2
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      12 |        3
(1 row)
```

One to Many

pgr_bdAstarCost(**Edges SQL**, **start vid**, **end vids**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertex \(6\) to vertices \(\{10, 12\}\) on a **directed** graph with heuristic \(3\) and factor \(3.5\)

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  6, ARRAY[10, 12],
  heuristic => 3, factor := 3.5
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
         6 |      12 |        3
(2 rows)
```

Many to One

pgr_bdAstarCost(**Edges SQL**, **start vids**, **end vid**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertices \(\{6, 8\}\) to vertex \(10\) on an **undirected** graph with heuristic \(4\)

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[6, 8], 10,
  false, heuristic => 4
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        1
         8 |      10 |        3
(2 rows)
```

Many to Many

pgr_bdAstarCost(**Edges SQL**, **start vids**, **end vids**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
From vertices \(\{6, 8\}\) to vertices \(\{10, 12\}\) on a **directed** graph with factor \(0.5\)

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[6, 8], ARRAY[10, 12],
  factor => 0.5
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
         6 |      12 |        3
         8 |      10 |        5
         8 |      12 |        1
(4 rows)
```

Combinations

pgr_bdAstarCost(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, heuristic, factor, epsilon]

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

Example:
Using a combinations table on a **directed** graph with factor \(0.5\).

The combinations table:

```
SELECT * FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM combinations',
  factor => 0.5
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        6
         6 |       5 |        1
         6 |      15 |        4
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

aStar optional Parameters

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. • 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: $h(v) = abs(\Delta x) + abs(\Delta y)$
factor	FLOAT	1	For units manipulation. $factor > 0$.
epsilon	FLOAT	1	For less restricted results. $epsilon >= 1$.

See **heuristics** available and **factor** handling.

Inner Queries

Edges SQL

Parameter	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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.

Parameter	Type	Description
target	ANY-INTEGER	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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	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_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[7, 10, 15, 10, 10, 15], ARRAY[10, 7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 2:
Making **start vids** the same as **end vids**.

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  ARRAY[7, 10, 15], ARRAY[7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 3:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_bdAstarCost(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2
  FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |       7 |        1
         6 |      10 |        5
        12 |      10 |        4
(3 rows)
```

See Also

- **Bidirectional A* - Family of functions**
- **Cost - Category**
- **Sample Data**

Indices and tables

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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

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, start vids, [options])
options: [directed, heuristic, factor, epsilon]

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

Example:
Symmetric cost matrix for vertices \(\{5, 6, 10, 15\}\) on an **undirected** graph using heuristic \(2\)

```
SELECT * FROM pgr_bdAstarCostMatrix(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edges',
  (SELECT array_agg(id) FROM vertices WHERE id IN (5, 6, 10, 15)),
  directed => false, heuristic => 2
);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        2
         5 |      15 |        3
         6 |       5 |        1
         6 |      10 |        1
         6 |      15 |        2
        10 |       5 |        2
        10 |       6 |        1
        10 |      15 |        1
        15 |       5 |        3
        15 |       6 |        2
        15 |      10 |        1
(12 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Optional parameters

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

aStar optional Parameters

Parameter	Type	Default	Description
heuristic	INTEGER	5	Heuristic number. Current valid values 0~5. • 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: \(h(v) = abs(\Delta x) + abs(\Delta y)\)
factor	FLOAT	1	For units manipulation. \(factor > 0\).
epsilon	FLOAT	1	For less restricted results. \(epsilon >= 1\).

See **heuristics** available and **factor** handling.

Inner Queries

Edges SQL

Parameter	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.

Parameter	Type	Default	Description
y2	**ANY-NUMERICAL**		Y coordinate of target vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Result Columns

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	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with **pgr_TSP**

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_bdAstarCostMatrix(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edges',
    (SELECT array_agg(id) FROM vertices WHERE id IN (5, 6, 10, 15)),
    directed=> false, heuristic => 2
  )
  $$,
  randomize => false
);
NOTICE:  pgr_TSP no longer solving with simulated annealing
HINT:  Ignoring annealing parameters
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    5 |    0 |        0
   2 |    6 |    1 |        1
   3 |   10 |    1 |        2
   4 |   15 |    1 |        3
   5 |    5 |    3 |        6
(5 rows)
```

See Also

- **Bidirectional A* - Family of functions**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- **Sample Data**

Indices and tables

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

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 works for directed and undirected graphs.
- 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 \(0\)
- When \((x,y)\) coordinates for the same vertex identifier differ:
  - A random selection of the vertex's \((x,y)\) coordinates is used.
- Running time: \(O((E + V) * \log V)\)
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_astar

See **heuristics** available and **factor** handling.

See Also

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

Indices and tables

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

Previous versions of this page

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

`pgr_bdDijkstra`

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

Boost Graph Inside

Availability:

- Version 3.2.0
  - New **proposed** signature:
    - pgr_bdDijkstra(**Combinations**)
- 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**)

Description

The main characteristics are:

- Process is done only on edges with positive costs.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values (u, v) is ∞
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- 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])
pgr_bdDijkstra(Edges SQL, Combinations SQL , [directed])

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

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 6 to vertex 10 on a **directed** graph

```
SELECT * FROM pgr_bdDijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  6, 10, true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 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 6 to vertices $\{10, 17\}$ on a **directed** graph

```
SELECT * FROM pgr_bdDijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  6, ARRAY[10, 17]);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |      10 |    6 |    4 |    1 |        0
   2 |        2 |      10 |    7 |    8 |    1 |        1
   3 |        3 |      10 |   11 |    9 |    1 |        2
   4 |        4 |      10 |   16 |   16 |    1 |        3
   5 |        5 |      10 |   15 |    3 |    1 |        4
   6 |        6 |      10 |   10 |   -1 |    0 |        5
   7 |        1 |      17 |    6 |    4 |    1 |        0
   8 |        2 |      17 |    7 |    8 |    1 |        1
   9 |        3 |      17 |   11 |   11 |    1 |        2
  10 |        4 |      17 |   12 |   13 |    1 |        3
  11 |        5 |      17 |   17 |   -1 |    0 |        4
(11 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 \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

```
SELECT * FROM pgr_bdDijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  ARRAY[6, 1], 17);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         1 |    1 |    6 |    1 |        0
   2 |        2 |         1 |    3 |    7 |    1 |        1
   3 |        3 |         1 |    7 |    8 |    1 |        2
   4 |        4 |         1 |   11 |   11 |    1 |        3
   5 |        5 |         1 |   12 |   13 |    1 |        4
   6 |        6 |         1 |   17 |   -1 |    0 |        5
   7 |        1 |         6 |    6 |    4 |    1 |        0
   8 |        2 |         6 |    7 |    8 |    1 |        1
   9 |        3 |         6 |   11 |   11 |    1 |        2
  10 |        4 |         6 |   12 |   13 |    1 |        3
  11 |        5 |         6 |   17 |   -1 |    0 |        4
(11 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 \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

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

Combinations

pgr_bdDijkstra(**Edges SQL**, **Combinations SQL**, [directed])

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

Example:
Using a combinations table on an **undirected** graph

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_bdDijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    2 |    1 |        1
   5 |        3 |         5 |      10 |   10 |   -1 |    0 |        2
   6 |        1 |         6 |       5 |    6 |    1 |    1 |        0
   7 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
   8 |        1 |         6 |      15 |    6 |    2 |    1 |        0
   9 |        2 |         6 |      15 |   10 |    3 |    1 |        1
  10 |        3 |         6 |      15 |   15 |   -1 |    0 |        2
(10 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Column	Type	Description
end vids	ARRAY[BIGINT]	Array of identifiers of ending vertices.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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_vid to node.

Additional Examples

Example 1:

Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:

Making **start vids** the same as **end vids**.

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

Example 3:

Manually assigned vertex combinations.

```
SELECT * FROM pgr_bdDijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

See Also

- **Bidirectional Dijkstra - Family of functions**
- **Sample Data**
- **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**

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

pgr_bdDijkstraCost

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

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** signature:
    - `pgr_bdDijkstraCost` (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - New **proposed** function

Description

The `pgr_bdDijkstraCost` function sumarizes of the cost of the shortest path using the bidirectional Dijkstra Algorithm.

- Process is done only on edges with positive costs.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values (u, v) is ∞
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- 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
- It does not return a path.
- Returns the sum of the costs of the shortest path of each pair combination of nodes requested.
- 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 **aggregate cost** of \((u, v)\) is the same as for \((v, u)\).
- Any duplicated value in the start or end vertex identifiers are ignored.
- The returned values are ordered:
    - start_vid ascending
    - end_vid ascending

Signatures

Summary

```
pgr_bdDijkstraCost(Edges SQL, start vid, end vid , [directed])
pgr_bdDijkstraCost(Edges SQL, start vid, end vids, [directed])
pgr_bdDijkstraCost(Edges SQL, start vids, end vid , [directed])
pgr_bdDijkstraCost(Edges SQL, start vids, end vids, [directed])
pgr_bdDijkstraCost(Edges SQL, Combinations SQL, [ directed])

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

One to One

```
pgr_bdDijkstraCost(Edges SQL, start vid, end vid , [directed])

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

Example:
From vertex \(6\) to vertex \(10\) on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 10, true);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
(1 row)
```

One to Many

```
pgr_bdDijkstraCost(Edges SQL, start vid, end vids, [directed])

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

Example:
From vertex \(6\) to vertices \(\{10, 17\}\) on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, ARRAY[10, 17]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
         6 |      17 |        4
(2 rows)
```

Many to One

```
pgr_bdDijkstraCost(Edges SQL, start vids, end vid , [directed])

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

Example:
From vertices \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[6, 1], 17);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |      17 |        5
         6 |      17 |        4
(2 rows)
```

Many to Many

pgr_bdDijkstraCost(**Edges SQL**, **start vids**, **end vids**, [directed])

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

Example:
From vertices \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[6, 1], ARRAY[10, 17],
  directed => false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |      10 |        4
         1 |      17 |        5
         6 |      10 |        1
         6 |      17 |        4
(4 rows)
```

Combinations

pgr_bdDijkstraCost(**Edges SQL**, **Combinations SQL**, [directed])

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

Example:
Using a combinations table on an **undirected** graph

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        2
         6 |       5 |        1
         6 |      15 |        2
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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	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_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[7, 10, 15, 10, 10, 15], ARRAY[10, 7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 2:

Making **start vids** the same as **end vids**.

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[7, 10, 15], ARRAY[7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 3:

Manually assigned vertex combinations.

```
SELECT * FROM pgr_bdDijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |       7 |        1
         6 |      10 |        5
        12 |      10 |        4
(3 rows)
```

See Also

- **Bidirectional Dijkstra - Family of functions**
- **Sample Data**
- **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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

pgr_bdDijkstraCostMatrix

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

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0

- New **proposed** function

Description

Using bidirectional Dijkstra algorithm, calculate and return a cost matrix.

- Process is done only on edges with positive costs.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values (u, v) is ∞
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- 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

The main Characteristics are:

- Can be used as input to **pgr_TSP**.
  - Use directly when the resulting matrix is symmetric and there is no ∞ 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 ∞ 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.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The aggregate 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 aggregate cost in the non included values *(u, v)* is ∞.
- Let be the case the values returned are stored in a table:
  - The unique index would be the pair: `(start_vid, end_vid)`.
- Depending on the function and its parameters, the results can be symmetric.
  - The aggregate 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

Signatures

Summary

```
pgr_bdDijkstraCostMatrix(Edges SQL, start vids, [directed])

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

Example:
Symmetric cost matrix for vertices $\{5, 6, 10, 15\}$ on an **undirected** graph

```
SELECT * FROM pgr_bdDijkstraCostMatrix(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  (SELECT array_agg(id)
    FROM vertices
    WHERE id IN (5, 6, 10, 15)),
  false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        2
         5 |      15 |        3
         6 |       5 |        1
         6 |      10 |        1
         6 |      15 |        2
        10 |       5 |        2
        10 |       6 |        1
        10 |      15 |        1
        15 |       5 |        3
        15 |       6 |        2
        15 |      10 |        1
(12 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

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	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with **pgr_TSP**.

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_bdDijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edges',
    (SELECT array_agg(id)
      FROM vertices
      WHERE id IN (5, 6, 10, 15)),
    false)
  $$);
NOTICE:  pgr_TSP no longer solving with simulated annaeling
HINT:  Ignoring annaeling parameters
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    5 |    0 |        0
   2 |    6 |    1 |        1
   3 |   10 |    1 |        2
   4 |   15 |    1 |        3
   5 |    5 |    3 |        6
(5 rows)
```

See Also

- **Bidirectional Dijkstra - Family of functions**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- **Sample Data**

Indices and tables

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

Synopsis

Based on Dijkstra’s algorithm, the bidirectional search finds a shortest path a starting vertex to an ending vertex.

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.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values\((v, v)\) is \(0\)
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values\((u, v)\) is \(\infty\)
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- 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**

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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.

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_makeConnected - Experimental** - Details of edges to make graph connected.

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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

Description

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

The main characteristics are:

- Works for **undirected** graphs.
- Components are described by vertices
- The returned values are ordered:
  - `component` ascending
  - `node` ascending
- Running time: \(O(V + E)\)

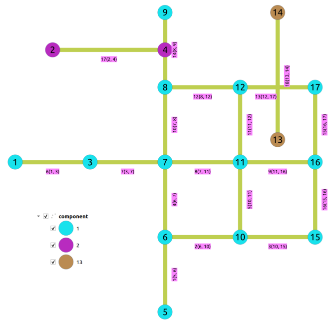
Signatures

```
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 edges'
);
 seq | component | node
-----+-----------+------
   1 |         1 |    1
   2 |         1 |    3
   3 |         1 |    5
   4 |         1 |    6
   5 |         1 |    7
   6 |         1 |    8
   7 |         1 |    9
   8 |         1 |   10
   9 |         1 |   11
  10 |         1 |   12
  11 |         1 |   15
  12 |         1 |   16
  13 |         1 |   17
  14 |         2 |    2
  15 |         2 |    4
  16 |        13 |   13
  17 |        13 |   14
(17 rows)
```



Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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. • Has the value of the minimum node identifier in the component.
node	BIGINT	Identifier of the vertex that belongs to the component.

Additional Examples

Connecting disconnected components

To get the graph connectivity:

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

In this example, the component \(2\) consists of vertices \(\{2, 4\}\) and both vertices are also part of the dead end result set.

This graph needs to be connected.

Note

With the original graph of this documentation, there would be 3 components as the crossing edge in this graph is a different component.

Prepare storage for connection information

```
ALTER TABLE vertices ADD COLUMN component BIGINT;
ALTER TABLE
ALTER TABLE edges ADD COLUMN component BIGINT;
ALTER TABLE
```

Save the vertices connection information

```
UPDATE vertices SET component = c.component
FROM (SELECT * FROM pgr_connectedComponents(
  'SELECT id, source, target, cost, reverse_cost FROM edges'
)) AS c
WHERE id = node;
UPDATE 18
```

Save the edges connection information

```
UPDATE edges SET component = v.component
FROM (SELECT id, component FROM vertices) AS v
WHERE source = v.id;
UPDATE 20
```

Get the closest vertex

The closest vertex to component 1 is vertex 4. And the closest edge to vertex 4 is edge 14.

```
WITH
edges_sql AS (SELECT id, geom FROM edges WHERE component = 1),
point_sql AS (SELECT geom AS point FROM vertices WHERE component = 2),
results AS (
  SELECT
    id::BIGINT AS edge_id,
    ST_LineLocatePoint(geom, point) AS fraction,
    CASE WHEN ST_Intersects(ST_Buffer(geom, 2, 'side=right endcap=flat'), point)
         THEN 'r'
         ELSE 'l' END::CHAR AS side,
    geom <-> point AS distance,
    point,
    ST_MakeLine(point, ST_ClosestPoint(geom, point)) AS new_line
  FROM  edges_sql, point_sql
  WHERE ST_DWithin(geom, point, 2)
  ORDER BY geom <-> point),
prepare_cap AS (
  SELECT row_number() OVER (PARTITION BY point ORDER BY point, distance) AS rn, *
  FROM results),
cap AS (
  SELECT edge_id, fraction, side, distance, point, new_line
  FROM prepare_cap
  WHERE rn <= 1
)
SELECT edge_id, fraction, side, distance, point AS geom, new_line AS edge, id AS closest_vertex
INTO closest
FROM cap JOIN vertices ON (point = geom) ORDER BY distance LIMIT 1;
SELECT 1
```

The `edge` can be used to connect the components, using the `fraction` information about the edge 14 to split the connecting edge.

Connecting components

There are three basic ways to connect the components

- From the vertex to the starting point of the edge
- From the vertex to the ending point of the edge
- From the vertex to the closest vertex on the edge
  - This solution requires the edge to be split.

The following query shows the three ways to connect the components:

```
WITH
info AS (
  SELECT
    edge_id, fraction, side, distance, ce.geom, edge, v.id AS closest,
    source, target, capacity, reverse_capacity, e.geom AS e_geom
  FROM closest AS ce
  JOIN vertices AS v USING (geom)
  JOIN edges AS e ON (edge_id = e.id)
  ORDER BY distance LIMIT 1),
three_options AS (
  SELECT
    closest AS source, target, 0 AS cost, 0 AS reverse_cost,
    capacity, reverse_capacity,
    ST_X(geom) AS x1, ST_Y(geom) AS y1,
    ST_X(ST_EndPoint(e_geom)) AS x2, ST_Y(ST_EndPoint(e_geom)) AS y2,
    ST_MakeLine(geom, ST_EndPoint(e_geom)) AS geom
  FROM info

  UNION

  SELECT closest, source, 0, 0, capacity, reverse_capacity,
    ST_X(geom) AS x1, ST_Y(geom) AS y1,
    ST_X(ST_StartPoint(e_geom)) AS x2, ST_Y(ST_StartPoint(e_geom)) AS y2,
    ST_MakeLine(info.geom, ST_StartPoint(e_geom))
  FROM info
  /*
  UNION
  -- This option requires splitting the edge
  SELECT closest, NULL, 0, 0, capacity, reverse_capacity,
    ST_X(geom) AS x1, ST_Y(geom) AS y1,
    ST_X(ST_EndPoint(edge)) AS x2, ST_Y(ST_EndPoint(edge)) AS y2,
    edge
  FROM info */
  )

INSERT INTO edges
  (source, target,
    cost, reverse_cost,
    capacity, reverse_capacity,
    x1, y1, x2, y2,
    geom)
(SELECT
    source, target, cost, reverse_cost, capacity, reverse_capacity,
    x1, y1, x2, y2, geom
  FROM three_options);
INSERT 0 2
```

Checking components

Ignoring the edge that requires further work. The graph is now fully connected as there is only one component.

```
SELECT * FROM pgr_connectedComponents(
  'SELECT id, source, target, cost, reverse_cost FROM edges'
);
 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 |         1 |   14
  15 |         1 |   15
  16 |         1 |   16
  17 |         1 |   17
  18 |         1 |   18
(18 rows)
```

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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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

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:

- Works for **directed** graphs.
- Components are described by vertices identifiers.
- The returned values are ordered:
  - component ascending
  - node ascending
- Running time: \(O(V + E)\)

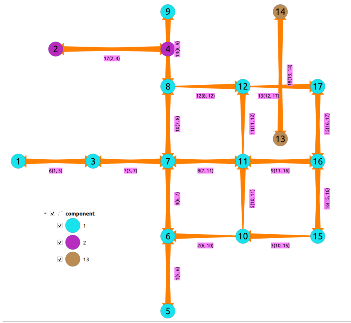
Signatures

```
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 edges'
);
 seq | component | node
-----+-----------+------
   1 |         1 |    1
   2 |         1 |    3
   3 |         1 |    5
   4 |         1 |    6
   5 |         1 |    7
   6 |         1 |    8
   7 |         1 |    9
   8 |         1 |   10
   9 |         1 |   11
  10 |         1 |   12
  11 |         1 |   15
  12 |         1 |   16
  13 |         1 |   17
  14 |         2 |    2
  15 |         2 |    4
  16 |        13 |   13
  17 |        13 |   14
(17 rows)
```



Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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. • Has the value of the minimum node identifier in the component.
node	BIGINT	Identifier of the vertex that belongs to thecomponent.

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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

pgr_biconnectedComponents — Biconnected components of an undirected 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

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:

- Works for **undirected** graphs.
- Components are described by edges.
- The returned values are ordered:
  - component ascending.
  - edge ascending.
- Running time: \(O(V + E)\)

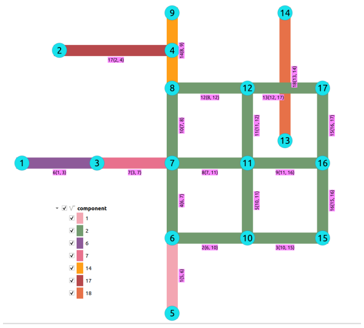
Signatures

```
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 edges'
);
 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	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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. • Has the value of the minimum edge identifier in the component.
edge	BIGINT	Identifier of the edge that belongs to the component.

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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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

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:

- Works for **undirected** graphs.
- The returned values are ordered:
  - node ascending
- Running time: \(O(V + E)\)

Signatures

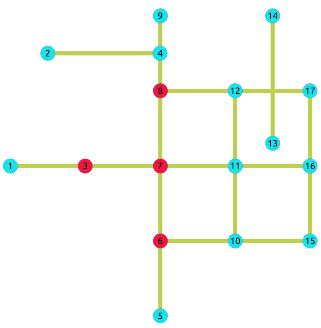
```
pgr_articulationPoints(Edges SQL)

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 edges'
);
 node
------
    3
    6
    7
    8
(4 rows)
```

Nodes in red are the articulation points.



Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5**

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

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:

- Works for **undirected** graphs.
- The returned values are ordered:
  - edge ascending
- Running time: \(O(E * (V + E))\)

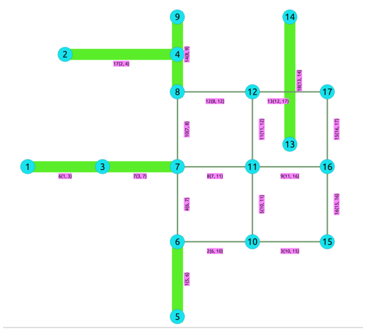
Signatures

```
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 edges'
);
 edge
------
    1
    6
    7
   14
   17
   18
(6 rows)
```



Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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**

- **Supported versions: Latest (3.3) 3.2**

pgr_makeConnected - Experimental

pgr_makeConnected — Set of edges that will connect the 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 3.2.0
  - New **experimental** function

Description

Adds the minimum number of edges needed to make the input graph connected. The algorithm first identifies all of the connected components in the graph, then adds edges to connect those components together in a path. For example, if a graph contains three connected components A, B, and C, make_connected will add two edges. The two edges added might consist of one connecting a vertex in A with a vertex in B and one connecting a vertex in B with a vertex in C.

The main characteristics are:

- Works for **undirected** graphs.
- It will give a minimum list of all edges which are needed in the graph to make connect it.
- The algorithm does not considers traversal costs in the calculations.
- The algorithm does not considers geometric topology in the calculations.
- Running time: \(O(V + E)\)

Signatures

```
pgr_makeConnected(Edges SQL)

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

Example:
Query done on **Sample Data** network gives the list of edges that are needed to connect the graph.

```
SELECT * FROM pgr_makeConnected(
  'SELECT id, source, target, cost, reverse_cost FROM edges'
);
 seq | start_vid | end_vid
-----+-----------+---------
   1 |         5 |       2
   2 |         4 |      13
(2 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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_vid, end_vid)

Column	Type	Description
seq	BIGINT	Sequential value starting from **1**.
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.

See Also

- **https://www.boost.org/libs/graph/doc/make_connected.html**
- The queries use the **Sample Data** network.

Indices and tables

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

See Also

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

Contraction - Family of functions

- **pgr_contraction**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

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

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 is `v`
  - column `id` descending when type is `e`

Signatures

Summary

The pgr_contraction function has the following signature:

pgr_contraction(**Edges SQL**, **contraction order**, [**options**])

options: [max_cycles, forbidden_vertices, directed]
RETURNS SET OF (type, id, contracted_vertices, source, target, cost)

Example:
Making a dead end and linear contraction in that order on an undirected graph.

```
SELECT * FROM pgr_contraction(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[1, 2], directed => false);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 v    |  4 | {2}                 |     -1 |     -1 |   -1
 v    |  7 | {1,3}               |     -1 |     -1 |   -1
 v    | 14 | {13}                |     -1 |     -1 |   -1
 e    | -1 | {5,6}               |      7 |     10 |    2
 e    | -2 | {8,9}               |      7 |     12 |    2
 e    | -3 | {17}                |     12 |     16 |    2
 e    | -4 | {15}                |     10 |     16 |    2
(7 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
contraction Order	ARRAY[**ANY-INTEGER**]	Ordered contraction operations. • 1 = Dead end contraction • 2 = Linear contraction

Optional Parameters

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

Contraction optional parameters

Column	Type	Default	Description
forbidden_vertices	ARRAY[**ANY-INTEGER**]	**Empty**	Identifiers of vertices forbidden for contraction.
max_cycles	INTEGER	\(1\)	Number of times the contraction operations on contraction_order will be performed.

Inner Queries

Edges SQL

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)
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 (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.
		  ◦ Column id has a positive value
		• e when the row is an edge.
		  ◦ Column id has a negative value

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.
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 type, id, contracted_vertices FROM pgr_contraction(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[1]);
 type | id | contracted_vertices
------+----+---------------------
 v    |  4 | {2}
 v    |  6 | {5}
 v    |  7 | {1,3}
 v    |  8 | {9}
 v    | 14 | {13}
(5 rows)
```

Example:
Only linear contraction

```
SELECT * FROM pgr_contraction(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[2]);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 e    | -1 | {3}                 |      1 |      7 |    2
 e    | -2 | {3}                 |      7 |      1 |    2
(2 rows)
```

See Also

- **Contraction - Family of functions**

Indices and tables

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

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

Contraction of the leaf nodes of the graph.

Dead end

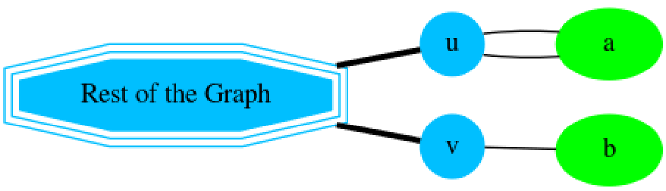
A node is considered a **dead end** node when

- On undirected graphs:
  - The number of adjacent vertices is 1.
- On directed graphs:
  - 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.

Dead end vertex on undirected graph

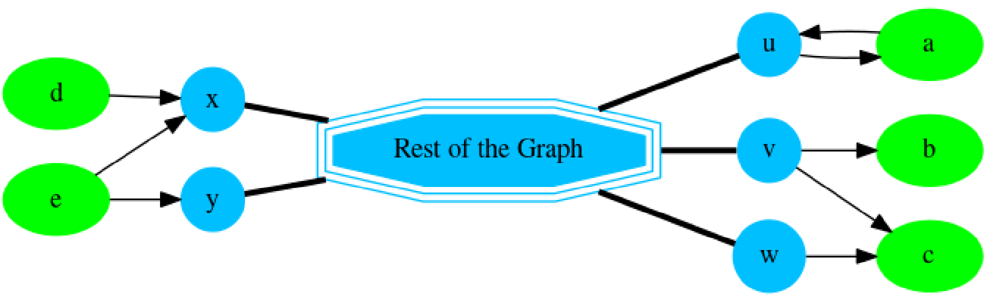
- The green nodes are **dead end** nodes
- The blue nodes have an unlimited number of edges.



Node	Adjecent nodes	Number of adjacent nodes
a	$\{u\}$	1
b	$\{v\}$	1

Dead end vertex on directed graph

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



Node	Adjecent nodes	Number of adjacent nodes	Number of incoming edges	Number of outgoing edges
a	$\{u\}$	1		
b	$\{v\}$	1		
c	$\{v, w\}$	2	2	0
d	$\{x\}$	1		
e	$\{x, y\}$	2	0	2

From above, nodes $\{a, b, d\}$ are dead ends because the number of adjacent vertices is 1. No further checks are needed for those nodes.

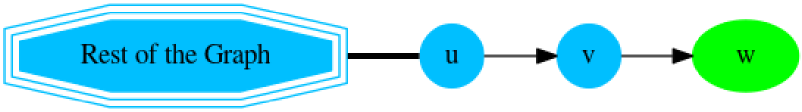
On the following table, nodes $\{c, e\}$ because the even that the number of adjacent vertices is not 1 for

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

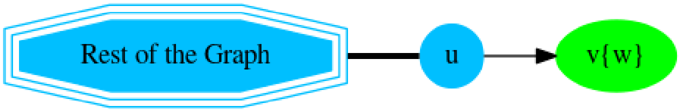
- There are no incoming edges and has at least one outgoing edge.

Operation: Dead End Contraction

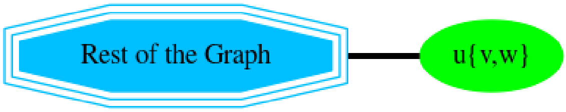
The dead end contraction will stop until there are no more dead end nodes. For example from the following graph where \(w\) is the **dead end** node:



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.

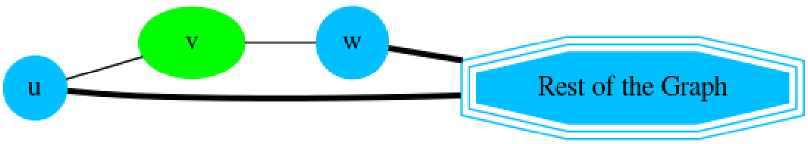
In case of a directed graph, a node is considered a *linear* node when

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

Linear vertex on undirected graph

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

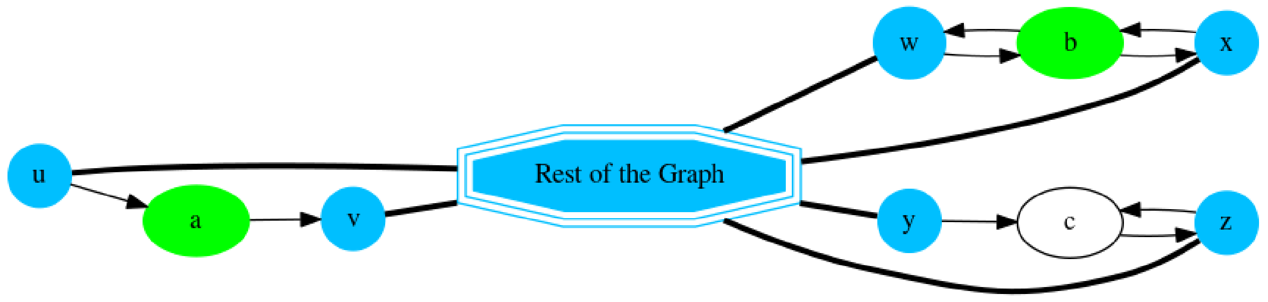
Undirected



Node	Adjecent nodes	Number of adjacent nodes
\(v\)	\(\{u, w\}\)	2

Linear vertex on directed graph

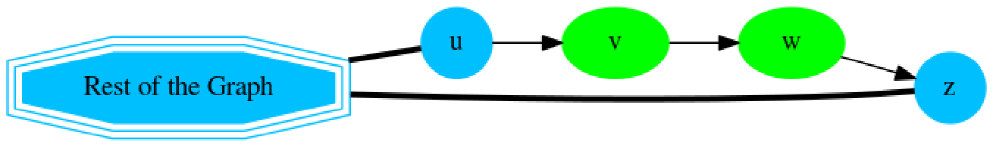
- The green nodes are **linear** nodes
- The blue nodes have an unlimited number of incoming and outgoing edges.
- The white node is not linear because the linearity is not symetrical.
    - It is possible to go \(y \rightarrow c \rightarrow z\)
    - It’s not possible to go \(z \rightarrow c \rightarrow y\)



Node	Adjecent nodes	Number of adjacent nodes	Is symmetrical?
\(a\)	\(\{u, v\}\)	2	yes
\(b\)	\(\{w, x\}\)	2	yes
\(c\)	\(\{y, z\}\)	2	no

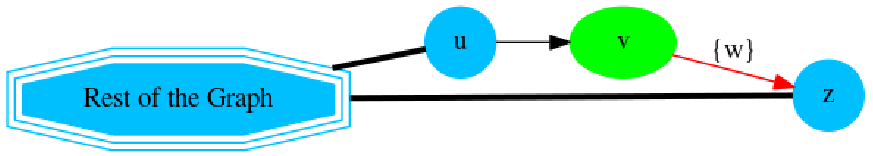
Operation: Linear Contraction

The linear contraction will stop when there are no more linear nodes. For example from the following graph where \(v\) and \(w\) are **linear** nodes:



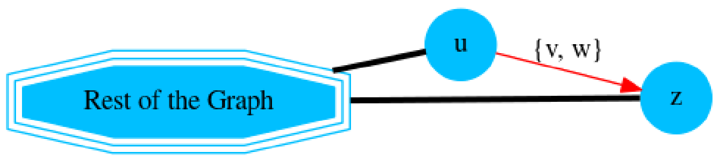
Contracting \(w\),

- 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.

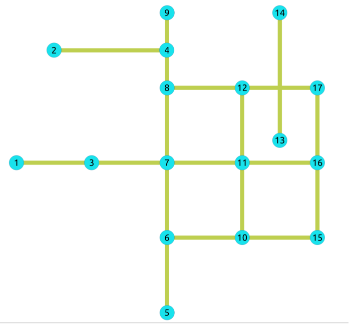
Construction of the graph in the database

Original Data

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

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

The original graph:



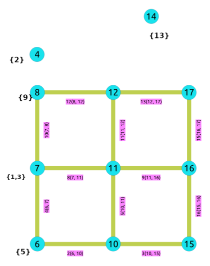
Contraction results

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

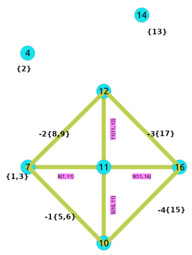
Observe that vertices, for example,\(6\) do not appear in the results because it was not affected by the contraction algorithm.

```
SELECT * FROM pgr_contraction(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  array[1, 2], directed => false);
 type | id | contracted_vertices | source | target | cost
------+----+---------------------+--------+--------+------
 v    |  4 | {2}                 |     -1 |     -1 |   -1
 v    |  7 | {1,3}               |     -1 |     -1 |   -1
 v    | 14 | {13}                |     -1 |     -1 |   -1
 e    | -1 | {5,6}               |      7 |     10 |    2
 e    | -2 | {8,9}               |      7 |     12 |    2
 e    | -3 | {17}                |     12 |     16 |    2
 e    | -4 | {15}                |     10 |     16 |    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

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.
`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 vertices ADD is_contracted BOOLEAN DEFAULT false;
ALTER TABLE
ALTER TABLE vertices ADD contracted_vertices BIGINT[];
ALTER TABLE
ALTER TABLE edges ADD is_new BOOLEAN DEFAULT false;
ALTER TABLE
ALTER TABLE edges 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 edges',
  array[1, 2], directed => false);
SELECT 7
```

The vertex table update

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

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

Fill `contracted_vertices` with the information from the results tha belong to the vertices.

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

The modified vertices table:

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

The edge table update

Insert the new edges generated by pgr_contraction.

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

The contracted graph

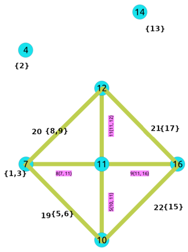
Vertices that belong to the contracted graph.

```
SELECT id
FROM vertices
WHERE is_contracted = false
ORDER BY id;
 id
----
  4
  7
 10
 11
 12
 14
 16
(7 rows)
```

Edges that belong to the contracted graph.

```
WITH
vertices_in_graph AS (
  SELECT id
  FROM vertices
  WHERE is_contracted = false
)
SELECT id, source, target, cost, reverse_cost, contracted_vertices
FROM edges
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 |     10 |     11 |    1 |           -1 |
  8 |      7 |     11 |    1 |            1 |
  9 |     11 |     16 |    1 |            1 |
 11 |     11 |     12 |    1 |           -1 |
 19 |      7 |     10 |    2 |           -1 | {5,6}
 20 |      7 |     12 |    2 |           -1 | {8,9}
 21 |     12 |     16 |    2 |           -1 | {17}
 22 |     10 |     16 |    2 |           -1 | {15}
(8 rows)
```

Contracted graph



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   $$
10   WITH
11   vertices_in_graph AS (
12     SELECT id
13     FROM vertices
14     WHERE is_contracted = false
15   )
16   SELECT id, source, target, cost, reverse_cost
17   FROM edges
18   WHERE source IN (SELECT * FROM vertices_in_graph)
19   AND target IN (SELECT * FROM vertices_in_graph)
20   $$,
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(10, 12);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   10 |    5 |    1 |        0
   2 |        2 |   11 |   11 |    1 |        1
   3 |        3 |   12 |   -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(15, 12);
 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(15, 1);
 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   $$
10   WITH
11   edges_to_expand AS (
12     SELECT id
13     FROM edges
14     WHERE ARRAY[$$ || departure || $$]::BIGINT[] <@ contracted_vertices
15     OR ARRAY[$$ || destination || $$]::BIGINT[] <@ contracted_vertices
16   ),
17
18   vertices_in_graph AS (
19     SELECT id
20     FROM vertices
21     WHERE is_contracted = false
22
23     UNION
24
25     SELECT unnest(contracted_vertices)
26     FROM edges
27     WHERE id IN (SELECT id FROM edges_to_expand)
28   )
29
30   SELECT id, source, target, cost, reverse_cost
31   FROM edges
32   WHERE source IN (SELECT * FROM vertices_in_graph)
33   AND target IN (SELECT * FROM vertices_in_graph)
34   $$,
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(10, 12);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   10 |    5 |    1 |        0
   2 |        2 |   11 |   11 |    1 |        1
   3 |        3 |   12 |   -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 node\(4\).

```
SELECT * FROM my_dijkstra(15, 12);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   15 |   16 |    1 |        0
   2 |        2 |   16 |   21 |    2 |        1
   3 |        3 |   12 |   -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 node\(7\).

```
SELECT * FROM my_dijkstra(15, 1);
 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   $$
10   WITH
11   edges_to_expand AS (
12     SELECT id
13     FROM edges
14     WHERE ARRAY[$$ || departure || $$]::BIGINT[] <@ contracted_vertices
15     OR ARRAY[$$ || destination || $$]::BIGINT[] <@ contracted_vertices
16   ),
17
18   vertices_to_expand AS (
19     SELECT id
20     FROM vertices
21     WHERE ARRAY[$$ || departure || $$]::BIGINT[] <@ contracted_vertices
22     OR ARRAY[$$ || destination || $$]::BIGINT[] <@ contracted_vertices
23   ),
24
25   vertices_in_graph AS (
26     SELECT id
27     FROM vertices
28     WHERE is_contracted = false
29
30     UNION
31
32     SELECT unnest(contracted_vertices)
33     FROM edges
34     WHERE id IN (SELECT id FROM edges_to_expand)
35
36     UNION
37
38     SELECT unnest(contracted_vertices)
39     FROM vertices
40     WHERE id IN (SELECT id FROM vertices_to_expand)
41   )
42
43   SELECT id, source, target, cost, reverse_cost
44   FROM edges
45   WHERE source IN (SELECT * FROM vertices_in_graph)
46   AND target IN (SELECT * FROM vertices_in_graph)
47   $$,
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(10, 12);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   10 |    5 |    1 |        0
   2 |        2 |   11 |   11 |    1 |        1
   3 |        3 |   12 |   -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(15, 12);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   15 |   16 |    1 |        0
   2 |        2 |   16 |   21 |    2 |        1
   3 |        3 |   12 |   -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 node\(7\).

```
SELECT * FROM my_dijkstra(15, 1);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   15 |    3 |    1 |        0
   2 |        2 |   10 |   19 |    2 |        1
   3 |        3 |    7 |    7 |    1 |        3
   4 |        4 |    3 |    6 |    1 |        4
   5 |        5 |    1 |   -1 |    0 |        5
(5 rows)
```

See Also

- **pgr_contraction**
- **Sample Data**
- **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**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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_dijkstraNear - Proposed** - Get the route to the nearest vertex.
- **pgr_dijkstraNearCost - Proposed** - Get the cost to the nearest vertex.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_dijkstra

pgr_dijkstra — Shortest path(s) using Dijkstra algorithm.

Boost Graph Inside

Availability

- Version 3.1.0
  - New **Proposed** functions:
    - pgr_dijkstra (**Combinations**)
- 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**)

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 to an ending vertex. This implementation can be used with a directed graph and an undirected graph.

- Process is done only on edges with positive costs.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values\((v, v)\) is \(0\)
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values\((u, v)\) is \(\infty\)
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- 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])
pgr_dijkstra(Edges SQL, Combinations SQL , [directed])

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

One to One

```
pgr_dijkstra(Edges SQL, start vid, end vid , [directed])

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

Example:
From vertex \(6\) to vertex \(10\) on a **directed** graph

```
SELECT * FROM pgr_Dijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  6, 10, true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 rows)
```

One to Many

pgr_dijkstra(**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 \(6\) to vertices \(\{10, 17\}\) on a **directed**

```
SELECT * FROM pgr_Dijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  6, ARRAY[10, 17]);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |      10 |    6 |    4 |    1 |        0
   2 |        2 |      10 |    7 |    8 |    1 |        1
   3 |        3 |      10 |   11 |    9 |    1 |        2
   4 |        4 |      10 |   16 |   16 |    1 |        3
   5 |        5 |      10 |   15 |    3 |    1 |        4
   6 |        6 |      10 |   10 |   -1 |    0 |        5
   7 |        1 |      17 |    6 |    4 |    1 |        0
   8 |        2 |      17 |    7 |    8 |    1 |        1
   9 |        3 |      17 |   11 |    9 |    1 |        2
  10 |        4 |      17 |   16 |   15 |    1 |        3
  11 |        5 |      17 |   17 |   -1 |    0 |        4
(11 rows)
```

Many to One

pgr_dijkstra(**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 \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

```
SELECT * FROM pgr_Dijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  ARRAY[6, 1], 17);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         1 |    1 |    6 |    1 |        0
   2 |        2 |         1 |    3 |    7 |    1 |        1
   3 |        3 |         1 |    7 |    8 |    1 |        2
   4 |        4 |         1 |   11 |   11 |    1 |        3
   5 |        5 |         1 |   12 |   13 |    1 |        4
   6 |        6 |         1 |   17 |   -1 |    0 |        5
   7 |        1 |         6 |    6 |    4 |    1 |        0
   8 |        2 |         6 |    7 |    8 |    1 |        1
   9 |        3 |         6 |   11 |   11 |    1 |        2
  10 |        4 |         6 |   12 |   13 |    1 |        3
  11 |        5 |         6 |   17 |   -1 |    0 |        4
(11 rows)
```

Many to Many

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
```

Example:
From vertices \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

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

Combinations

```
pgr_dijkstra(Edges SQL, Combinations SQL , [directed])

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

Example:
Using a combinations table on an **undirected** graph

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_Dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    2 |    1 |        1
   5 |        3 |         5 |      10 |   10 |   -1 |    0 |        2
   6 |        1 |         6 |       5 |    6 |    1 |    1 |        0
   7 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
   8 |        1 |         6 |      15 |    6 |    2 |    1 |        0
   9 |        2 |         6 |      15 |   10 |    3 |    1 |        1
  10 |        3 |         6 |      15 |   15 |   -1 |    0 |        2
(10 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below

Column	Type	Description
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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.

Column	Type	Description
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_vid to node.

Additional Examples

Example:
Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:
Making **start_vids** the same as **end_vids**

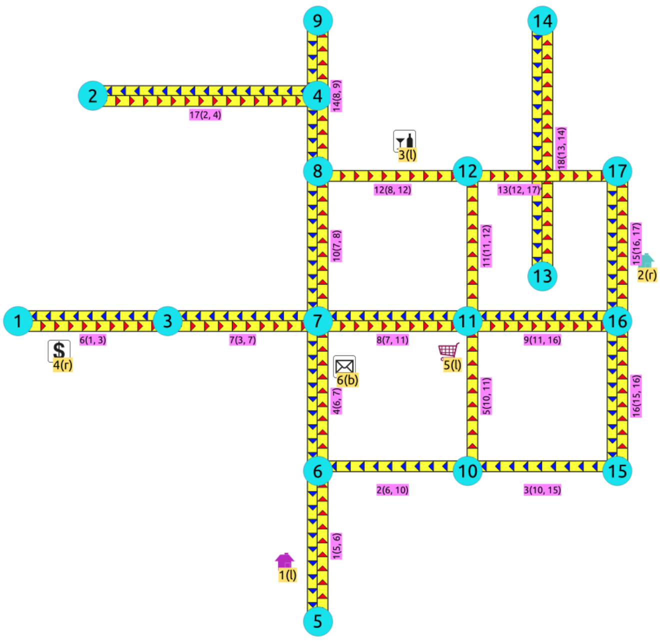
```
SELECT * FROM pgr_Dijkstra(
  'select id, source, target, cost, reverse_cost from edges',
  ARRAY[7, 10, 15], ARRAY[7, 10, 15]);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         7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

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_Dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

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

For directed graphs with `cost` and `reverse_cost` columns



Directed graph with cost and reverse cost columns

1) Path from \(6\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 10
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 rows)
```

2) Path from \(6\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 7
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   -1 |    0 |        1
(2 rows)
```

3) Path from \(12\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  12, 10
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   12 |   13 |    1 |        0
   2 |        2 |   17 |   15 |    1 |        1
   3 |        3 |   16 |   16 |    1 |        2
   4 |        4 |   15 |    3 |    1 |        3
   5 |        5 |   10 |   -1 |    0 |        4
(5 rows)
```

4) Path from \(12\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  12, 7
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   12 |   13 |    1 |        0
   2 |        2 |   17 |   15 |    1 |        1
   3 |        3 |   16 |    9 |    1 |        2
   4 |        4 |   11 |    8 |    1 |        3
   5 |        5 |    7 |   -1 |    0 |        4
(5 rows)
```

5) Using One to Many to get the solution of examples 1 and 2

Paths \(\{6\}\rightarrow\{10, 7\}\)

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

6) Using Many to One to get the solution of examples 2 and 4

Paths \(\{6, 12\}\rightarrow\{7\}\)

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

7) Using Many to Many to get the solution of examples 1 to 4

Paths \(\{6, 12\}\rightarrow\{10, 7\}\)

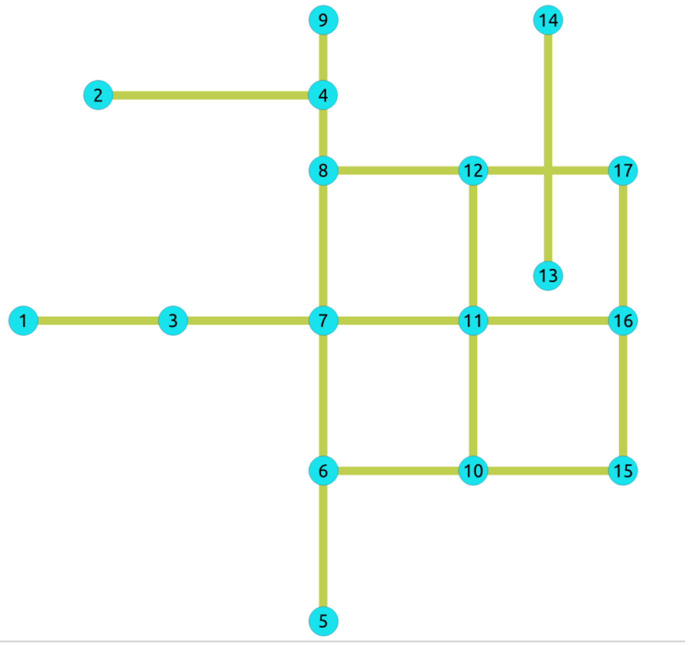
```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[6, 12], ARRAY[10,7]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |     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

8) Using Combinations to get the solution of examples 1 to 3

Paths $\{6\}\rightarrow\{10, 7\}\cup\{12\}\rightarrow\{10\}$

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

For undirected graphs with `cost` and `reverse_cost` columns



Undirected graph with cost and reverse cost columns

9) Path from 6 to 10

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 10,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    2 |    1 |        0
   2 |        2 |   10 |   -1 |    0 |        1
(2 rows)
```

10) Path from 6 to 7

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 7,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   -1 |    0 |        1
(2 rows)
```

11) Path from \(12\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  12, 10,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   12 |   11 |    1 |        0
   2 |        2 |   11 |    5 |    1 |        1
   3 |        3 |   10 |   -1 |    0 |        2
(3 rows)
```

12) Path from \(12\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  12, 7,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   12 |   12 |    1 |        0
   2 |        2 |    8 |   10 |    1 |        1
   3 |        3 |    7 |   -1 |    0 |        2
(3 rows)
```

13) Using One to Many to get the solution of examples 9 and 10

Paths \(\{6\}\rightarrow\{10, 7\}\)

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

14) Using Many to One to get the solution of examples 10 and 12

Paths \(\{6, 12\}\rightarrow\{7\}\)

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

15) Using Many to Many to get the solution of examples 9 to 12

Paths \(\{6, 12\}\rightarrow\{10, 7\}\)

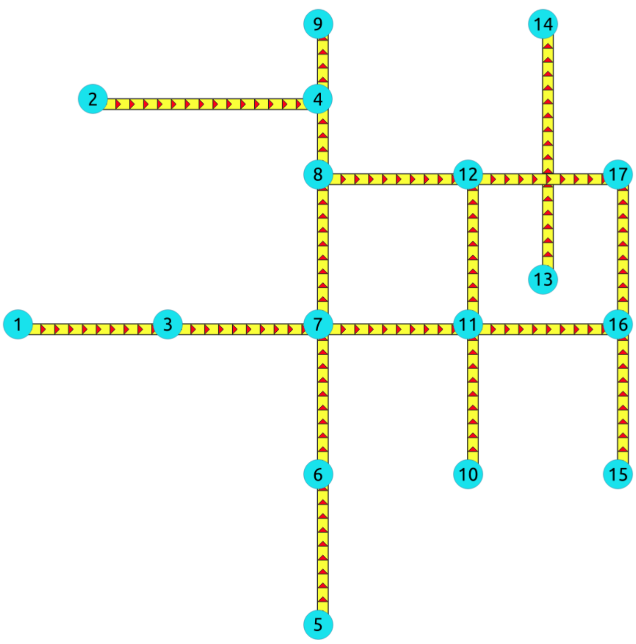
```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[6, 12], ARRAY[10,7],
  false
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |        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

16) Using Combinations to get the solution of examples 9 to 11

Paths \(\{6\}\rightarrow\{10, 7\}\cup\{12\}\rightarrow\{10\}\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)',
  false
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    2 |    1 |        0
   4 |        2 |         6 |      10 |   10 |   -1 |    0 |        1
   5 |        1 |        12 |      10 |   12 |   11 |    1 |        0
   6 |        2 |        12 |      10 |   11 |    5 |    1 |        1
   7 |        3 |        12 |      10 |   10 |   -1 |    0 |        2
(7 rows)
```

For directed graphs only with `cost` column



Directed graph only with cost column

17) Path from \(6\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  6, 10
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

18) Path from \(6\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  6, 7
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   -1 |    0 |        1
(2 rows)
```

19) Path from \(12\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  12, 10
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

20) Path from \(12\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  12, 7
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

21) Using One to Many to get the solution of examples 17 and 18

Paths \(\{6\}\rightarrow\{10, 7\}\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  6, ARRAY[10,7]
);
 seq | path_seq | end_vid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |       7 |    6 |    4 |    1 |        0
   2 |        2 |       7 |    7 |   -1 |    0 |        1
(2 rows)
```

22) Using Many to One to get the solution of examples 18 and 20

Paths \(\{6, 12\}\rightarrow\{7\}\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  ARRAY[6,12], 7
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         6 |    6 |    4 |    1 |        0
   2 |        2 |         6 |    7 |   -1 |    0 |        1
(2 rows)
```

23) Using Many to Many to get the solution of examples 17 to 20

Paths \(\{6, 12\}\rightarrow\{10, 7\}\)

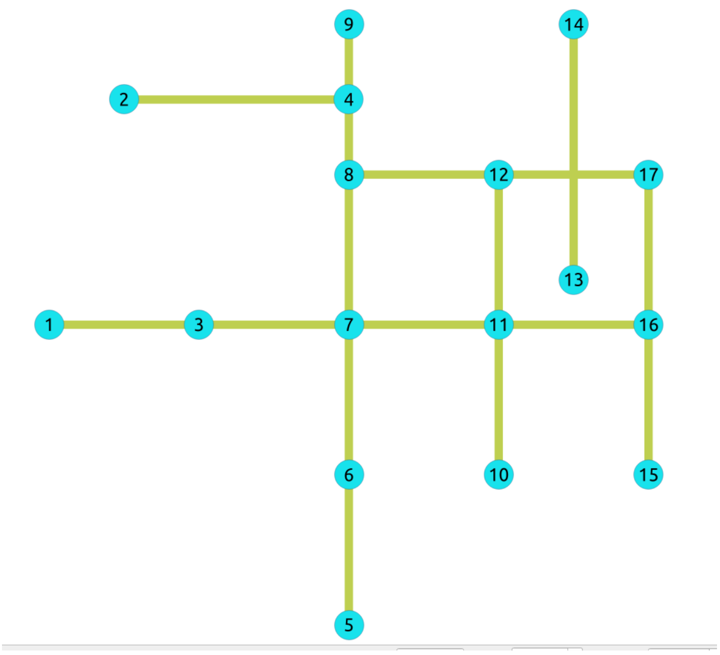
```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  ARRAY[6, 12], ARRAY[10,7]
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
(2 rows)
```

24) Using Combinations to get the solution of examples 17 to 19

Paths \(\{6\}\rightarrow\{10, 7\}\cup\{12\}\rightarrow\{10\}\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)'
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
(2 rows)
```

For undirected graphs only with `cost` column



Undirected graph only with cost column

25) Path from \(6\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  6, 10,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    5 |    1 |        2
   4 |        4 |   10 |   -1 |    0 |        3
(4 rows)
```

26) Path from \(6\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  6, 7,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   -1 |    0 |        1
(2 rows)
```

27) Path from \(12\) to \(10\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  12, 10,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   12 |   11 |    1 |        0
   2 |        2 |   11 |    5 |    1 |        1
   3 |        3 |   10 |   -1 |    0 |        2
(3 rows)
```

28) Path from \(12\) to \(7\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  12, 7,
  false
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   12 |   12 |    1 |        0
   2 |        2 |    8 |   10 |    1 |        1
   3 |        3 |    7 |   -1 |    0 |        2
(3 rows)
```

29) Using One to Many to get the solution of examples 25 and 26

Paths \(\{6\}\rightarrow\{10, 7\}\)

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

30) Using Many to One to get the solution of examples 26 and 28

Paths \(\{6, 12\}\rightarrow\{7\}\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  ARRAY[6,12], 7,
  false
);
 seq | path_seq | start_vid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |         6 |    6 |    4 |    1 |        0
   2 |        2 |         6 |    7 |   -1 |    0 |        1
   3 |        1 |        12 |   12 |   12 |    1 |        0
   4 |        2 |        12 |    8 |   10 |    1 |        1
   5 |        3 |        12 |    7 |   -1 |    0 |        2
(5 rows)
```

31) Using Many to Many to get the solution of examples 25 to 28

Paths \(\{6, 12\}\rightarrow\{10, 7\}\)

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

32) Using Combinations to get the solution of examples 25 to 27

Paths \(\{6\}\rightarrow\{10, 7\}\cup\{12\}\rightarrow\{10\}\)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)',
  false
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    5 |    1 |        2
   6 |        4 |         6 |      10 |   10 |   -1 |    0 |        3
   7 |        1 |        12 |      10 |   12 |   11 |    1 |        0
   8 |        2 |        12 |      10 |   11 |    5 |    1 |        1
   9 |        3 |        12 |      10 |   10 |   -1 |    0 |        2
(9 rows)
```

Equvalences between signatures

The following examples find the path for \(\{6\}\rightarrow\{10\}\)

33) Using One to One

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 10
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 rows)
```

34) Using One to Many

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

35) Using Many to One

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

36) Using Many to Many

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

37) Using Combinations

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES(6, 10)) AS combinations (source, target)'
);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   2 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   3 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   4 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   5 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   6 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
(6 rows)
```

See Also

- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**
- The queries use the **Sample Data** network.

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.3**

pgr_dijkstraCost

`pgr_dijkstraCost` - Total cost of the shortest path(s) using Dijkstra algorithm.

Boost Graph Inside

Availability

- Version 3.1.0
  - New **proposed** signature:
    - `pgr_dijkstraCost` (**Combinations**)
- Version 2.2.0
  - New **Official** function

Description

The `pgr_dijkstraCost` function sumarizes of the cost of the shortest path(s) using Dijkstra Algorithm.

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 to an ending vertex. This implementation can be used with a directed graph and an undirected graph.

- Process is done only on edges with positive costs.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values (u, v) is ∞
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- Running time: $O(| start\ vids | * (V \log V + E))$

- It does not return a path.
- Returns the sum of the costs of the shortest path of each pair combination of nodes requested.
- 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 **aggregate cost** of (u, v) is the same as for (v, u).
- Any duplicated value in the start or end vertex identifiers are ignored.
- The returned values are ordered:
  - `start_vid` ascending
  - `end_vid` ascending

Signatures

Summary

```
pgr_dijkstraCost(Edges SQL, start vid, end vid , [directed])
pgr_dijkstraCost(Edges SQL, start vid, end vids, [directed])
pgr_dijkstraCost(Edges SQL, start vids, end vid , [directed])
pgr_dijkstraCost(Edges SQL, start vids, end vids, [directed])
pgr_dijkstraCost(Edges SQL, Combinations SQL, [directed])

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

One to One

```
pgr_dijkstraCost(Edges SQL, start vid, end vid , [directed])

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

Example:
From vertex 6 to vertex 10 on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 10, true);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
(1 row)
```

One to Many

```
pgr_dijkstraCost(Edges SQL, start vid, end vids, [directed])

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

Example:
From vertex \(6\) to vertices \(\{10, 17\}\) on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, ARRAY[10, 17]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      10 |        5
         6 |      17 |        4
(2 rows)
```

Many to One

```
pgr_dijkstraCost(Edges SQL, start vids, end vid , [directed])

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

Example:
From vertices \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[6, 1], 17);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |      17 |        5
         6 |      17 |        4
(2 rows)
```

Many to Many

```
pgr_dijkstraCost(Edges SQL, start vids, end vids, [directed])

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

Example:
From vertices \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[6, 1], ARRAY[10, 17],
  directed => false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         1 |      10 |        4
         1 |      17 |        5
         6 |      10 |        1
         6 |      17 |        4
(4 rows)
```

Combinations

```
pgr_dijkstraCost(Edges SQL, Combinations SQL, [directed])

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

Example:
Using a combinations table on an **undirected** graph

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        2
         6 |       5 |        1
         6 |      15 |        2
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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	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 edges',
  ARRAY[7, 10, 15, 10, 10, 15], ARRAY[10, 7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 2:
Making **start_vids** the same as **end_vids**

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  ARRAY[7, 10, 15], ARRAY[7, 10, 15]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         7 |      10 |        4
         7 |      15 |        3
        10 |       7 |        2
        10 |      15 |        3
        15 |       7 |        3
        15 |      10 |        1
(6 rows)
```

Example 3:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_dijkstraCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |       7 |        1
         6 |      10 |        5
        12 |      10 |        4
(3 rows)
```

See Also

- **Dijkstra - Family of functions**
- **Sample Data**
- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**

Indices and tables

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

- **Unsupported versions: 2.6 2.5 2.4 2.3**

pgr_dijkstraCostMatrix

`pgr_dijkstraCostMatrix` - Calculates a cost matrix using **pgr_dijkstra**.

Boost Graph Inside

Availability

- Version 3.0.0
  - **Official** function
- Version 2.3.0
  - New **proposed** function

Description

Using Dijkstra algorithm, calculate and return a cost matrix.

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 to an ending vertex. This implementation can be used with a directed graph and an undirected graph.

The main Characteristics are:

- Can be used as input to **pgr_TSP**.
  - Use directly when the resulting matrix is symmetric and there is no ∞ 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 ∞ 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.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The aggregate 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 aggregate cost in the non included values *(u, v)* is ∞.
- Let be the case the values returned are stored in a table:
  - The unique index would be the pair: `(start_vid, end_vid)`.
- Depending on the function and its parameters, the results can be symmetric.
  - The aggregate 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

Signatures

Summary

```
pgr_dijkstraCostMatrix(Edges SQL, start vids, [directed])

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

Example:
Symmetric cost matrix for vertices $\{5, 6, 10, 15\}$ on an **undirected** graph

```
SELECT * FROM pgr_dijkstraCostMatrix(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  (SELECT array_agg(id)
    FROM vertices
    WHERE id IN (5, 6, 10, 15)),
  false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         5 |       6 |        1
         5 |      10 |        2
         5 |      15 |        3
         6 |       5 |        1
         6 |      10 |        1
         6 |      15 |        2
        10 |       5 |        2
        10 |       6 |        1
        10 |      15 |        1
        15 |       5 |        3
        15 |       6 |        2
        15 |      10 |        1
(12 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

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	Aggregate cost from start_vid to end_vid.

Additional Examples

Example:
Use with **pgr_TSP**.

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_dijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edges',
    (SELECT array_agg(id)
      FROM vertices
      WHERE id IN (5, 6, 10, 15)),
    false)
  $$);
NOTICE:  pgr_TSP no longer solving with simulated annaeling
HINT:  Ignoring annaeling parameters
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    5 |    0 |        0
   2 |    6 |    1 |        1
   3 |   10 |    1 |        2
   4 |   15 |    1 |        3
   5 |    5 |    3 |        6
(5 rows)
```

See Also

- **Dijkstra - Family of functions**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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)

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

pgr_drivingDistance(**Edges SQL**, **Root vid**, **distance**, [directed])
pgr_drivingDistance(**Edges SQL**, **Root vids**, **distance**, [**options**])
options: [directed, equicost]

RETURNS SET OF (seq, [from_v,] node, edge, cost, agg_cost)

Single Vertex

```
pgr_drivingDistance(Edges SQL, Root vid, distance, [directed])

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

Example:
From vertex \(11\) for a distance of\(3.0\)

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  11, 3.0);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   11 |   -1 |    0 |        0
   2 |    7 |    8 |    1 |        1
   3 |   12 |   11 |    1 |        1
   4 |   16 |    9 |    1 |        1
   5 |    3 |    7 |    1 |        2
   6 |    6 |    4 |    1 |        2
   7 |    8 |   10 |    1 |        2
   8 |   15 |   16 |    1 |        2
   9 |   17 |   15 |    1 |        2
  10 |    1 |    6 |    1 |        3
  11 |    5 |    1 |    1 |        3
  12 |    9 |   14 |    1 |        3
  13 |   10 |    3 |    1 |        3
(13 rows)
```

Multiple Vertices

```
pgr_drivingDistance(Edges SQL, Root vids, distance, [options])
options: [directed, equicost]

RETURNS SET OF (seq, from_v, node, edge, cost, agg_cost)
```

Example:
From vertices \(\{11, 16\}\) for a distance of\(3.0\) with equi-cost on a directed graph

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  array[11, 16], 3.0, equicost => true);
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |     11 |   11 |   -1 |    0 |        0
   2 |     11 |    7 |    8 |    1 |        1
   3 |     11 |   12 |   11 |    1 |        1
   4 |     11 |    3 |    7 |    1 |        2
   5 |     11 |    6 |    4 |    1 |        2
   6 |     11 |    8 |   10 |    1 |        2
   7 |     11 |    1 |    6 |    1 |        3
   8 |     11 |    5 |    1 |    1 |        3
   9 |     11 |    9 |   14 |    1 |        3
  10 |     16 |   16 |   -1 |    0 |        0
  11 |     16 |   15 |   16 |    1 |        1
  12 |     16 |   17 |   15 |    1 |        1
  13 |     16 |   10 |    3 |    1 |        2
(13 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	Edges SQL as described below.
Root vid	BIGINT	Identifier of the root vertex of the tree.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.
distance	FLOAT	Upper limit for the inclusion of a node in the result.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Optional parameters

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

Driving distance optional parameters

Column	Type	Default	Description
equicost	BOOLEAN	true	• When true the node will only appear in the closest from_v list. • When false which resembles several calls using the single starting point signatures. Tie brakes are arbitrary.

Inner Queries

Edges SQL

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)
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, from_v, node, edge, cost, agg_cost)

Parameter	Type	Description
seq	BIGINT	Sequential value starting from \(1\).
[from_v]	BIGINT	Identifier of the root vertex.
node	BIGINT	Identifier of node within the limits from from_v.
edge	BIGINT	Identifier of the edge used to arrive to node. • \(0\) when node = from_v.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from from_v to node.

Where:

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

Additional Examples

Example:
From vertices \(\{11, 16\}\) for a distance of \(3.0\) on an undirected graph

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  array[11, 16], 3.0, directed => false);
 seq | from_v | node | edge | cost | agg_cost
-----+--------+------+------+------+----------
   1 |     11 |   11 |   -1 |    0 |        0
   2 |     11 |    7 |    8 |    1 |        1
   3 |     11 |   10 |    5 |    1 |        1
   4 |     11 |   12 |   11 |    1 |        1
   5 |     11 |   16 |    9 |    1 |        1
   6 |     11 |    3 |    7 |    1 |        2
   7 |     11 |    6 |    2 |    1 |        2
   8 |     11 |    8 |   10 |    1 |        2
   9 |     11 |   15 |    3 |    1 |        2
  10 |     11 |   17 |   15 |    1 |        2
  11 |     11 |    1 |    6 |    1 |        3
  12 |     11 |    5 |    1 |    1 |        3
  13 |     11 |    9 |   14 |    1 |        3
  14 |     16 |   16 |   -1 |    0 |        0
  15 |     16 |   11 |    9 |    1 |        1
  16 |     16 |   15 |   16 |    1 |        1
  17 |     16 |   17 |   15 |    1 |        1
  18 |     16 |    7 |    8 |    1 |        2
  19 |     16 |   10 |    5 |    1 |        2
  20 |     16 |   12 |   13 |    1 |        2
  21 |     16 |    3 |    7 |    1 |        3
  22 |     16 |    6 |    4 |    1 |        3
  23 |     16 |    8 |   10 |    1 |        3
(23 rows)
```

See Also

- **pgr_alphaShape** - Alpha shape computation
- **Sample Data** network.

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_KSP

`pgr_KSP` — Yen's algorithm for K shortest paths using Dijkstra.

Boost Graph Inside

Availability

- Version 2.1.0
  - Signature change
    - Old signature no longer supported
- Version 2.0.0
  - **Official** function

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**, [**options**])
options: [directed, heap_paths]

RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)

```
OR EMPTY SET
```

Example:

Get 2 paths from \(6\) to \(17\) on a directed graph.

```
SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 17, 2);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    6 |    4 |    1 |        0
   2 |       1 |        2 |    7 |   10 |    1 |        1
   3 |       1 |        3 |    8 |   12 |    1 |        2
   4 |       1 |        4 |   12 |   13 |    1 |        3
   5 |       1 |        5 |   17 |   -1 |    0 |        4
   6 |       2 |        1 |    6 |    4 |    1 |        0
   7 |       2 |        2 |    7 |    8 |    1 |        1
   8 |       2 |        3 |   11 |    9 |    1 |        2
   9 |       2 |        4 |   16 |   15 |    1 |        3
  10 |       2 |        5 |   17 |   -1 |    0 |        4
(10 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	SQL query as described.
start vid	**ANY-INTEGER**	Identifier of the departure vertex.
end vid	**ANY-INTEGER**	Identifier of the departure vertex.
K	**ANY-INTEGER**	Number of required paths

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

Optional parameters

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

KSP Optional parameters

Column	Type	Default	Description
heap_paths	BOOLEAN	false	• When false Returns at most K paths • When true all the calculated paths while processing are returned. • Roughly, when the shortest path has N edges, the heap will contain about than N * K paths for small value of K and K > 5.

Inner Queries

Edges SQL

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)
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	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Path identifier. • Has value **1** for the first of a path from **start vid** to **end_vid**
path_seq	INTEGER	Relative position in the path. Has value **1** for the beginning of a 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.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence. • \(0\) for the last node of the path.
agg_cost	FLOAT	Aggregate cost from **start vid** to node.

Additional Examples

Example:
Get 2 paths from \(6\) to \(17\) on an undirected graph

Also get the paths in the heap.

```
SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 17, 2,
  directed => false, heap_paths => true
);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    6 |    4 |    1 |        0
   2 |       1 |        2 |    7 |   10 |    1 |        1
   3 |       1 |        3 |    8 |   12 |    1 |        2
   4 |       1 |        4 |   12 |   13 |    1 |        3
   5 |       1 |        5 |   17 |   -1 |    0 |        4
   6 |       2 |        1 |    6 |    4 |    1 |        0
   7 |       2 |        2 |    7 |    8 |    1 |        1
   8 |       2 |        3 |   11 |   11 |    1 |        2
   9 |       2 |        4 |   12 |   13 |    1 |        3
  10 |       2 |        5 |   17 |   -1 |    0 |        4
  11 |       3 |        1 |    6 |    4 |    1 |        0
  12 |       3 |        2 |    7 |    8 |    1 |        1
  13 |       3 |        3 |   11 |    9 |    1 |        2
  14 |       3 |        4 |   16 |   15 |    1 |        3
  15 |       3 |        5 |   17 |   -1 |    0 |        4
  16 |       4 |        1 |    6 |    2 |    1 |        0
  17 |       4 |        2 |   10 |    5 |    1 |        1
  18 |       4 |        3 |   11 |    9 |    1 |        2
  19 |       4 |        4 |   16 |   15 |    1 |        3
  20 |       4 |        5 |   17 |   -1 |    0 |        4
(20 rows)
```

See Also

- **K shortest paths - Category**
- **Sample Data**
- **https://en.wikipedia.org/wiki/K_shortest_path_routing**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

pgr_dijkstraVia - Proposed

pgr_dijkstraVia — 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

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 $i < size\_of(via\;vertices)$.

Route:
is a sequence of paths.
Path:
is a section of the route.

Signatures

One Via

pgr_dijkstraVia(**Edges SQL**, **via vertices**, [**options**])
options: [directed, strict, U_turn_on_edge]

RETURNS SET OF (seq, path_id, 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 $\{5, 1, 8\}$ in that order on an **directed** graph.

```
SELECT * FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edges order by id',
  ARRAY[5, 1, 8]);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost | route_agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------+----------------
   1 |       1 |        1 |         5 |       1 |    5 |    1 |    1 |        0 |              0
   2 |       1 |        2 |         5 |       1 |    6 |    4 |    1 |        1 |              1
   3 |       1 |        3 |         5 |       1 |    7 |    7 |    1 |        2 |              2
   4 |       1 |        4 |         5 |       1 |    3 |    6 |    1 |        3 |              3
   5 |       1 |        5 |         5 |       1 |    1 |   -1 |    0 |        4 |              4
   6 |       2 |        1 |         1 |       8 |    1 |    6 |    1 |        0 |              4
   7 |       2 |        2 |         1 |       8 |    3 |    7 |    1 |        1 |              5
   8 |       2 |        3 |         1 |       8 |    7 |   10 |    1 |        2 |              6
   9 |       2 |        4 |         1 |       8 |    8 |   -2 |    0 |        3 |              7
(9 rows)
```

Parameters

Parameter	Type	Default	Description
Edges SQL	TEXT		SQL query as described.
via vertices	ARRAY [**ANY-INTEGER**]		Array of ordered vertices identifiers that are going to be visited.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Optional parameters

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

Via optional parameters

Parameter	Type	Default	Description
strict	BOOLEAN	false	• When true if a path is missing stops and returns **EMPTY SET** • When false ignores missing paths returning all paths found
U_turn_on_edge	BOOLEAN	true	• When true departing from a visited vertex will not try to avoid

Inner Queries

Edges SQL

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)
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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Identifier of a path. Has value **1** for the first path.
path_seq	INTEGER	Relative position in the path. Has value **1** for the beginning of a 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 path sequence.
agg_cost	FLOAT	Aggregate cost from start_vid to node.
route_agg_cost	FLOAT	Total cost from start_vid of seq = 1 to end_vid of the current seq.

Additional Examples

- **The main query**
  - **Aggregate cost of the third path.**
  - **Route's aggregate cost of the route at the end of the third path.**
  - **Nodes visited in the route.**
  - **The aggregate costs of the route when the visited vertices are reached.**
  - **Status of "passes in front" or "visits" of the nodes.**

All this examples are about the route that visits the vertices\(\{5, 7, 1, 8, 15\}\) in that order on a **directed** graph.

The main query

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

Aggregate cost of the third path.

```
SELECT agg_cost FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edges order by id',
  ARRAY[5, 7, 1, 8, 15])
WHERE path_id = 3 AND edge <0;
 agg_cost
----------
        3
(1 row)
```

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 edges order by id',
  ARRAY[5, 7, 1, 8, 15])
WHERE path_id = 3 AND edge < 0;
 route_agg_cost
----------------
              7
(1 row)
```

Nodes visited in the route.

```
SELECT row_number() over () as node_seq, node
FROM  pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edges order by id',
  ARRAY[5, 7, 1, 8, 15])
WHERE edge <> -1 ORDER BY seq;
 node_seq | node
----------+------
        1 |    5
        2 |    6
        3 |    7
        4 |    3
        5 |    1
        6 |    3
        7 |    7
        8 |    8
        9 |   12
       10 |   17
       11 |   16
       12 |   15
(12 rows)
```

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 edges order by id',
  ARRAY[5, 7, 1, 8, 15])
WHERE edge < 0;
 path_id | route_agg_cost
---------+----------------
       1 |              2
       2 |              4
       3 |              7
       4 |             11
(4 rows)
```

Status of "passes in front" or "visits" of the nodes.

```
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 edges order by id',
  ARRAY[5, 7, 1, 8, 15])
WHERE agg_cost  <> 0 or seq = 1;
 seq | route_agg_cost | node | agg_cost |     status
-----+----------------+------+----------+-----------------
   1 |              0 |    5 |        0 | passes in front
   2 |              1 |    6 |        1 | passes in front
   3 |              2 |    7 |        2 | visits
   5 |              3 |    3 |        1 | passes in front
   6 |              4 |    1 |        2 | visits
   8 |              5 |    3 |        1 | passes in front
   9 |              6 |    7 |        2 | passes in front
  10 |              7 |    8 |        3 | visits
  12 |              8 |   12 |        1 | passes in front
  13 |              9 |   17 |        2 | passes in front
  14 |             10 |   16 |        3 | passes in front
  15 |             11 |   15 |        4 | passes in front
(12 rows)
```

See Also

- **Via - Category**.
- **Dijkstra - Family of functions**.
- **Sample Data** network.
- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2**

pgr_dijkstraNear - Proposed

pgr_dijkstraNear — Using Dijkstra's algorithm, finds the route that leads to the nearest vertex.

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 3.3.0
  - Promoted to **proposed** function
- Version 3.2.0
  - New **experimental** function

Description

Given a graph, a starting vertex and a set of ending vertices, this function finds the shortest path from the starting vertex to the nearest ending vertex.

Characteristics

- Uses Dijkstra algorithm.
- Works for **directed** and **undirected** graphs.
- When there are more than one path to the same vertex with same cost:
  - The algorithm will return just one path
- Optionally allows to find more than one path.
  - When more than one path is to be returned:
    - Results are sorted in increasing order of:
      - aggregate cost
      - Within the same value of aggregate costs:
        - results are sorted by (source, target)
- Running time: Dijkstra running time: $drt = O((|E| + |V|)log|V|)$
  - One to Many; drt
  - Many to One: drt
  - Many to Many: $drt * |Starting vids|$
  - Combinations: $drt * |Starting vids|$

Signatures

Summary

```
pgr_dijkstraNear(Edges SQL, start vid, end vids, [options A])
pgr_dijkstraNear(Edges SQL, start vids, end vid, [options A])
pgr_dijkstraNear(Edges SQL, start vids, end vids, [options B])
pgr_dijkstraNear(Edges SQL, Combinations SQL, [options B])
options A: [directed, cap]
options B: [directed, cap, global]

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

One to Many

```
pgr_dijkstraNear(Edges SQL, start vid, end vids, [options])
options: [directed, cap]

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

Example:
Departing on car from vertex 6 find the nearest subway station.

- Using a **directed** graph for car routing.
- The subway stations are on the following vertices $\{1, 10, 11\}$
- The defaults used:
  - *directed => true*
  - *cap => 1*

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

```

The result shows that station at vertex\(11\) is the nearest.

Many to One

pgr_dijkstraNear(**Edges SQL**, **start vids**, **end vid**, [**options**])
options: [directed, cap]

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

Example:
Departing on a car from a subway station find the nearest**two** stations to vertex \(2\)

- Using a **directed** graph for car routing.
- The subway stations are on the following vertices\(\{ 1, 10, 11\}\)
- On line *4*: using the positional parameter: *directed* set to `true`
- In line *5*: using named parameter *cap => 2*

```
1  SELECT * FROM pgr_dijkstraNear(
2   'SELECT id, source, target, cost, reverse_cost FROM edges',
3   ARRAY[10, 11, 1], 6,
4   true,
5   cap => 2);
6   seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
7  -----+----------+-----------+---------+------+------+------+----------
8     1 |        1 |        10 |       6 |   10 |    2 |    1 |        0
9     2 |        2 |        10 |       6 |    6 |   -1 |    0 |        1
10    3 |        1 |        11 |       6 |   11 |    8 |    1 |        0
11    4 |        2 |        11 |       6 |    7 |    4 |    1 |        1
12    5 |        3 |        11 |       6 |    6 |   -1 |    0 |        2
13 (5 rows)
14
```

The result shows that station at vertex\(10\) is the nearest and the next best is\(11\).

Many to Many

pgr_dijkstraNear(**Edges SQL**, **start vids**, **end vids**, [**options**])
options: [directed, cap, global]

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

Example:
Find the best pedestrian connection between two lines of buses

- Unsing an **undirected** graph for pedestrian routing
- The first subway line stations are at\(\{15, 16\}\)
- The second subway line stations stops are at\(\{1, 10, 11\}\)
- On line *4*: using the named parameter: *directed => false*
- The defaults used:
  - *cap => 1*
  - *global => true*

```
1  SELECT * FROM pgr_dijkstraNear(
2   'SELECT id, source, target, cost, reverse_cost FROM edges',
3   ARRAY[15, 16], ARRAY[10, 11, 1],
4   directed => false);
5   seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
6  -----+----------+-----------+---------+------+------+------+----------
7     1 |        1 |        15 |      10 |   15 |    3 |    1 |        0
8     2 |        2 |        15 |      10 |   10 |   -1 |    0 |        1
9  (2 rows)
10
```

For a pedestrian the best connection is to get on/off is at vertex\(15\) of the first subway line and at vertex\(10\) of the second subway line.

Only *one* route is returned because *global* is `true` and *cap* is `1`

Combinations

pgr_dijkstraNear(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, cap, global]

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

Example:
Find the best car connection between all the stations of two subway lines

- Using a **directed** graph for car routing.
- The first subway line stations stops are at\(\{1, 10, 11\}\)
- The second subway line stations are at\(\{15, 16\}\)

The combinations contents:

```
SELECT unnest(ARRAY[10, 11, 1]) as source, target
FROM (SELECT unnest(ARRAY[15, 16]) AS target) a
  UNION
SELECT unnest(ARRAY[15, 16]), target
FROM (SELECT unnest(ARRAY[10, 11, 1]) AS target) b ORDER BY source, target;
 source | target
--------+--------
      1 |     15
      1 |     16
     10 |     15
     10 |     16
     11 |     15
     11 |     16
     15 |      1
     15 |     10
     15 |     11
     16 |      1
     16 |     10
     16 |     11
(12 rows)
```

The query:

- lines *3~4* sets the start vertices to be from the first subway line and the ending vertices to be from the second subway line
- lines *6~7* sets the start vertices to be from the first subway line and the ending vertices to be from the first subway line
- On line *8*: using the named parameter is *global => false*
- The defaults used:
  - *directed => true*
  - *cap => 1*

```
 1 SELECT * FROM pgr_dijkstraNear(
 2   'SELECT id, source, target, cost, reverse_cost FROM edges',
 3   'SELECT unnest(ARRAY[10, 11, 1]) as source, target
 4   FROM (SELECT unnest(ARRAY[15, 16]) AS target) a
 5   UNION
 6   SELECT unnest(ARRAY[15, 16]), target
 7   FROM (SELECT unnest(ARRAY[10, 11, 1]) AS target) b',
 8   global => false);
 9  seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
10 -----+----------+-----------+---------+------+------+------+----------
11    1 |        1 |        11 |      16 |   11 |    9 |    1 |        0
12    2 |        2 |        11 |      16 |   16 |   -1 |    0 |        1
13    3 |        1 |        15 |      10 |   15 |    3 |    1 |        0
14    4 |        2 |        15 |      10 |   10 |   -1 |    0 |        1
15    5 |        1 |        16 |      11 |   16 |    9 |    1 |        0
16    6 |        2 |        16 |      11 |   11 |   -1 |    0 |        1
17    7 |        1 |        10 |      16 |   10 |    5 |    1 |        0
18    8 |        2 |        10 |      16 |   11 |    9 |    1 |        1
19    9 |        3 |        10 |      16 |   16 |   -1 |    0 |        2
20   10 |        1 |         1 |      16 |    1 |    6 |    1 |        0
21   11 |        2 |         1 |      16 |    3 |    7 |    1 |        1
22   12 |        3 |         1 |      16 |    7 |    8 |    1 |        2
23   13 |        4 |         1 |      16 |   11 |    9 |    1 |        3
24   14 |        5 |         1 |      16 |   16 |   -1 |    0 |        4
25 (14 rows)
26
```

From the results:

- making a connection from the first subway line $\{1, 10, 11\}$ to the second $\{15, 16\}$:
  - The best connections from all the stations from the first line are: ${(1 \rightarrow 16) (10 \rightarrow 16) (11 \rightarrow 16)}$
  - The best one is $(11 \rightarrow 16)$ with a cost of 1 (lines: *11* and *12*)
- making a connection from the second subway line $\{15, 16\}$ to the first $\{1, 10, 11\}$:
  - The best connections from all the stations from the second line are: ${(15 \rightarrow 10) (16 \rightarrow 11)}$
  - Both are equaly good as they have the same cost. (lines: *13* and *14* and lines: *15* and *16*)

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Dijkstra optional parameters

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

Near optional parameters

Parameter	Type	Default	Description
cap	BIGINT	1	Find at most cap number of nearest shortest paths
global	BOOLEAN	true	• When true: only cap limit results will be returned • When false: cap limit per Start vid will be returned

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

Returns (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	Relative position in the path. Has value**1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex of the current path.
end_vid	BIGINT	Identifier of the ending vertex of the current path.
node	BIGINT	Identifier of the node in the path fromstart_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go fromnode 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_vid to node.

See Also

- **Dijkstra - Family of functions**
- **pgr_dijkstraNearCost - Proposed**
- **Sample Data** network.
- boost: **https://www.boost.org/libs/graph/doc/table_of_contents.html**
- Wikipedia: **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**

Indices and tables

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

- **Supported versions: Latest** (**3.3**) **3.2**

pgr_dijkstraNearCost - Proposed

pgr_dijkstraNearCost — Using dijkstra algorithm, finds the route that leads to the nearest vertex.

> **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 3.3.0
  - Promoted to **proposed** function
- Version 3.2.0
  - New **experimental** function

Description

Given a graph, a starting vertex and a set of ending vertices, this function finds the shortest path from the starting vertex to the nearest ending vertex.

Characteristics

- Uses Dijkstra algorithm.
- Works for **directed** and **undirected** graphs.
- When there are more than one path to the same vertex with same cost:
  - The algorithm will return just one path
- Optionally allows to find more than one path.
  - When more than one path is to be returned:
    - Results are sorted in increasing order of:
      - aggregate cost
      - Within the same value of aggregate costs:
        - results are sorted by (source, target)
- Running time: Dijkstra running time: \(drt = O((|E| + |V|)log|V|)\)
  - One to Many; \(drt\)
  - Many to One: \(drt\)
  - Many to Many: \(drt * |Starting vids|\)
  - Combinations: \(drt * |Starting vids|\)

Signatures

Summary

pgr_dijkstraNearCost(**Edges SQL**, **start vid**, **end vids**, [**options A**])
pgr_dijkstraNearCost(**Edges SQL**, **start vids**, **end vid**, [**options A**])
pgr_dijkstraNearCost(**Edges SQL**, **start vids**, **end vids**, [**options B**])
pgr_dijkstraNearCost(**Edges SQL**, **Combinations SQL**, [**options B**])
options A: [directed, cap]
options B: [directed, cap, global]

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

One to Many

pgr_dijkstraNearCost(**Edges SQL**, **start vid**, **end vids**, [**options**])
options: [directed, cap]

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

Example:
Departing on car from vertex \(6\) find the nearest subway station.

- Using a **directed** graph for car routing.
- The subway stations are on the following vertices\(\{1, 10, 11\}\)
- The defaults used:
  - *directed => true*
  - *cap => 1*

```
SELECT * FROM pgr_dijkstraNearCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, ARRAY[10, 11, 1]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |      11 |        2
(1 row)

```

The result shows that station at vertex\(11\) is the nearest.

Many to One

pgr_dijkstraNearCost(**Edges SQL**, **start vids**, **end vid**, [**options**])
options: [directed, cap]

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

Example:
Departing on a car from a subway station find the nearest**two** stations to vertex \(6\)

- Using a **directed** graph for car routing.
- The subway stations are on the following vertices\(\{1, 10, 11\}\)
- On line *4*: using the positional parameter: *directed* set to `true`
- In line *5*: using named parameter *cap => 2*

```
1  SELECT * FROM pgr_dijkstraNearCost(
2   'SELECT id, source, target, cost, reverse_cost FROM edges',
3   ARRAY[10, 11, 1], 6,
4   true,
5   cap => 2) ORDER BY agg_cost;
6  start_vid | end_vid | agg_cost
7  -----------+---------+----------
8         10 |       6 |        1
9         11 |       6 |        2
10 (2 rows)
11
```

The result shows that station at vertex\(10\) is the nearest and the next best is\(11\).

Many to Many

pgr_dijkstraNearCost(**Edges SQL**, **start vids**, **end vids**, [**options**])
options: [directed, cap, global]

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

Example:
Find the best pedestrian connection between two lines of buses

- Unsing an **undirected** graph for pedestrian routing
- The first subway line stations are at\(\{15, 16\}\)
- The second subway line stations stops are at\(\{1, 10, 11\}\)
- On line *4*: using the named parameter: *directed => false*
- The defaults used:
    - *cap => 1*
    - *global => true*

```
1 SELECT * FROM pgr_dijkstraNearCost(
2  'SELECT id, source, target, cost, reverse_cost FROM edges',
3  ARRAY[15, 16], ARRAY[10, 11, 1],
4  directed => false);
5 start_vid | end_vid | agg_cost
6 -----------+---------+----------
7        15 |      10 |        1
8 (1 row)
9
```

For a pedestrian the best connection is to get on/off is at vertex\(15\) of the first subway line and at vertex\(10\) of the second subway line.

Only *one* route is returned because *global* is `true` and *cap* is `1`

Combinations

pgr_dijkstraNearCost(**Edges SQL**, **Combinations SQL**, [**options**])
options: [directed, cap, global]

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

Example:
Find the best car connection between all the stations of two subway lines

- Using a **directed** graph for car routing.
- The first subway line stations stops are at\(\{1, 10, 11\}\)
- The second subway line stations are at\(\{15, 16\}\)

The combinations contents:

```
SELECT unnest(ARRAY[10, 11, 1]) as source, target
FROM (SELECT unnest(ARRAY[15, 16]) AS target) a
  UNION
SELECT unnest(ARRAY[15, 16]), target
FROM (SELECT unnest(ARRAY[10, 11, 1]) AS target) b ORDER BY source, target;
 source | target
--------+--------
      1 |     15
      1 |     16
     10 |     15
     10 |     16
     11 |     15
     11 |     16
     15 |      1
     15 |     10
     15 |     11
     16 |      1
     16 |     10
     16 |     11
(12 rows)
```

The query:

- lines *3~4* sets the start vertices to be from the fisrt subway line and the ending vertices to be from the second subway line
- lines *6~7* sets the start vertices to be from the first subway line and the ending vertices to be from the first subway line
- On line *8*: using the named parameter is *global => false*
- The defaults used:
  - *directed => true*
  - *cap => 1*

```
1  SELECT * FROM pgr_dijkstraNearCost(
2    'SELECT id, source, target, cost, reverse_cost FROM edges',
3    'SELECT unnest(ARRAY[10, 11, 1]) as source, target
4     FROM (SELECT unnest(ARRAY[15, 16]) AS target) a
5       UNION
6     SELECT unnest(ARRAY[15, 16]), target
7     FROM (SELECT unnest(ARRAY[10, 11, 1]) AS target) b',
8    global => false);
9   start_vid | end_vid | agg_cost
10 -----------+---------+----------
11         11 |      16 |        1
12         15 |      10 |        1
13         16 |      11 |        1
14         10 |      16 |        2
15          1 |      16 |        4
16 (5 rows)
17
```

From the results:

- making a connection from the first subway line $\{1, 10, 11\}$ to the second $\{15, 16\}$:
  - The best connections from all the stations from the first line are: ${(1 \rightarrow 16) (10 \rightarrow 16) (11 \rightarrow 16)}$
  - The best one is $(11 \rightarrow 16)$ with a cost of 1 (lines: *1*)
- making a connection from the second subway line $\{15, 16\}$ to the first $\{1, 10, 11\}$:
  - The best connections from all the stations from the second line are: ${(15 \rightarrow 10) (16 \rightarrow 11)}$
  - Both are equaly good as they have the same cost. (lines: *12* and *13*)

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Dijkstra optional parameters

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

Near optional parameters

Parameter	Type	Default	Description
cap	BIGINT	1	Find at most cap number of nearest shortest paths
global	BOOLEAN	true	When true: only cap limit results will be returned When false: cap limit per Start vid will be returned

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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	Aggregate cost from start_vid to end_vid.

See Also

- **Dijkstra - Family of functions**
- **pgr_dijkstraNear - Proposed**
- **Sample Data** network.
- boost: **https://www.boost.org/libs/graph/doc/table_of_contents.html**
- Wikipedia: **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**

Indices and tables

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

Introduction

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 to an ending vertex. 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.
  - A negative value on a cost column is interpreted as the edge does not exist.
- Values are returned when there is a path.
- When there is no path:
  - When the starting vertex and ending vertex are the same.
    - The **aggregate cost** of the non included values\((v, v)\) is \(0\)
  - When the starting vertex and ending vertex are the different and there is no path:
    - The **aggregate cost** the non included values\((u, v)\) is \(\infty\)
- For optimization purposes, any duplicated value in the starting vertices or on the ending vertices are ignored.
- Running time: \(O(| start\ vids | * (V \log V + E))\)

The Dijkstra family functions are based on the Dijkstra algorithm.

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

Column	Type	Default	Description
directed	BOOLEAN	true	- When true the graph is considered *Directed* - When false the graph is considered as *Undirected*.

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Advanced documentation

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

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.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4**

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 3.2.0
  - New **proposed** signature
    - pgr_maxFlow (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.4.0
  - New **Proposed** function

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**, **start vid**, **end vid**)
pgr_maxFlow(**Edges SQL**, **start vid**, **end vids**)
pgr_maxFlow(**Edges SQL**, **start vids**, **end vid**)
pgr_maxFlow(**Edges SQL**, **start vids**, **end vids**)
pgr_maxFlow(**Edges SQL**, **Combinations SQL**)

RETURNS BIGINT

One to One

pgr_maxFlow(**Edges SQL**, **start vid**, **end vid**)

RETURNS BIGINT

Example:
From vertex 11 to vertex 12

```
SELECT * FROM pgr_maxFlow(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, 12);
 pgr_maxflow
-------------
         230
(1 row)
```

One to Many

pgr_maxFlow(**Edges SQL**, **start vid**, **end vids**)

RETURNS BIGINT

Example:
From vertex 11 to vertices $\{5, 10, 12\}$

```
SELECT * FROM pgr_maxFlow(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, ARRAY[5, 10, 12]);
 pgr_maxflow
-------------
         340
(1 row)
```

Many to One

```
pgr_maxFlow(Edges SQL, start vids, end vid)
RETURNS BIGINT
```

Example:
From vertices \(\{11, 3, 17\}\) to vertex \(12\)

```
SELECT * FROM pgr_maxFlow(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], 12);
 pgr_maxflow
-------------
         230
(1 row)
```

Many to Many

```
pgr_maxFlow(Edges SQL, start vids, end vids)
RETURNS BIGINT
```

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_maxFlow(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 pgr_maxflow
-------------
         360
(1 row)
```

Combinations

```
pgr_maxFlow(Edges SQL, Combinations SQL)
RETURNS BIGINT
```

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\).

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_maxFlow(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)');
 pgr_maxflow
-------------
          80
(1 row)
```

Parameters

Column	Type	Description

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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)
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

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

Type	Description
BIGINT	Maximum flow possible from the source(s) to the target(s)

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_maxFlow(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)');
 pgr_maxflow
-------------
          80
(1 row)
```

See Also

- **Flow - Family of functions**
  - **pgr_pushRelabel**
- **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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

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 3.2.0
  - New **proposed** signature
    - pgr_boykovKolmogorov (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from pgr_maxFlowBoykovKolmogorov
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

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, start vid, end vid)
pgr_boykovKolmogorov(Edges SQL, start vid, end vids)
pgr_boykovKolmogorov(Edges SQL, start vids, end vid)
pgr_boykovKolmogorov(Edges SQL, start vids, end vids)
pgr_boykovKolmogorov(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One

```
pgr_boykovKolmogorov(Edges SQL, start vid, end vid)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertex \(11\) to vertex \(12\)

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, 12);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         7 |       8 |  100 |                30
   2 |   12 |         8 |      12 |  100 |                 0
   3 |    8 |        11 |       7 |  100 |                30
   4 |   11 |        11 |      12 |  130 |                 0
(4 rows)
```

One to Many

```
pgr_boykovKolmogorov(Edges SQL, start vid, end vids)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertex \(11\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, ARRAY[5, 10, 12]);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         6 |       5 |   50 |                80
   2 |    4 |         7 |       6 |   50 |                 0
   3 |   10 |         7 |       8 |   80 |                50
   4 |   12 |         8 |      12 |   80 |                20
   5 |    8 |        11 |       7 |  130 |                 0
   6 |   11 |        11 |      12 |  130 |                 0
   7 |    9 |        11 |      16 |   80 |                50
   8 |    3 |        15 |      10 |   80 |                50
   9 |   16 |        16 |      15 |   80 |                 0
(9 rows)
```

Many to One

```
pgr_boykovKolmogorov(Edges SQL, start vids, end vid)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertices \(\{11, 3, 17\}\) to vertex \(12\)

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], 12);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    7 |         3 |       7 |   50 |                 0
   2 |   10 |         7 |       8 |  100 |                30
   3 |   12 |         8 |      12 |  100 |                 0
   4 |    8 |        11 |       7 |   50 |                80
   5 |   11 |        11 |      12 |  130 |                 0
(5 rows)
```

Many to Many

```
pgr_boykovKolmogorov(Edges SQL, start vids, end vids)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    7 |         3 |       7 |   50 |                 0
   2 |    1 |         6 |       5 |   50 |                80
   3 |    4 |         7 |       6 |   50 |                 0
   4 |   10 |         7 |       8 |  100 |                30
   5 |   12 |         8 |      12 |  100 |                 0
   6 |    8 |        11 |       7 |  100 |                30
   7 |   11 |        11 |      12 |  130 |                 0
   8 |    9 |        11 |      16 |   80 |                50
   9 |    3 |        15 |      10 |   80 |                50
  10 |   16 |        16 |      15 |   80 |                 0
(10 rows)
```

Combinations

```
pgr_boykovKolmogorov(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\).

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)');
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    4 |         6 |       7 |   80 |                20
   2 |    8 |         7 |      11 |   80 |                20
   3 |    9 |        11 |      16 |   80 |                50
   4 |   16 |        16 |      15 |   80 |                 0
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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)
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

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

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).

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)');
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    4 |         6 |       7 |   80 |                20
   2 |    8 |         7 |      11 |   80 |                20
   3 |    9 |        11 |      16 |   80 |                50
   4 |   16 |        16 |      15 |   80 |                 0
(4 rows)
```

See Also

- **Flow - Family of functions**
  - **pgr_edmondsKarp**
  - **pgr_pushRelabel**
- **https://www.boost.org/libs/graph/doc/boykov_kolmogorov_max_flow.html**

Indices and tables

pgr_edmondsKarp

`pgr_edmondsKarp` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Edmonds Karp Algorithm.

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** signature
    - `pgr_edmondsKarp` (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from `pgr_maxFlowEdmondsKarp`
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

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, start vid, end vid)
pgr_edmondsKarp(Edges SQL, start vid, end vids)
pgr_edmondsKarp(Edges SQL, start vids, end vid)
pgr_edmondsKarp(Edges SQL, start vids, end vids)
pgr_edmondsKarp(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One

```
pgr_edmondsKarp(Edges SQL, start vid, end vid)
```

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

Example:
From vertex \(11\) to vertex \(12\)

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, 12);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         7 |       8 |  100 |                30
   2 |   12 |         8 |      12 |  100 |                 0
   3 |    8 |        11 |       7 |  100 |                30
   4 |   11 |        11 |      12 |  130 |                 0
(4 rows)
```

One to Many

pgr_edmondsKarp(**Edges SQL**, **start vid**, **end vids**)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

Example:
From vertex \(11\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, ARRAY[5, 10, 12]);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    1 |         6 |       5 |   50 |                80
   2 |    4 |         7 |       6 |   50 |                 0
   3 |   10 |         7 |       8 |   80 |                50
   4 |   12 |         8 |      12 |   80 |                20
   5 |    8 |        11 |       7 |  130 |                 0
   6 |   11 |        11 |      12 |  130 |                 0
   7 |    9 |        11 |      16 |   80 |                50
   8 |    3 |        15 |      10 |   80 |                50
   9 |   16 |        16 |      15 |   80 |                 0
(9 rows)
```

Many to One

pgr_edmondsKarp(**Edges SQL**, **start vids**, **end vid**)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

Example:
From vertices \(\{11, 3, 17\}\) to vertex \(12\)

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], 12);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    7 |         3 |       7 |   50 |                 0
   2 |   10 |         7 |       8 |  100 |                30
   3 |   12 |         8 |      12 |  100 |                 0
   4 |    8 |        11 |       7 |   50 |                80
   5 |   11 |        11 |      12 |  130 |                 0
(5 rows)
```

Many to Many

pgr_edmondsKarp(**Edges SQL**, **start vids**, **end vids**)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)

```
OR EMPTY SET
```

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    7 |         3 |       7 |   50 |                 0
   2 |    1 |         6 |       5 |   50 |                80
   3 |    4 |         7 |       6 |   50 |                 0
   4 |   10 |         7 |       8 |  100 |                30
   5 |   12 |         8 |      12 |  100 |                 0
   6 |    8 |        11 |       7 |  100 |                30
   7 |   11 |        11 |      12 |  130 |                 0
   8 |    9 |        11 |      16 |   80 |                50
   9 |    3 |        15 |      10 |   80 |                50
  10 |   16 |        16 |      15 |   80 |                 0
(10 rows)
```

Combinations

```
pgr_edmondsKarp(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\).

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)');
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    4 |         6 |       7 |   80 |                20
   2 |    8 |         7 |      11 |   80 |                20
   3 |    9 |        11 |      16 |   80 |                50
   4 |   16 |        16 |      15 |   80 |                 0
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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.
capacity	**ANY-INTEGER**		Weight of the edge (source, target)
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

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

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).

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)');
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    4 |         6 |       7 |   80 |                20
   2 |    8 |         7 |      11 |   80 |                20
   3 |    9 |        11 |      16 |   80 |                50
   4 |   16 |        16 |      15 |   80 |                 0
(4 rows)
```

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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

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 3.2.0
  - New **proposed** signature
    - pgr_pushRelabel (**Combinations**)
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from pgr_maxFlowPushRelabel
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

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, start vid, end vid)
pgr_pushRelabel(Edges SQL, start vid, end vids)
pgr_pushRelabel(Edges SQL, start vids, end vid)
pgr_pushRelabel(Edges SQL, start vids, end vids)
pgr_pushRelabel(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One

```
pgr_pushRelabel(Edges SQL, start vid, end vid)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
```

OR EMPTY SET

Example:
From vertex \(11\) to vertex \(12\)

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, 12);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         7 |       8 |  100 |                30
   2 |   12 |         8 |      12 |  100 |                 0
   3 |    8 |        11 |       7 |  100 |                30
   4 |   11 |        11 |      12 |  130 |                 0
(4 rows)
```

One to Many

pgr_pushRelabel(**Edges SQL**, **start vid**, **end vids**)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

Example:
From vertex \(11\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  11, ARRAY[5, 10, 12]);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    6 |         1 |       3 |   50 |                 0
   2 |    6 |         3 |       1 |   50 |                50
   3 |    7 |         3 |       7 |   50 |                 0
   4 |    1 |         6 |       5 |   30 |               100
   5 |    7 |         7 |       3 |   50 |                80
   6 |    4 |         7 |       6 |   30 |                20
   7 |   10 |         7 |       8 |  100 |                30
   8 |   12 |         8 |      12 |  100 |                 0
   9 |    8 |        11 |       7 |  130 |                 0
  10 |   11 |        11 |      12 |  130 |                 0
  11 |    9 |        11 |      16 |   80 |                50
  12 |    3 |        15 |      10 |   80 |                50
  13 |   16 |        16 |      15 |   80 |                 0
(13 rows)
```

Many to One

pgr_pushRelabel(**Edges SQL**, **start vids**, **end vid**)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

Example:
From vertices \(\{11, 3, 17\}\) to vertex \(12\)

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], 12);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |   10 |         7 |       8 |  100 |                30
   2 |   12 |         8 |      12 |  100 |                 0
   3 |    8 |        11 |       7 |  100 |                30
   4 |   11 |        11 |      12 |  130 |                 0
(4 rows)
```

Many to Many

pgr_pushRelabel(**Edges SQL**, **start vids**, **end vids**)

```
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    7 |         3 |       7 |   20 |                30
   2 |    1 |         6 |       5 |   50 |                80
   3 |    4 |         7 |       6 |   50 |                 0
   4 |   10 |         7 |       8 |  100 |                30
   5 |   12 |         8 |      12 |  100 |                 0
   6 |    8 |        11 |       7 |  130 |                 0
   7 |   11 |        11 |      12 |  130 |                 0
   8 |    9 |        11 |      16 |   80 |                50
   9 |    3 |        15 |      10 |   80 |                50
  10 |   16 |        16 |      15 |   80 |                 0
(10 rows)
```

Combinations

```
pgr_pushRelabel(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\).

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)');
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    4 |         6 |       7 |   80 |                20
   2 |    8 |         7 |      11 |   80 |                20
   3 |   11 |        11 |      12 |   50 |                80
   4 |    9 |        11 |      16 |   30 |               100
   5 |   13 |        12 |      17 |   50 |                50
   6 |   16 |        16 |      15 |   80 |                 0
   7 |   15 |        17 |      16 |   50 |                 0
(7 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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)
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

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

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).

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id, source, target, capacity, reverse_capacity
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)');
 seq | edge | start_vid | end_vid | flow | residual_capacity
-----+------+-----------+---------+------+-------------------
   1 |    4 |         6 |       7 |   80 |                20
   2 |    8 |         7 |      11 |   80 |                20
   3 |   11 |        11 |      12 |   50 |                80
   4 |    9 |        11 |      16 |   30 |               100
   5 |   13 |        12 |      17 |   50 |                50
   6 |   16 |        16 |      15 |   80 |                 0
   7 |   15 |        17 |      16 |   50 |                 0
(7 rows)
```

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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

pgr_edgeDisjointPaths

pgr_edgeDisjointPaths — Calculates edge disjoint paths between two groups of vertices.

Boost Graph Inside

Availability

- Version 3.2.0
  - New **proposed** function:
    - pgr_edgeDisjointPaths(Combinations)
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

Description

Calculates the edge disjoint paths between two groups of vertices. Utilizes underlying maximum flow algorithms to calculate the paths.

The main characteristics 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.
- Uses **pgr_boykovKolmogorov** to calculate the paths.

Signatures

Summary

```
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])
pgr_edgeDisjointPaths(Edges SQL, Combinations SQL, [directed])

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

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 \(11\) to vertex \(12\)

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges',
  11, 12);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   11 |    8 |    1 |        0
   2 |       1 |        2 |    7 |   10 |    1 |        1
   3 |       1 |        3 |    8 |   12 |    1 |        2
   4 |       1 |        4 |   12 |   -1 |    0 |        3
   5 |       2 |        1 |   11 |   11 |    1 |        0
   6 |       2 |        2 |   12 |   -1 |    0 |        1
(6 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 11 to vertices $\{5, 10, 12\}$

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges',
  11, ARRAY[5, 10, 12]);
 seq | path_id | path_seq | end_vid | node | edge | cost | agg_cost
-----+---------+----------+---------+------+------+------+----------
   1 |       1 |        1 |       5 |   11 |    8 |    1 |        0
   2 |       1 |        2 |       5 |    7 |    4 |    1 |        1
   3 |       1 |        3 |       5 |    6 |    1 |    1 |        2
   4 |       1 |        4 |       5 |    5 |   -1 |    0 |        3
   5 |       2 |        1 |      10 |   11 |    9 |    1 |        0
   6 |       2 |        2 |      10 |   16 |   16 |    1 |        1
   7 |       2 |        3 |      10 |   15 |    3 |    1 |        2
   8 |       2 |        4 |      10 |   10 |   -1 |    0 |        3
   9 |       3 |        1 |      12 |   11 |    8 |    1 |        0
  10 |       3 |        2 |      12 |    7 |   10 |    1 |        1
  11 |       3 |        3 |      12 |    8 |   12 |    1 |        2
  12 |       3 |        4 |      12 |   12 |   -1 |    0 |        3
  13 |       4 |        1 |      12 |   11 |   11 |    1 |        0
  14 |       4 |        2 |      12 |   12 |   -1 |    0 |        1
(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 $\{11, 3, 17\}$ to vertex 12

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

Many to Many

```
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
```

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------
   1 |       1 |        1 |         3 |       5 |    3 |    7 |    1 |        0
   2 |       1 |        2 |         3 |       5 |    7 |    4 |    1 |        1
   3 |       1 |        3 |         3 |       5 |    6 |    1 |    1 |        2
   4 |       1 |        4 |         3 |       5 |    5 |   -1 |    0 |        3
   5 |       2 |        1 |         3 |      10 |    3 |    7 |    1 |        0
   6 |       2 |        2 |         3 |      10 |    7 |    8 |    1 |        1
   7 |       2 |        3 |         3 |      10 |   11 |    9 |    1 |        2
   8 |       2 |        4 |         3 |      10 |   16 |   16 |    1 |        3
   9 |       2 |        5 |         3 |      10 |   15 |    3 |    1 |        4
  10 |       2 |        6 |         3 |      10 |   10 |   -1 |    0 |        5
  11 |       3 |        1 |         3 |      12 |    3 |    7 |    1 |        0
  12 |       3 |        2 |         3 |      12 |    7 |    8 |    1 |        1
  13 |       3 |        3 |         3 |      12 |   11 |   11 |    1 |        2
  14 |       3 |        4 |         3 |      12 |   12 |   -1 |    0 |        3
  15 |       4 |        1 |        11 |       5 |   11 |    8 |    1 |        0
  16 |       4 |        2 |        11 |       5 |    7 |    4 |    1 |        1
  17 |       4 |        3 |        11 |       5 |    6 |    1 |    1 |        2
  18 |       4 |        4 |        11 |       5 |    5 |   -1 |    0 |        3
  19 |       5 |        1 |        11 |      10 |   11 |    9 |    1 |        0
  20 |       5 |        2 |        11 |      10 |   16 |   16 |    1 |        1
  21 |       5 |        3 |        11 |      10 |   15 |    3 |    1 |        2
  22 |       5 |        4 |        11 |      10 |   10 |   -1 |    0 |        3
  23 |       6 |        1 |        11 |      12 |   11 |    8 |    1 |        0
  24 |       6 |        2 |        11 |      12 |    7 |   10 |    1 |        1
  25 |       6 |        3 |        11 |      12 |    8 |   12 |    1 |        2
  26 |       6 |        4 |        11 |      12 |   12 |   -1 |    0 |        3
  27 |       7 |        1 |        11 |      12 |   11 |   11 |    1 |        0
  28 |       7 |        2 |        11 |      12 |   12 |   -1 |    0 |        1
  29 |       8 |        1 |        17 |       5 |   17 |   15 |    1 |        0
  30 |       8 |        2 |        17 |       5 |   16 |   16 |    1 |        1
  31 |       8 |        3 |        17 |       5 |   15 |    3 |    1 |        2
  32 |       8 |        4 |        17 |       5 |   10 |    2 |    1 |        3
  33 |       8 |        5 |        17 |       5 |    6 |    1 |    1 |        4
  34 |       8 |        6 |        17 |       5 |    5 |   -1 |    0 |        5
  35 |       9 |        1 |        17 |      10 |   17 |   15 |    1 |        0
  36 |       9 |        2 |        17 |      10 |   16 |   16 |    1 |        1
  37 |       9 |        3 |        17 |      10 |   15 |    3 |    1 |        2
  38 |       9 |        4 |        17 |      10 |   10 |   -1 |    0 |        3
  39 |      10 |        1 |        17 |      12 |   17 |   15 |    1 |        0
  40 |      10 |        2 |        17 |      12 |   16 |    9 |    1 |        1
  41 |      10 |        3 |        17 |      12 |   11 |   11 |    1 |        2
  42 |      10 |        4 |        17 |      12 |   12 |   -1 |    0 |        3
(42 rows)
```

Combinations

```
pgr_edgeDisjointPaths(Edges SQL, Combinations SQL, [directed])

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

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\) on an undirected graph.

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)',
  directed => false);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------
   1 |       1 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   2 |       1 |        2 |         5 |      10 |    6 |    2 |   -1 |        1
   3 |       1 |        3 |         5 |      10 |   10 |   -1 |    0 |        0
   4 |       2 |        1 |         6 |      15 |    6 |    4 |    1 |        0
   5 |       2 |        2 |         6 |      15 |    7 |    8 |    1 |        1
   6 |       2 |        3 |         6 |      15 |   11 |    9 |    1 |        2
   7 |       2 |        4 |         6 |      15 |   16 |   16 |    1 |        3
   8 |       2 |        5 |         6 |      15 |   15 |   -1 |    0 |        4
   9 |       3 |        1 |         6 |      15 |    6 |    2 |   -1 |        0
  10 |       3 |        2 |         6 |      15 |   10 |    3 |   -1 |       -1
  11 |       3 |        3 |         6 |      15 |   15 |   -1 |    0 |       -2
(11 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Path identifier. • Has value **1** for the first of a path from start_vid to end_vid.
path_seq	INTEGER	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** • **Combinations**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many** • **Combinations**
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_vid to node.

Additional Examples

Example:
Manually assigned vertex combinations on an undirected graph.

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)',
  directed => false);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------
   1 |       1 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   2 |       1 |        2 |         5 |      10 |    6 |    2 |   -1 |        1
   3 |       1 |        3 |         5 |      10 |   10 |   -1 |    0 |        0
   4 |       2 |        1 |         6 |      15 |    6 |    4 |    1 |        0
   5 |       2 |        2 |         6 |      15 |    7 |    8 |    1 |        1
   6 |       2 |        3 |         6 |      15 |   11 |    9 |    1 |        2
   7 |       2 |        4 |         6 |      15 |   16 |   16 |    1 |        3
   8 |       2 |        5 |         6 |      15 |   15 |   -1 |    0 |        4
   9 |       3 |        1 |         6 |      15 |    6 |    2 |   -1 |        0
  10 |       3 |        2 |         6 |      15 |   10 |    3 |   -1 |       -1
  11 |       3 |        3 |         6 |      15 |   15 |   -1 |    0 |       -2
(11 rows)
```

See Also

- **Flow - Family of functions**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

pgr_maxCardinalityMatch

`pgr_maxCardinalityMatch` — Calculates a maximum cardinality matching in a graph.

Boost Graph Inside

Availability

- Version 3.3.3
  - `directed` optional parameter ignored and removed from the documentation as the algorithm works only on undirected graphs
- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from `pgr_maximumCardinalityMatching`
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

Description

The main characteristics are:

- Works for **undirected** graphs.
- 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.
- Running time: \(O(E*V * \alpha(E,V))\)
  - \(\alpha(E,V)\) is the inverse of the **Ackermann function**.

Signatures

```
pgr_maxCardinalityMatch(Edges SQL)

RETURNS SET OF (seq, edge_id, source, target)
OR EMPTY SET
```

Example:
Using all edges.

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

Parameters

Parameter	Type	Description
Edges SQL	`TEXT`	**Edges SQL** as described below.

Inner Queries

Edges SQL

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.
going	**ANY-NUMERICAL**		A positive value represents the existence of the edge (source, target).
coming	**ANY-NUMERICAL**	-1	A positive value represents the existence of the edge (target, source)

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

ANY-NUMERICAL:
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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_maxFlowMinCost - Experimental

pgr_maxFlowMinCost — Calculates the edges that minimizes the total cost of the maximum flow on a 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 3.2.0
  - New **experimental** function:
    - `pgr_maxFlowMinCost` (**Combinations**)
- Version 3.0.0
- New **experimental** function

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, start vid, end vid)
pgr_maxFlowMinCost(Edges SQL, start vid, end vids)
pgr_maxFlowMinCost(Edges SQL, start vids, end vid)
pgr_maxFlowMinCost(Edges SQL, start vids, end vids)
pgr_maxFlowMinCost(Edges SQL, Combinations SQL)

RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

One to One

```
pgr_maxFlowMinCost(Edges SQL, start vid, end vid)

RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

Example:

From vertex \(11\) to vertex \(12\)

```
SELECT * FROM pgr_maxFlowMinCost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  11, 12);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |   10 |      7 |      8 |  100 |                30 |  100 |      100
   2 |   12 |      8 |     12 |  100 |                 0 |  100 |      200
   3 |    8 |     11 |      7 |  100 |                30 |  100 |      300
   4 |   11 |     11 |     12 |  130 |                 0 |  130 |      430
(4 rows)
```

One to Many

pgr_maxFlowMinCost(**Edges SQL**, **start vid**, **end vids**)

RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET

Example:
From vertex \(11\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_maxFlowMinCost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  11, ARRAY[5, 10, 12]);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    1 |      6 |      5 |   30 |               100 |   30 |       30
   2 |    4 |      7 |      6 |   30 |                20 |   30 |       60
   3 |   10 |      7 |      8 |  100 |                30 |  100 |      160
   4 |   12 |      8 |     12 |  100 |                 0 |  100 |      260
   5 |    8 |     11 |      7 |  130 |                 0 |  130 |      390
   6 |   11 |     11 |     12 |  130 |                 0 |  130 |      520
   7 |    9 |     11 |     16 |   80 |                50 |   80 |      600
   8 |    3 |     15 |     10 |   80 |                50 |   80 |      680
   9 |   16 |     16 |     15 |   80 |                 0 |   80 |      760
(9 rows)
```

Many to One

pgr_maxFlowMinCost(**Edges SQL**, **start vids**, **end vid**)

RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET

Example:
From vertices \(\{11, 3, 17\}\) to vertex \(12\)

```
SELECT * FROM pgr_maxFlowMinCost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  ARRAY[11, 3, 17], 12);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    7 |      3 |      7 |   50 |                 0 |   50 |       50
   2 |   10 |      7 |      8 |  100 |                30 |  100 |      150
   3 |   12 |      8 |     12 |  100 |                 0 |  100 |      250
   4 |    8 |     11 |      7 |   50 |                80 |   50 |      300
   5 |   11 |     11 |     12 |  130 |                 0 |  130 |      430
(5 rows)
```

Many to Many

pgr_maxFlowMinCost(**Edges SQL**, **start vids**, **end vids**)

RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_maxFlowMinCost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    7 |      3 |      7 |   50 |                 0 |   50 |       50
   2 |    1 |      6 |      5 |   50 |                80 |   50 |      100
   3 |    4 |      7 |      6 |   50 |                 0 |   50 |      150
   4 |   10 |      7 |      8 |  100 |                30 |  100 |      250
   5 |   12 |      8 |     12 |  100 |                 0 |  100 |      350
   6 |    8 |     11 |      7 |  100 |                30 |  100 |      450
   7 |   11 |     11 |     12 |  130 |                 0 |  130 |      580
   8 |    9 |     11 |     16 |   30 |               100 |   30 |      610
   9 |    3 |     15 |     10 |   80 |                50 |   80 |      690
  10 |   16 |     16 |     15 |   80 |                 0 |   80 |      770
  11 |   15 |     17 |     16 |   50 |                 0 |   50 |      820
(11 rows)
```

Combinations

pgr_maxFlowMinCost(**Edges SQL**, **Combinations SQL**)

RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\).

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_maxFlowMinCost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)');
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    4 |      6 |      7 |   80 |                20 |   80 |       80
   2 |    8 |      7 |     11 |   80 |                20 |   80 |      160
   3 |    9 |     11 |     16 |   80 |                50 |   80 |      240
   4 |   16 |     16 |     15 |   80 |                 0 |   80 |      320
(4 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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
capacity	**ANY-INTEGER**		Capacity of the edge (source, target) • When negative: edge (target, source) 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 exist
reverse_cost	**ANY-NUMERICAL**	\(-1\)	Weight of the edge (target, source) if it exist

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

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).
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.

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_maxFlowMinCost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)');
 seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+------+--------+--------+------+-------------------+------+----------
   1 |    4 |      6 |      7 |   80 |                20 |   80 |       80
   2 |    8 |      7 |     11 |   80 |                20 |   80 |      160
   3 |    9 |     11 |     16 |   80 |                50 |   80 |      240
   4 |   16 |     16 |     15 |   80 |                 0 |   80 |      320
(4 rows)
```

See Also

- **Flow - Family of functions**
- **https://www.boost.org/libs/graph/doc/successive_shortest_path_nonnegative_weights.html**

Indices and tables

pgr_maxFlowMinCost_Cost - Experimental

pgr_maxFlowMinCost_Cost — Calculates the minimum total cost of the maximum flow on a 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 3.2.0
  - New **experimental** function:
    - pgr_maxFlowMinCost_Cost (**Combinations**)
- Version 3.0.0
  - New **experimental** function

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

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 **pgr_maxFlowMinCost - Experimental**.

- 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_Cost(Edges SQL, start vid, end vid)
pgr_maxFlowMinCost_Cost(Edges SQL, start vid, end vids)
pgr_maxFlowMinCost_Cost(Edges SQL, start vids, end vid)
pgr_maxFlowMinCost_Cost(Edges SQL, start vids, end vids)
pgr_maxFlowMinCost_Cost(Edges SQL, Combinations SQL)

RETURNS FLOAT
```

One to One

```
pgr_maxFlowMinCost_Cost(Edges SQL, start vid, end vid)

RETURNS FLOAT
```

Example:
From vertex \(11\) to vertex \(12\)

```
SELECT * FROM pgr_maxFlowMinCost_Cost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  11, 12);
 pgr_maxflowmincost_cost
-------------------------
                     430
(1 row)
```

One to Many

```
pgr_maxFlowMinCost_Cost(Edges SQL, start vid, end vids)

RETURNS FLOAT
```

Example:
From vertex \(11\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_maxFlowMinCost_Cost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  ARRAY[11, 3, 17], 12);
 pgr_maxflowmincost_cost
-------------------------
                     430
(1 row)
```

Many to One

```
pgr_maxFlowMinCost_Cost(Edges SQL, start vids, end vid)

RETURNS FLOAT
```

Example:
From vertices \(\{11, 3, 17\}\) to vertex \(12\)

```
SELECT * FROM pgr_maxFlowMinCost_Cost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  11, ARRAY[5, 10, 12]);
 pgr_maxflowmincost_cost
-------------------------
                     760
(1 row)
```

Many to Many

```
pgr_maxFlowMinCost_Cost(Edges SQL, start vids, end vids)

RETURNS FLOAT
```

Example:
From vertices \(\{11, 3, 17\}\) to vertices \(\{5, 10, 12\}\)

```
SELECT * FROM pgr_maxFlowMinCost_Cost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  ARRAY[11, 3, 17], ARRAY[5, 10, 12]);
 pgr_maxflowmincost_cost
-------------------------
                     820
(1 row)
```

Combinations

```
pgr_maxFlowMinCost_Cost(Edges SQL, Combinations SQL)

RETURNS FLOAT
```

Example:
Using a combinations table, equivalent to calculating result from vertices\(\{5, 6\}\) to vertices \(\{10, 15, 14\}\).

The combinations table:

```
SELECT source, target FROM combinations
WHERE target NOT IN (5, 6);
 source | target
--------+--------
      5 |     10
      6 |     15
      6 |     14
(3 rows)
```

The query:

```
SELECT * FROM pgr_maxFlowMinCost_Cost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  'SELECT * FROM combinations WHERE target NOT IN (5, 6)');
 pgr_maxflowmincost_cost
-------------------------
                     320
(1 row)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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 (target, source) 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 exist
reverse_cost	**ANY-NUMERICAL**	\(-1\)	Weight of the edge (target, source) if it exist

Where:

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Resturn Columns

Type	Description
FLOAT	Minimum Cost Maximum Flow possible from the source(s) to the target(s)

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_maxFlowMinCost_Cost(
  'SELECT id, source, target, capacity, reverse_capacity, cost, reverse_cost
  FROM edges',
  'SELECT * FROM (VALUES (5, 10), (6, 15), (6, 14)) AS t(source, target)');
 pgr_maxflowmincost_cost
-------------------------
                     320
(1 row)
```

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**

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.

Inner Queries

Edges SQL

Capacity edges

- **pgr_pushRelabel**
- **pgr_edmondsKarp**
- **pgr_boykovKolmogorov**

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

Capacity-Cost edges

- **pgr_maxFlowMinCost - Experimental**
- **pgr_maxFlowMinCost_Cost - Experimental**

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 (target, source) 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 exist
reverse_cost	**ANY-NUMERICAL**	\(-1\)	Weight of the edge (target, source) if it exist

Where:

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

Cost edges

- **pgr_edgeDisjointPaths**

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

Used in

- **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).
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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

Kruskal - Family of functions

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

Boost Graph Inside

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_kruskal

pgr_kruskal — Minimum spanning tree of a graph using Kruskal’s algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

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.
- 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.
- The total weight of all the edges in the tree or forest is minimized.
- 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 (edge, cost)
OR EMPTY SET
```

Example:
Minimum spanning forest

```
SELECT * FROM pgr_kruskal(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges 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	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_kruskalBFS

pgr_kruskalBFS — Kruskal's algorithm for Minimum Spanning Tree with breadth First Search ordering.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Description

Visits and extracts the nodes information in Breath 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.
- 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.
- The total weight of all the edges in the tree or forest is minimized.
- 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 6

```
SELECT * FROM pgr_kruskalBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     3 |         6 |   16 |   16 |    1 |        3
   6 |     4 |         6 |   17 |   15 |    1 |        4
   7 |     5 |         6 |   12 |   13 |    1 |        5
   8 |     6 |         6 |   11 |   11 |    1 |        6
   9 |     6 |         6 |    8 |   12 |    1 |        6
  10 |     7 |         6 |    7 |   10 |    1 |        7
  11 |     7 |         6 |    9 |   14 |    1 |        7
  12 |     8 |         6 |    3 |    7 |    1 |        8
  13 |     9 |         6 |    1 |    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 $\{9, 6\}$ with $depth \leq 3$

```
SELECT * FROM pgr_kruskalBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  ARRAY[9, 6], max_depth => 3);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     3 |         6 |   16 |   16 |    1 |        3
   6 |     0 |         9 |    9 |   -1 |    0 |        0
   7 |     1 |         9 |    8 |   14 |    1 |        1
   8 |     2 |         9 |    7 |   10 |    1 |        2
   9 |     2 |         9 |   12 |   12 |    1 |        2
  10 |     3 |         9 |    3 |    7 |    1 |        3
  11 |     3 |         9 |   11 |   11 |    1 |        3
  12 |     3 |         9 |   17 |   13 |    1 |        3
(12 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • 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. • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

BFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit of the depth of the tree. • When negative throws an error.

Inner Queries

Edges SQL

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)
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)

Parameter	Type	Description

Parameter	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.
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.

Where:

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

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- **Sample Data**
- **Boost: Kruskal's algorithm**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_kruskalDD

pgr_kruskalDD — Catchament nodes using Kruskal's algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Description

Using Kruskal's algorithm, extracts the nodes that have aggregate costs less than or equal to a **distance** 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.
- 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.

- The total weight of all the edges in the tree or forest is minimized.

- Kruskal’s running time: \(O(E * log E)\)
- 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.
- Edge \((u, v)\) will not be included when:
  - The distance from the **root** to \(u\) > limit distance.
  - The distance from the **root** to \(v\) > limit distance.
  - No new nodes are created on the graph, so when is within the limit and is not within the limit, the edge is not included.
- 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 \(6\) with \(distance \leq 3.5\)

```
SELECT * FROM pgr_kruskalDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  6, 3.5);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     3 |         6 |   16 |   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 \(\{9, 6\}\) with \(distance \leq 3.5\)

```
SELECT * FROM pgr_kruskalDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  ARRAY[9, 6], 3.5);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     3 |         6 |   16 |   16 |    1 |        3
   6 |     0 |         9 |    9 |   -1 |    0 |        0
   7 |     1 |         9 |    8 |   14 |    1 |        1
   8 |     2 |         9 |    7 |   10 |    1 |        2
   9 |     3 |         9 |    3 |    7 |    1 |        3
  10 |     2 |         9 |   12 |   12 |    1 |        2
  11 |     3 |         9 |   11 |   11 |    1 |        3
  12 |     3 |         9 |   17 |   13 |    1 |        3
(12 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	Edges SQL as described below.

Parameter	Type	Description
Root vid	BIGINT	Identifier of the root vertex of the tree.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.
distance	FLOAT	Upper limit for the inclusion of a node in the result.

Where:

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

Inner Queries

Edges SQL

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)
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)

Parameter	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.
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.

Where:

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

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- **Sample Data**
- **Boost: Kruskal's algorithm**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_kruskalDFS

pgr_kruskalDFS — Kruskal's algorithm for Minimum Spanning Tree with Depth First Search ordering.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

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.
- 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.

- The total weight of all the edges in the tree or forest is minimized.
- 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 having as root vertex 6

```
SELECT * FROM pgr_kruskalDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     3 |         6 |   16 |   16 |    1 |        3
   6 |     4 |         6 |   17 |   15 |    1 |        4
   7 |     5 |         6 |   12 |   13 |    1 |        5
   8 |     6 |         6 |   11 |   11 |    1 |        6
   9 |     6 |         6 |    8 |   12 |    1 |        6
  10 |     7 |         6 |    7 |   10 |    1 |        7
  11 |     8 |         6 |    3 |    7 |    1 |        8
  12 |     9 |         6 |    1 |    6 |    1 |        9
  13 |     7 |         6 |    9 |   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 \(\{9, 6\}\) with \(depth \leq 3\)

```
SELECT * FROM pgr_kruskalDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  ARRAY[9, 6], max_depth => 3);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     3 |         6 |   16 |   16 |    1 |        3
   6 |     0 |         9 |    9 |   -1 |    0 |        0
   7 |     1 |         9 |    8 |   14 |    1 |        1
   8 |     2 |         9 |    7 |   10 |    1 |        2
   9 |     3 |         9 |    3 |    7 |    1 |        3
  10 |     2 |         9 |   12 |   12 |    1 |        2
  11 |     3 |         9 |   11 |   11 |    1 |        3
  12 |     3 |         9 |   17 |   13 |    1 |        3
(12 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • 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. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

DFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	\(9223372036854775807\)	Upper limit of the depth of the tree. • When negative throws an error.

Inner Queries

Edges SQL

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)
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)

Parameter	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.
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.

Where:

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

See Also

- **Spanning Tree - Category**
- **Kruskal - Family of functions**
- **Sample Data**
- **Boost: Kruskal's algorithm**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

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

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.
- 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.
- The total weight of all the edges in the tree or forest is minimized.
- Kruskal's running time: \(O(E * log E)\)

Inner Queries

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)
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**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

Prim - Family of functions

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

Boost Graph Inside

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_prim

pgr_prim — Minimum spanning forest of a graph using Prim's algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

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 edges WHERE id < 14'
) ORDER BY edge;
 edge | cost
------+------
    1 |    1
    2 |    1
    3 |    1
    4 |    1
    6 |    1
    7 |    1
    8 |    1
    9 |    1
   10 |    1
   12 |    1
   13 |    1
(11 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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**

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**

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

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 \(6\)

```
SELECT * FROM pgr_primBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     1 |         6 |    7 |    4 |    1 |        1
   5 |     2 |         6 |   15 |    3 |    1 |        2
   6 |     2 |         6 |   11 |    5 |    1 |        2
   7 |     2 |         6 |    3 |    7 |    1 |        2
   8 |     2 |         6 |    8 |   10 |    1 |        2
   9 |     3 |         6 |   16 |    9 |    1 |        3
  10 |     3 |         6 |   12 |   11 |    1 |        3
  11 |     3 |         6 |    1 |    6 |    1 |        3
  12 |     3 |         6 |    9 |   14 |    1 |        3
  13 |     4 |         6 |   17 |   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 \(\{9, 6\}\) with \(depth \leq 3\)

```
SELECT * FROM pgr_primBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  ARRAY[9, 6], max_depth => 3);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     1 |         6 |    7 |    4 |    1 |        1
   5 |     2 |         6 |   15 |    3 |    1 |        2
   6 |     2 |         6 |   11 |    5 |    1 |        2
   7 |     2 |         6 |    3 |    7 |    1 |        2
   8 |     2 |         6 |    8 |   10 |    1 |        2
   9 |     3 |         6 |   16 |    9 |    1 |        3
  10 |     3 |         6 |   12 |   11 |    1 |        3
  11 |     3 |         6 |    1 |    6 |    1 |        3
  12 |     3 |         6 |    9 |   14 |    1 |        3
  13 |     0 |         9 |    9 |   -1 |    0 |        0
  14 |     1 |         9 |    8 |   14 |    1 |        1
  15 |     2 |         9 |    7 |   10 |    1 |        2
  16 |     3 |         9 |    6 |    4 |    1 |        3
  17 |     3 |         9 |    3 |    7 |    1 |        3
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • When value is \(0\) then gets the spanning forest starting in aleatory nodes for each tree in the forest.

Parameter	Type	Description
root vids	ARRAY [**ANY-INTEGER**]	Array of identifiers of the root vertices. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

BFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	\(9223372036854775807\)	Upper limit of the depth of the tree. • When negative throws an error.

Inner Queries

Edges SQL

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)
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)

Parameter	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.
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.

Where:

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

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: Latest (3.3) 3.2 3.1 3.0**

pgr_primDD

`pgr_primDD` — Catchament nodes using Prim's algorithm.

Boost Graph Inside

Availability

- Version 3.0.0
  - New **Official** function

Description

Using Prim's algorithm, extracts the nodes that have aggregate costs less than or equal to a **distance** 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.
- 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: $O(E * log V)$

- 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.
- Edge (u, v) will not be included when:
  - The distance from the **root** to u > limit distance.
  - The distance from the **root** to v > limit distance.
  - No new nodes are created on the graph, so when is within the limit and is not within the limit, the edge is not included.

- Returned tree nodes from a root vertex are on Depth First Search order.
- Depth First Search running time: $O(E + V)$

Signatures

```
pgr_primDD(Edges SQL, root vid, distance)
pgr_primDD(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 \(6\) with \(distance \leq 3.5\)

```
SELECT * FROM pgr_primDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  6, 3.5);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     2 |         6 |   11 |    5 |    1 |        2
   6 |     3 |         6 |   16 |    9 |    1 |        3
   7 |     3 |         6 |   12 |   11 |    1 |        3
   8 |     1 |         6 |    7 |    4 |    1 |        1
   9 |     2 |         6 |    3 |    7 |    1 |        2
  10 |     3 |         6 |    1 |    6 |    1 |        3
  11 |     2 |         6 |    8 |   10 |    1 |        2
  12 |     3 |         6 |    9 |   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 \(\{9, 6\}\) with \(distance \leq 3.5\)

```
SELECT * FROM pgr_primDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  ARRAY[9, 6], 3.5);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     2 |         6 |   11 |    5 |    1 |        2
   6 |     3 |         6 |   16 |    9 |    1 |        3
   7 |     3 |         6 |   12 |   11 |    1 |        3
   8 |     1 |         6 |    7 |    4 |    1 |        1
   9 |     2 |         6 |    3 |    7 |    1 |        2
  10 |     3 |         6 |    1 |    6 |    1 |        3
  11 |     2 |         6 |    8 |   10 |    1 |        2
  12 |     3 |         6 |    9 |   14 |    1 |        3
  13 |     0 |         9 |    9 |   -1 |    0 |        0
  14 |     1 |         9 |    8 |   14 |    1 |        1
  15 |     2 |         9 |    7 |   10 |    1 |        2
  16 |     3 |         9 |    6 |    4 |    1 |        3
  17 |     3 |         9 |    3 |    7 |    1 |        3
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	Edges SQL as described below.
Root vid	BIGINT	Identifier of the root vertex of the tree.
Root vids	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.
distance	FLOAT	Upper limit for the inclusion of a node in the result.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

ANY-NUMERIC:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Inner Queries

Edges SQL

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)
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)

Parameter	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.
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.

Where:

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

See Also

- **Spanning Tree - Category**
- **Prim - Family of functions**
- **Sample Data**
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

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

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 \(6\)

```
SELECT * FROM pgr_primDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     2 |         6 |   11 |    5 |    1 |        2
   6 |     3 |         6 |   16 |    9 |    1 |        3
   7 |     3 |         6 |   12 |   11 |    1 |        3
   8 |     4 |         6 |   17 |   13 |    1 |        4
   9 |     1 |         6 |    7 |    4 |    1 |        1
  10 |     2 |         6 |    3 |    7 |    1 |        2
  11 |     3 |         6 |    1 |    6 |    1 |        3
  12 |     2 |         6 |    8 |   10 |    1 |        2
  13 |     3 |         6 |    9 |   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 \(\{9, 6\}\) with \(depth \leq 3\)

```
SELECT * FROM pgr_primDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  ARRAY[9, 6], max_depth => 3);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     2 |         6 |   11 |    5 |    1 |        2
   6 |     3 |         6 |   16 |    9 |    1 |        3
   7 |     3 |         6 |   12 |   11 |    1 |        3
   8 |     1 |         6 |    7 |    4 |    1 |        1
   9 |     2 |         6 |    3 |    7 |    1 |        2
  10 |     3 |         6 |    1 |    6 |    1 |        3
  11 |     2 |         6 |    8 |   10 |    1 |        2
  12 |     3 |         6 |    9 |   14 |    1 |        3
  13 |     0 |         9 |    9 |   -1 |    0 |        0
  14 |     1 |         9 |    8 |   14 |    1 |        1
  15 |     2 |         9 |    7 |   10 |    1 |        2
  16 |     3 |         9 |    6 |    4 |    1 |        3
  17 |     3 |         9 |    3 |    7 |    1 |        3
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • 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. • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

DFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit of the depth of the tree. • When negative throws an error.

Inner Queries

Edges SQL

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)
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)

Parameter	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.
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.

Where:

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

See Also

- **Spanning Tree - Category**
- **Prim - Family of functions**
- **Sample Data**
- **Boost: Prim's algorithm documentation**
- **Wikipedia: Prim's algorithm**

Indices and tables

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

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 Queries

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)
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**
- Wikipedia: **Prim's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3)**

Reference

- **pgr_version**
- **pgr_full_version**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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

Description

Returns pgRouting version information.

Signature

```
pgr_version()
RETURNS TEXT
```

Example:
pgRouting Version for this documentation

```
SELECT pgr_version();
 pgr_version
-------------
 3.3.4
(1 row)
```

Result Columns

Type	Description
TEXT	pgRouting version

See Also

- Reference
- pgr_full_version

Indices and tables

- Index
- Search Page

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_full_version

pgr_full_version — Get the details of pgRouting version information.

Availability

- Version 3.0.0
  - New **official** function

Description

Get complete details of pgRouting version information

Signatures

```
pgr_full_version()

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

Example:
Information about when this documentation was built

```
SELECT version, library FROM pgr_full_version();
 version |      library
---------+-----------------
 3.3.4   | pgrouting-3.3.4
(1 row)
```

Return 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

Column	Type	Description
hash	TEXT	Git hash of pgRouting build

See Also

- **Reference**
- **pgr_version**

Indices and tables

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

See Also

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

Topology - Family of Functions

The pgRouting's topology of a network represented with a graph in form of two tables: and edge table and a vertex table.

Attributes associated to the tables help to indicate if the graph is directed or undirected, if an edge is one way on a directed graph, and depending on the final application needs, suitable topology(s) need to be created.

pgRouting suplies some functions to create a routing topology and to analyze the topology.

Additional functions to create a graph:

- **Contraction - Family of functions**

Additional functions to analyze a graph:

- **Components - Family of functions**

The following functions modify the database directly therefore the user must have special permissions given by the administrators to use them.

- **pgr_createTopology** - create a topology based on the geometry.
- **pgr_createVerticesTable** - 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.

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.

These proposed functions do not modify the database.

- **pgr_extractVertices – Proposed** - Extracts vertex information based on the edge table information.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_createTopology

pgr_createTopology — Builds a network topology based on the geometry information.

Availability

- Version 2.0.0
  - Renamed from version 1.x
  - **Official** function

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

```
pgr_createTopology(edge_table, tolerance, [options])
options: [the_geom, id, source, target, rows_where, clean]

RETURNS VARCHAR
```

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 issource.
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 wheresource 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 theid 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('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

When the arguments are given in the order described:

We get the same result AS the simplest way to use the function.

```
SELECT  pgr_createTopology('edges', 0.001,
    'geom', 'id', 'source', 'target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_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('edges', 0.001,
    'id', 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, 'id', 'geom', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  ----> PGR ERROR in pgr_createTopology: Wrong type of Column id:geom
HINT:    ----> Expected type of geom is integer,smallint or bigint but USER-DEFINED was found
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('edges', 0.001,
    the_geom:='geom', id:='id', source:='source', target:='target');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('edges', 0.001,
    source:='source', id:='id', target:='target', the_geom:='geom');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('edges', 0.001, 'geom', source:='source');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Selecting rows based on the id.

```
SELECT  pgr_createTopology('edges', 0.001, 'geom', 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('edges', 0.001, 'geom',
    rows_where:='geom && (SELECT st_buffer(geom, 0.05) FROM edges 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('edges', 0.001, 'geom',
    rows_where:='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,  geom AS mygeom, source AS src , target AS tgt FROM edges) ;
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 appropiriate 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
HINT:    ----> Expected type of mygeom is integer,smallint or bigint but USER-DEFINED was found
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

Create a routing topology

An alternate method to create a routing topology use **pgr_extractVertices - Proposed**

Make 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
```

Clean up the columns of the routing topology to be created

```
UPDATE edges
SET source = NULL, target = NULL,
x1 = NULL, y1 = NULL,
x2 = NULL, y2 = NULL;
UPDATE 18
```

Create the vertices table

- When the `LINESTRING` has a SRID then use `geom::geometry(POINT, <SRID>)`
- For big edge tables that are been prepared,
  - Create it as `UNLOGGED` and

- After the table is created `ALTER TABLE .. SET LOGGED`

```
SELECT  * INTO vertices_table
FROM pgr_extractVertices('SELECT id, geom FROM edges ORDER BY id');
SELECT 17
```

Inspect the vertices table

```
SELECT *
FROM vertices_table;
 id | in_edges | out_edges |       x        |  y  |                    geom
----+----------+-----------+----------------+-----+--------------------------------------------
  1 |          | {6}       |              0 |   2 | 010100000000000000000000000000000000000040
  2 |          | {17}      |            0.5 | 3.5 | 0101000000000000000000E03F0000000000000C40
  3 | {6}      | {7}       |              1 |   2 | 0101000000000000000000F03F0000000000000040
  4 | {17}     |           | 1.999999999999 | 3.5 | 010100000068EEFFFFFFFFFF3F0000000000000C40
  5 |          | {1}       |              2 |   0 | 010100000000000000000000400000000000000000
  6 | {1}      | {2,4}     |              2 |   1 | 01010000000000000000000040000000000000F03F
  7 | {4,7}    | {8,10}    |              2 |   2 | 010100000000000000000000400000000000000040
  8 | {10}     | {12,14}   |              2 |   3 | 010100000000000000000000400000000000000840
  9 | {14}     |           |              2 |   4 | 010100000000000000000000400000000000001040
 10 | {2}      | {3,5}     |              3 |   1 | 01010000000000000000000840000000000000F03F
 11 | {5,8}    | {9,11}    |              3 |   2 | 010100000000000000000008400000000000000040
 12 | {11,12}  | {13}      |              3 |   3 | 010100000000000000000008400000000000000840
 13 |          | {18}      |            3.5 | 2.3 | 0101000000000000000000000C406666666666660240
 14 | {18}     |           |            3.5 |   4 | 0101000000000000000000000C400000000000001040
 15 | {3}      | {16}      |              4 |   1 | 01010000000000000000001040000000000000F03F
 16 | {9,16}   | {15}      |              4 |   2 | 010100000000000000000010400000000000000040
 17 | {13,15}  |           |              4 |   3 | 010100000000000000000010400000000000000840
(17 rows)
```

Create the routing topology on the edge table

Updating the `source` information

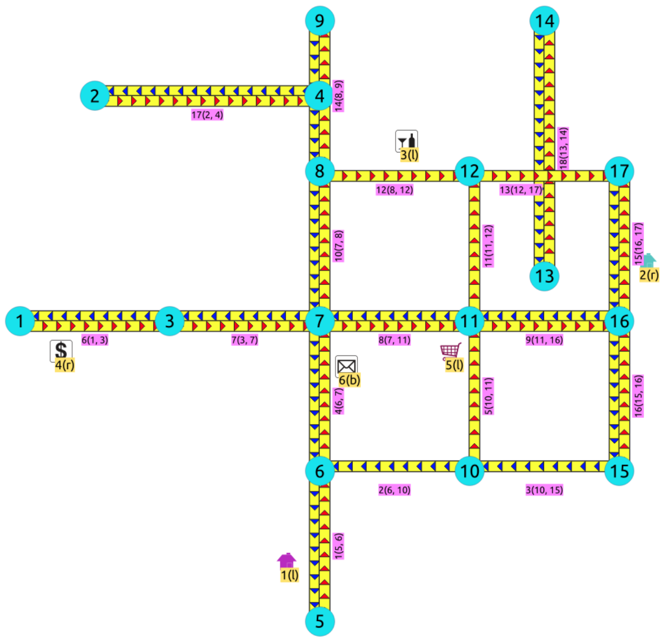
```
WITH
out_going AS (
  SELECT id AS vid, unnest(out_edges) AS eid, x, y
  FROM vertices_table
)
UPDATE edges
SET source = vid, x1 = x, y1 = y
FROM out_going WHERE id = eid;
UPDATE 18
```

Updating the `target` information

```
WITH
in_coming AS (
  SELECT id AS vid, unnest(in_edges) AS eid, x, y
  FROM vertices_table
)
UPDATE edges
SET target = vid, x2 = x, y2 = y
FROM in_coming WHERE id = eid;
UPDATE 18
```

Inspect the routing topology

```
SELECT id, source, target, x1, y1, x2, y2
FROM edges ORDER BY id;
 id | source | target | x1  | y1  |       x2       | y2
----+--------+--------+-----+-----+----------------+-----
  1 |      5 |      6 |   2 |   0 |              2 |   1
  2 |      6 |     10 |   2 |   1 |              3 |   1
  3 |     10 |     15 |   3 |   1 |              4 |   1
  4 |      6 |      7 |   2 |   1 |              2 |   2
  5 |     10 |     11 |   3 |   1 |              3 |   2
  6 |      1 |      3 |   0 |   2 |              1 |   2
  7 |      3 |      7 |   1 |   2 |              2 |   2
  8 |      7 |     11 |   2 |   2 |              3 |   2
  9 |     11 |     16 |   3 |   2 |              4 |   2
 10 |      7 |      8 |   2 |   2 |              2 |   3
 11 |     11 |     12 |   3 |   2 |              3 |   3
 12 |      8 |     12 |   2 |   3 |              3 |   3
 13 |     12 |     17 |   3 |   3 |              4 |   3
 14 |      8 |      9 |   2 |   3 |              2 |   4
 15 |     16 |     17 |   4 |   2 |              4 |   3
 16 |     15 |     16 |   4 |   1 |              4 |   2
 17 |      2 |      4 | 0.5 | 3.5 | 1.999999999999 | 3.5
 18 |     13 |     14 | 3.5 | 2.3 |            3.5 |   4
(18 rows)
```



Generated topology

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('edges',  0.001, 'geom', rows_where:='id < 6', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT pgr_createTopology('edges',  0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

The example uses the **Sample Data** network.

See Also

- **Topology - Family of Functions**
- **pgr_createVerticesTable**
- **pgr_analyzeGraph**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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

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('edges', 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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('edges', 'geom', 'source', 'target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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('edges', 'source', 'geom', 'target');
> NOTICE:  PROCESSING:
> NOTICE:  pgr_createVerticesTable('edges','source','geom','target','true')
> NOTICE:  Performing checks, please wait .....
> NOTICE:  ----> PGR ERROR in pgr_createVerticesTable: Wrong type of Column source: geom
> HINT:    ----> Expected type of 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('edges', the_geom:='geom', source:='source', target:='target');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createverticestable
-------------------------
 OK
(1 row)
```

Example 4:
Using a different ordering

```
SELECT  pgr_createVerticesTable('edges', source:='source', target:='target', the_geom:='geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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('edges', 'geom', source:='source');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','true')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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('edges', 'geom', rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','id < 10')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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('edges', 'geom',
    rows_where:='geom && (select st_buffer(geom,0.5) FROM edges WHERE id=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','geom && (select st_buffer(geom,0.5) FROM edges WHERE id=5)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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('edges', 'geom',
    rows_where:='geom && (select st_buffer(other_geom,0.5) FROM otherTable WHERE gid=100)');
NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edges','geom','source','target','geom && (select st_buffer(other_geom,0.5) FROM otherTable WHERE gid=100)')
NOTICE:  Performing checks, please wait .....
NOTICE:  Populating public.edges_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.edges is: public.edges_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, geom AS mygeom, source AS src ,target AS tgt FROM edges) ;
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 column `src` 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 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**

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_analyzeGraph

`pgr_analyzeGraph` — Analyzes the network topology.

Availability

- Version 2.0.0
  - **Official** function

Description

The function returns:

- `OK` after the analysis has finished.
- `FAIL` when the analysis was not completed due to an error.

```
pgr_analyzeGraph(edge_table, tolerance, [options])
options: [the_geom, id, source, target, rows_where]

RETURNS VARCHAR
```

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('edges',0.001, 'geom', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT  pgr_analyzeGraph('edges',0.001,'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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)
```

Arguments are given in the order described in the parameters:

```
SELECT  pgr_analyzeGraph('edges',0.001,'geom','id','source','target');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001,'id','geom','source','target');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'id','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.edges
 pgr_analyzegraph
------------------
 FAIL
(1 row)
```

When using the named notation

The order of the parameters do not matter:

```
SELECT  pgr_analyzeGraph('edges',0.001,the_geom:='geom',id:='id',source:='source',target:='target');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001,source:='source',id:='id',target:='target',the_geom:='geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001, 'geom', source:='source');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001, 'geom', rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001, 'geom', rows_where:='geom && (SELECT st_buffer(geom,0.05) FROM edge_table WHERE id=5)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'geom','id','source','target','geom && (SELECT st_buffer(geom,0.05) FROM edge_table WHERE id=5)')
NOTICE:  Performing checks, please wait ...
NOTICE:  Got relation "edge_table" does not exist
NOTICE:  ERROR: Condition is not correct. Please execute the following query to test your condition
NOTICE:  select count(*) from public.edges WHERE true    AND (geom && (SELECT st_buffer(geom,0.05) FROM edge_table WHERE id=5))
 pgr_analyzegraph
------------------
 FAIL
(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('edges',0.001, 'geom', rows_where:='geom && (SELECT st_buffer(geom,1) FROM otherTable WHERE gid=100)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'geom','id','source','target','geom && (SELECT st_buffer(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: 7
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 , geom AS mygeom FROM edges);
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 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_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('edges',0.001, 'geom', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

SELECT pgr_analyzeGraph('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001,'geom', rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001,'geom', rows_where:='id >= 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges',0.001,'geom', rows_where:='id < 17');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges', 0.001,'geom', rows_where:='id <17', clean := true);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT pgr_analyzeGraph('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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**
- **pgr_analyzeOneWay**
- **pgr_createVerticesTable**
- **pgr_nodeNetwork** to create nodes to a not noded edge table.

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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

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

```
pgr_analyzeOneWay(geom_table, s_in_rules, s_out_rules, t_in_rules, t_out_rules, [options])
options: [oneway, source, target, two_way_if_null]

RETURNS TEXT
```

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

```
ALTER TABLE edges ADD COLUMN dir TEXT;
ALTER TABLE
SELECT  pgr_createTopology('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, 'geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED FOR  0 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)

UPDATE edges SET
dir = CASE WHEN (cost>0 AND reverse_cost>0) THEN 'B'
           WHEN (cost>0 AND reverse_cost<0) THEN 'FT'
           WHEN (cost<0 AND reverse_cost>0) THEN 'TF'
           ELSE '' END;
UPDATE 18
SELECT pgr_analyzeOneWay('edges',
       ARRAY['', 'B', 'TF'],
       ARRAY['', 'B', 'FT'],
       ARRAY['', 'B', 'FT'],
       ARRAY['', 'B', 'TF'],
       oneway:='dir');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeOneway('edges','{"",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)
```

See Also

- **Topology - Family of Functions**
- **pgr_analyzeGraph**
- **pgr_createVerticesTable**
- **Sample Data**

Indices and tables

- **Index**

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, [**options**])
| **options:** [id, text the_geom, table_ending, rows_where, outall]

| RETURNS TEXT

Availability

- Version 2.0.0
  - **Official** function

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('edges', 0.001, 'geom', clean := TRUE);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

Now we can analyze the network.

```
SELECT pgr_analyzegraph('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges', 0.001, the_geom => 'geom');
NOTICE:  PROCESSING:
NOTICE:  id: id
NOTICE:  the_geom: geom
NOTICE:  table_ending: noded
NOTICE:  rows_where:
NOTICE:  outall: f
NOTICE:  pgr_nodeNetwork('edges', 0.001, 'id', '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.edges_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 edges_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

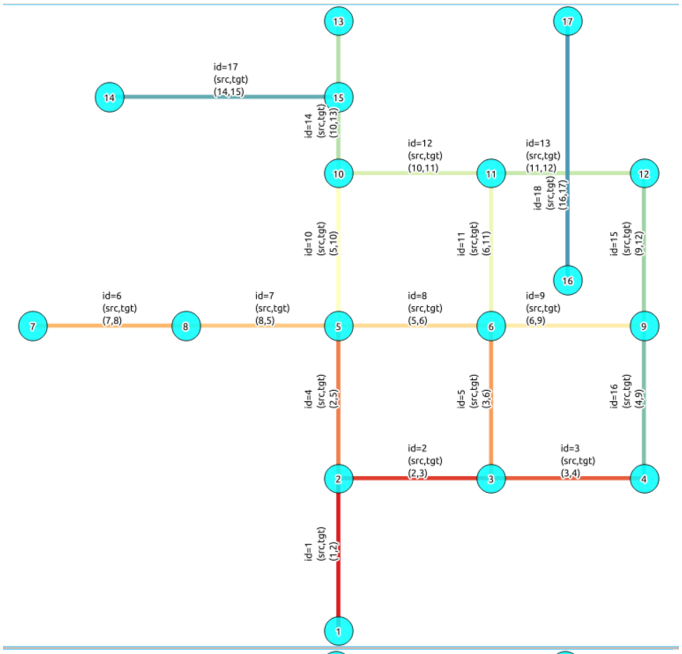
```
SELECT pgr_createTopology('edges_noded', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges_noded', 0.001, 'geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED FOR  21 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edges_noded is: public.edges_noded_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

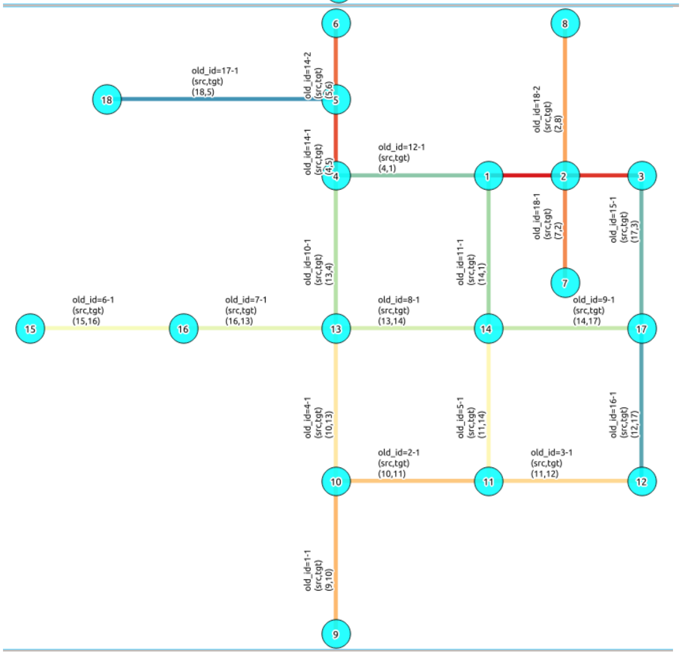
Now let’s analyze the new topology

```
SELECT pgr_analyzegraph('edges_noded', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges_noded',0.001,'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 edges drop column if exists old_id;
NOTICE:  column "old_id" of relation "edges" does not exist, skipping
ALTER TABLE
alter table edges add column old_id integer;
ALTER TABLE
insert into edges (old_id, cost, reverse_cost, geom)
  (with
     segmented as (select old_id,count(*) as i from edges_noded group by old_id)
  select  segments.old_id, cost, reverse_cost, segments.geom
     from edges as edges join edges_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('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edges', 0.001, '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.edges is: public.edges_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('edges', 0.001, 'geom', rows_where:='id not in (select old_id from edges where old_id is not null)');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'geom','id','source','target','id not in (select old_id from edges 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('edges', 0.001, 'geom', rows_where:='old_id is null');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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('edges', 0.001, 'geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edges',0.001,'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**

- **Supported versions: Latest** (**3.3**) **3.2** **3.1** **3.0**

pgr_extractVertices – Proposed

`pgr_extractVertices` — Extracts the vertices information

> **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.

Availability

- Version 3.3.0
  - Classified as **proposed** function
- Version 3.0.0
  - New **experimental** function

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)
OR EMTPY SET
```

Example:
Extracting the vertex information

```
SELECT  * FROM pgr_extractVertices(
  'SELECT id, geom FROM edges');
 id | in_edges | out_edges |       x        |  y  |                    geom
----+----------+-----------+----------------+-----+--------------------------------------------
  1 |          | {6}       |              0 |   2 | 010100000000000000000000000000000000000040
  2 |          | {17}      |            0.5 | 3.5 | 0101000000000000000000E03F0000000000000C40
  3 | {6}      | {7}       |              1 |   2 | 0101000000000000000000F03F0000000000000040
  4 | {17}     |           | 1.999999999999 | 3.5 | 010100000068EEFFFFFFFFFF3F0000000000000C40
  5 |          | {1}       |              2 |   0 | 010100000000000000000000400000000000000000
  6 | {1}      | {2,4}     |              2 |   1 | 01010000000000000000000040000000000000F03F
  7 | {4,7}    | {8,10}    |              2 |   2 | 010100000000000000000000400000000000000040
  8 | {10}     | {12,14}   |              2 |   3 | 010100000000000000000000400000000000000840
  9 | {14}     |           |              2 |   4 | 010100000000000000000000400000000000001040
 10 | {2}      | {3,5}     |              3 |   1 | 01010000000000000000000840000000000000F03F
 11 | {5,8}    | {9,11}    |              3 |   2 | 010100000000000000000008400000000000000040
 12 | {11,12}  | {13}      |              3 |   3 | 010100000000000000000008400000000000000840
 13 |          | {18}      |            3.5 | 2.3 | 01010000000000000000000C406666666666660240
 14 | {18}     |           |            3.5 |   4 | 01010000000000000000000C400000000000001040
 15 | {3}      | {16}      |              4 |   1 | 01010000000000000000001040000000000000F03F
 16 | {9,16}   | {15}      |              4 |   2 | 010100000000000000000010400000000000000040
 17 | {13,15}  |           |              4 |   3 | 010100000000000000000010400000000000000840
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below

Optional parameters

Parameter	Type	Default	Description
dryrun	BOOLEAN	false	• When true do not process and get in a NOTICE the resulting query.

Inner Queries

- **Edges SQL**
  - **When line geometry is known**
  - **When vertex geometry is known**
  - **When identifiers of vertices are known**

Edges SQL

When line geometry is known

Column	Type	Description
id	BIGINT	(Optional) identifier of the edge.
geom	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.
target	ANY-INTEGER	Identifier of the second end point vertex of the edge.

Result Columns

Column	Type	Description
id	BIGINT	Vertex identifier
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	X 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

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, geom FROM edges',
  dryrun => true);
NOTICE:
        WITH

        main_sql AS (
          SELECT id, geom FROM edges
        ),

        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)
```

Create a routing topology

Make 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
```

Clean up the columns of the routing topology to be created

```
UPDATE edges
SET source = NULL, target = NULL,
x1 = NULL, y1 = NULL,
x2 = NULL, y2 = NULL;
UPDATE 18
```

Create the vertices table

- When the `LINESTRING` has a SRID then use `geom::geometry(POINT, <SRID>)`
- For big edge tables that are been prepared,
  - Create it as `UNLOGGED` and
  - After the table is created `ALTER TABLE .. SET LOGGED`

```
SELECT  * INTO vertices_table
FROM pgr_extractVertices('SELECT id, geom FROM edges ORDER BY id');
SELECT 17
```

Inspect the vertices table

```
SELECT *
FROM vertices_table;
 id | in_edges | out_edges |       x        |  y  |                    geom
----+----------+-----------+----------------+-----+--------------------------------------------
  1 |          | {6}       |              0 |   2 | 010100000000000000000000000000000000000040
  2 |          | {17}      |            0.5 | 3.5 | 0101000000000000000000E03F0000000000000C40
  3 | {6}      | {7}       |              1 |   2 | 0101000000000000000000F03F0000000000000040
  4 | {17}     |           | 1.999999999999 | 3.5 | 010100000068EEFFFFFFFFFF3F0000000000000C40
  5 |          | {1}       |              2 |   0 | 010100000000000000000000400000000000000000
  6 | {1}      | {2,4}     |              2 |   1 | 0101000000000000000000004000000000000F03F
  7 | {4,7}    | {8,10}    |              2 |   2 | 010100000000000000000000400000000000000040
  8 | {10}     | {12,14}   |              2 |   3 | 010100000000000000000000400000000000000840
  9 | {14}     |           |              2 |   4 | 010100000000000000000000400000000000001040
 10 | {2}      | {3,5}     |              3 |   1 | 0101000000000000000000084000000000000F03F
 11 | {5,8}    | {9,11}    |              3 |   2 | 010100000000000000000008400000000000000040
 12 | {11,12}  | {13}      |              3 |   3 | 010100000000000000000008400000000000000840
 13 |          | {18}      |            3.5 | 2.3 | 0101000000000000000000C0406666666666660240
 14 | {18}     |           |            3.5 |   4 | 0101000000000000000000C400000000000001040
 15 | {3}      | {16}      |              4 |   1 | 0101000000000000000000104000000000000F03F
 16 | {9,16}   | {15}      |              4 |   2 | 010100000000000000000010400000000000000040
 17 | {13,15}  |           |              4 |   3 | 010100000000000000000010400000000000000840
(17 rows)
```

Create the routing topology on the edge table

Updating the `source` information

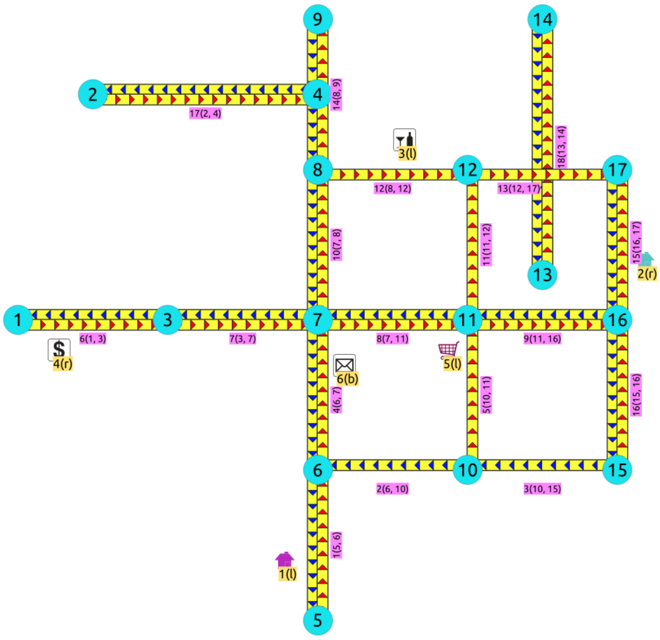
```
WITH
out_going AS (
  SELECT id AS vid, unnest(out_edges) AS eid, x, y
  FROM vertices_table
)
UPDATE edges
SET source = vid, x1 = x, y1 = y
FROM out_going WHERE id = eid;
UPDATE 18
```

Updating the `target` information

```
WITH
in_coming AS (
  SELECT id AS vid, unnest(in_edges) AS eid, x, y
  FROM vertices_table
)
UPDATE edges
SET target = vid, x2 = x, y2 = y
FROM in_coming WHERE id = eid;
UPDATE 18
```

Inspect the routing topology

```
SELECT id, source, target, x1, y1, x2, y2
FROM edges ORDER BY id;
 id | source | target | x1  | y1  |       x2       | y2
----+--------+--------+-----+-----+----------------+-----
  1 |      5 |      6 |   2 |   0 |              2 |   1
  2 |      6 |     10 |   2 |   1 |              3 |   1
  3 |     10 |     15 |   3 |   1 |              4 |   1
  4 |      6 |      7 |   2 |   1 |              2 |   2
  5 |     10 |     11 |   3 |   1 |              3 |   2
  6 |      1 |      3 |   0 |   2 |              1 |   2
  7 |      3 |      7 |   1 |   2 |              2 |   2
  8 |      7 |     11 |   2 |   2 |              3 |   2
  9 |     11 |     16 |   3 |   2 |              4 |   2
 10 |      7 |      8 |   2 |   2 |              2 |   3
 11 |     11 |     12 |   3 |   2 |              3 |   3
 12 |      8 |     12 |   2 |   3 |              3 |   3
 13 |     12 |     17 |   3 |   3 |              4 |   3
 14 |      8 |      9 |   2 |   3 |              2 |   4
 15 |     16 |     17 |   4 |   2 |              4 |   3
 16 |     15 |     16 |   4 |   1 |              4 |   2
 17 |      2 |      4 | 0.5 | 3.5 | 1.999999999999 | 3.5
 18 |     13 |     14 | 3.5 | 2.3 |            3.5 |   4
(18 rows)
```

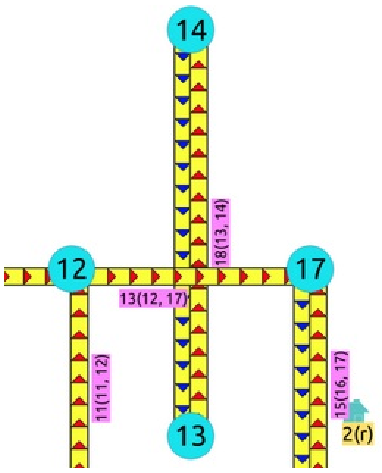


Generated topology

Crossing edges

To get the crossing edges:

```
SELECT a.id, b.id
FROM edges AS a, edges AS b
WHERE a.id < b.id AND st_crosses(a.geom, b.geom);
 id | id
----+----
 13 | 18
(1 row)
```



That information is correct, for example, when in terms of vehicles, is it a tunnel or bride crossing over another road.

It might be incorrect, for example:

1. When it is actually an intersection of roads, where vehicles can make turns.
2. When in terms of electrical lines, the electrical line is able to switch roads even on a tunnel or bridge.

When it is incorrect, it needs fixing:

1. For vehicles and pedestrians
   - If the data comes from OSM and was imported to the database using `osm2pgrouting`, the fix needs to be done in the **OSM portal** and the data imported again.
   - In general when the data comes from a supplier that has the data prepared for routing vehicles, and there is a problem, the data is to be fixed from the supplier
2. For very specific applications
   - The data is correct when from the point of view of routing vehicles or pedestrians.
   - The data needs a local fix for the specific application.

Once analyzed one by one the crossings, for the ones that need a local fix, the edges need to be **split**.

```
SELECT ST_AsText((ST_Dump(ST_Split(a.geom, b.geom))).geom)
FROM edges AS a, edges AS b
WHERE a.id = 13 AND b.id = 18
UNION
SELECT ST_AsText((ST_Dump(ST_Split(b.geom, a.geom))).geom)
FROM edges AS a, edges AS b
WHERE a.id = 13 AND b.id = 18;
        st_astext
---------------------------
 LINESTRING(3.5 2.3,3.5 3)
 LINESTRING(3 3,3.5 3)
 LINESTRING(3.5 3,4 3)
 LINESTRING(3.5 3,3.5 4)
(4 rows)
```

The new edges need to be added to the edges table, the rest of the attributes need to be updated in the new edges, the old edges need to be removed and the routing topology needs to be updated.

Adding split edges

For each pair of crossing edges a process similar to this one must be performed.

The columns inserted and the way are calculated are based on the application. For example, if the edges have a trait **name**, then that column is to be copied.

For pgRouting calculations

- **factor** based on the position of the intersection of the edges can be used to adjust the `cost` and `reverse_cost` columns.
- Capacity information, used on the **Flow - Family of functions** functions does not need to change when splitting edges.

```
WITH
first_edge AS (
  SELECT (ST_Dump(ST_Split(a.geom, b.geom))).path[1],
    (ST_Dump(ST_Split(a.geom, b.geom))).geom,
    ST_LineLocatePoint(a.geom,ST_Intersection(a.geom,b.geom)) AS factor
  FROM edges AS a, edges AS b
  WHERE a.id = 13 AND b.id = 18),
first_segments AS (
  SELECT path, first_edge.geom,
    capacity, reverse_capacity,
    CASE WHEN path=1 THEN factor * cost
         ELSE (1 - factor) * cost END AS cost,
    CASE WHEN path=1 THEN factor * reverse_cost
         ELSE (1 - factor) * reverse_cost END AS reverse_cost
  FROM first_edge , edges WHERE id = 13),
second_edge AS (
  SELECT (ST_Dump(ST_Split(b.geom, a.geom))).path[1],
    (ST_Dump(ST_Split(b.geom, a.geom))).geom,
    ST_LineLocatePoint(b.geom,ST_Intersection(a.geom,b.geom)) AS factor
  FROM edges AS a, edges AS b
  WHERE a.id = 13 AND b.id = 18),
second_segments AS (
  SELECT path, second_edge.geom,
    capacity, reverse_capacity,
    CASE WHEN path=1 THEN factor * cost
         ELSE (1 - factor) * cost END AS cost,
    CASE WHEN path=1 THEN factor * reverse_cost
         ELSE (1 - factor) * reverse_cost END AS reverse_cost
  FROM second_edge , edges WHERE id = 18),
all_segments AS (
  SELECT * FROM first_segments
  UNION
  SELECT * FROM second_segments)
INSERT INTO edges
  (capacity, reverse_capacity,
    cost, reverse_cost,
    x1, y1, x2, y2,
    geom)
(SELECT capacity, reverse_capacity, cost, reverse_cost,
  ST_X(ST_StartPoint(geom)), ST_Y(ST_StartPoint(geom)),
  ST_X(ST_EndPoint(geom)), ST_Y(ST_EndPoint(geom)),
  geom
  FROM all_segments);
INSERT 0 4
```

Adding new vertices

After adding all the split edges required by the application, the newly created vertices need to be added to the vertices table.

```
INSERT INTO vertices (in_edges, out_edges, x, y, geom)
(SELECT nv.in_edges, nv.out_edges, nv.x, nv.y, nv.geom
FROM pgr_extractVertices('SELECT id, geom FROM edges') AS nv
LEFT JOIN vertices AS v USING(geom) WHERE v.geom IS NULL);
INSERT 0 1
```

Updating edges topology

```
/* -- set the source information */
UPDATE edges AS e
SET source = v.id
FROM vertices AS v
WHERE source IS NULL AND ST_StartPoint(e.geom) = v.geom;
UPDATE 4
/* -- set the target information */
UPDATE edges AS e
SET target = v.id
FROM vertices AS v
WHERE target IS NULL AND ST_EndPoint(e.geom) = v.geom;
UPDATE 4
```

Removing the surplus edges

Once all significant information needed by the application has been transported to the new edges, then the crossing edges can be deleted.

```
DELETE FROM edges WHERE id IN (13, 18);
DELETE 2
```

There are other options to do this task, like creating a view, or a materialized view.

Updating vertices topology

To keep the graph consistent, the vertices topology needs to be updated

```
UPDATE vertices AS v SET
in_edges = nv.in_edges, out_edges = nv.out_edges
FROM (SELECT * FROM pgr_extractVertices('SELECT id, geom FROM edges')) AS nv
WHERE v.geom = nv.geom;
UPDATE 18
```

Checking for crossing edges

There are no crossing edges on the graph.

```
SELECT a.id, b.id
FROM edges AS a, edges AS b
WHERE a.id < b.id AND st_crosses(a.geom, b.geom);
 id | id
----+----
(0 rows)
```

Graphs without geometries

Using this table design for this example:

```
CREATE TABLE wiki (
  id SERIAL,
  source INTEGER,
  target INTEGER,
  cost INTEGER);
CREATE TABLE
```

Insert the data

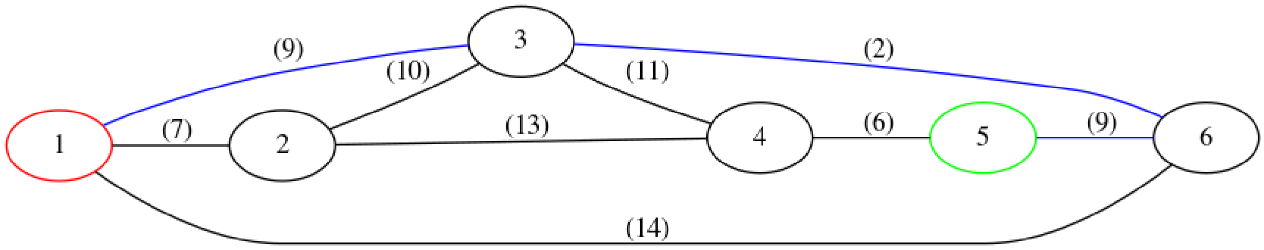
```
INSERT INTO wiki (source, target, cost) VALUES
(1, 2, 7),  (1, 3, 9), (1, 6, 14),
(2, 3, 10), (2, 4, 15),
(3, 6, 2),  (3, 4, 11),
(4, 5, 6),
(5, 6, 9);
INSERT 0 9
```

Find the shortest path

To solve this example **pgr_dijkstra** is used:

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM wiki',
  1, 5, false);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    1 |    2 |    9 |        0
   2 |        2 |    3 |    6 |    2 |        9
   3 |        3 |    6 |    9 |    9 |       11
   4 |        4 |    5 |   -1 |    0 |       20
(4 rows)
```

To go from \(1\) to \(5\) the path goes thru the following vertices:\(1 \rightarrow 3 \rightarrow 6 \rightarrow 5\)



Vertex information

To obtain the vertices information, use **pgr_extractVertices – Proposed**

```
SELECT id, in_edges, out_edges
FROM pgr_extractVertices('SELECT id, source, target FROM wiki');
 id | in_edges | out_edges
----+----------+-----------
  3 | {2,4}    | {6,7}
  5 | {8}      | {9}
  4 | {5,7}    | {8}
  2 | {1}      | {4,5}
  1 |          | {1,2,3}
  6 | {3,6,9}  |
(6 rows)
```

See Also

- **Topology - Family of Functions**
- **pgr_createVerticesTable**

Indices and tables

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

See Also

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

Traveling Sales Person - Family of functions

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_TSP

- pgr_TSP - Aproximation using *metric* algorithm.

Boost Graph Inside

Availability:

- Version 3.2.1
  - Metric Algorithm from **Boost library**
  - Simulated Annealing Algorithm no longer supported
    - The Simulated Annealing Algorithm related parameters are ignored: max_processing_time, tries_per_temperature, max_changes_per_temperature, max_consecutive_non_changes, initial_temperature, final_temperature, cooling_factor, randomize
- Version 2.3.0
  - Signature change
    - Old signature no longer supported
- Version 2.0.0
  - **Official** function

Description

Problem Definition

The travelling salesperson problem (TSP) 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?*

Characteristics

- This problem is an NP-hard optimization problem.
- Metric Algorithm is used
- Implementation generates solutions that *are twice as long as the optimal tour in the worst case* when:
  - Graph is undirected
  - Graph is fully connected
  - Graph where traveling costs on edges obey the triangle inequality.
- On an undirected graph:
  - The traveling costs are symmetric:
  - Traveling costs from u to v are just as much as traveling from v to u
- Can be Used with **Cost Matrix - Category** functions preferably with *directed => false*.
  - With directed => false
    - Will generate a graph that:
      - is undirected
      - is fully connected (As long as the graph has one component)
      - all traveling costs on edges obey the triangle inequality.
    - When start_vid = 0 OR end_vid = 0
      - The solutions generated is garanteed to be *twice as long as the optimal tour in the worst case*
    - When start_vid != 0 AND end_vid != 0 AND start_vid != end_vid
      - It is **not garanteed** that the solution will be, in the worse case, twice as long as the optimal tour, due to the fact that *end_vid* is forced to be in a fixed position.
  - With directed => true
    - It is **not garanteed** that the solution will be, in the worse case, twice as long as the optimal tour
    - Will generate a graph that:
      - is directed
      - is fully connected (As long as the graph has one component)
      - some (or all) traveling costs on edges might not obey the triangle inequality.
    - As an undirected graph is required, the directed graph is transformed as follows:
      - edges *(u, v)* and *(v, u)* is considered to be the same edge (denoted *(u, v)*
      - if agg_cost differs between one or more instances of edge *(u, v)*
      - The minimum value of the agg_cost all instances of edge *(u, v)* is going to be considered as the agg_cost of edge *(u, v)*
      - Some (or all) traveling costs on edges will still might not obey the triangle inequality.
- When the data is incomplete, but it is a connected graph:
  - the missing values will be calculated with dijkstra algorithm.

Signatures

Summary

```
pgr_TSP(Matrix SQL, [start_id, end_id])

RETURNS SET OF (seq, node, cost, agg_cost)
OR EMTPY SET
```

Example:
Using **pgr_dijkstraCostMatrix** to generate the matrix information

- **Line 4** Vertices \(\{2, 4, 13, 14\}\) are not included because they are not connected.

```
1 SELECT * FROM pgr_TSP(
2   $$SELECT * FROM pgr_dijkstraCostMatrix(
3     'SELECT id, source, target, cost, reverse_cost FROM edges',
4     (SELECT array_agg(id) FROM vertices WHERE id NOT IN (2, 4, 13, 14)),
5     directed => false) $$);
6  seq | node | cost | agg_cost
7 -----+------+------+----------
8    1 |    1 |    0 |        0
9    2 |    3 |    1 |        1
10   3 |    7 |    1 |        2
11   4 |    6 |    1 |        3
12   5 |    5 |    1 |        4
13   6 |   10 |    2 |        6
14   7 |   11 |    1 |        7
15   8 |   12 |    1 |        8
16   9 |   16 |    2 |       10
17  10 |   15 |    1 |       11
18  11 |   17 |    2 |       13
19  12 |    9 |    3 |       16
20  13 |    8 |    1 |       17
21  14 |    1 |    3 |       20
22 (14 rows)
23
```

Parameters

Parameter	Type	Description
Matrix SQL	TEXT	**Matrix SQL** as described below

TSP optional parameters

Column	Type	Default	Description
start_id	**ANY-INTEGER**	0	The first visiting vertex • When *0* any vertex can become the first visiting vertex.
end_id	**ANY-INTEGER**	0	Last visiting vertex before returning to start_vid. • When 0 any vertex can become the last visiting vertex before returning to start_id. • When NOT 0 and start_id = 0 then it is the first and last vertex

Inner Queries

Matrix SQL

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

Column	Type	Description
start_vid	ANY-INTEGER	Identifier of the starting vertex.
end_vid	ANY-INTEGER	Identifier of the ending vertex.
agg_cost	ANY-NUMERICAL	Cost for going from start_vid to end_vid

Result Columns

Returns SET OF (seq, node, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Row sequence.

Column	Type	Description
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 tour sequence.
agg_cost	FLOAT	Aggregate cost from the node at seq = 1 to the current node. • 0 for the first row in the tour sequence.

Additional Examples

- Start from vertex \(1\)
- Using points of interest to generate an asymetric matrix.
- Connected incomplete data

Start from vertex \(1\)

- **Line 6** start_vid => 1

```
1 SELECT * FROM pgr_TSP(
2   $$SELECT * FROM pgr_dijkstraCostMatrix(
3     'SELECT id, source, target, cost, reverse_cost FROM edges',
4     (SELECT array_agg(id) FROM vertices WHERE id NOT IN (2, 4, 13, 14)),
5     directed => false) $$,
6   start_id => 1);
7  seq | node | cost | agg_cost
8 -----+------+------+----------
9    1 |    1 |    0 |        0
10   2 |    3 |    1 |        1
11   3 |    7 |    1 |        2
12   4 |    6 |    1 |        3
13   5 |    5 |    1 |        4
14   6 |   10 |    2 |        6
15   7 |   11 |    1 |        7
16   8 |   12 |    1 |        8
17   9 |   16 |    2 |       10
18  10 |   15 |    1 |       11
19  11 |   17 |    2 |       13
20  12 |    9 |    3 |       16
21  13 |    8 |    1 |       17
22  14 |    1 |    3 |       20
23 (14 rows)
24
```

Using points of interest to generate an asymetric matrix.

To generate an asymmetric matrix:

- **Line 4** The side information of pointsOfInterset is ignored by not including it in the query
- **Line 6** Generating an asymetric matrix with directed => true
  - \(min(agg_cost(u, v), agg_cost(v, u))\) is going to be considered as the agg_cost
  - The solution that can be larger than *twice as long as the optimal tour* because:
    - Triangle inequality might not be satisfied.
    - start_id != 0 AND end_id != 0

```
1 SELECT * FROM pgr_TSP(
2   $$SELECT * FROM pgr_withPointsCostMatrix(
3     'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
4     'SELECT pid, edge_id, fraction from pointsOfInterest',
5     array[-1, 10, 7, 11, -6],
6     directed => true) $$,
7   start_id => 7,
8   end_id => 11);
9  seq | node | cost | agg_cost
10 -----+------+------+----------
11   1 |    7 |    0 |        0
12   2 |   -6 |  0.3 |      0.3
13   3 |   -1 |  1.3 |      1.6
14   4 |   10 |  1.6 |      3.2
15   5 |   11 |    1 |      4.2
16   6 |    7 |    1 |      5.2
17 (6 rows)
18
```

Connected incomplete data

Using selected edges \(\{2, 4, 5, 8, 9, 15\}\) the matrix is not complete.

```
SELECT * FROM pgr_dijkstraCostMatrix(
  $q1$SELECT id, source, target, cost, reverse_cost FROM edges WHERE id IN (2, 4, 5, 8, 9, 15)$q1$,
  (SELECT ARRAY[6, 7, 10, 11, 16, 17]),
  directed => true);
 start_vid | end_vid | agg_cost
-----------+---------+----------
         6 |       7 |        1
         6 |      11 |        2
         6 |      16 |        3
         6 |      17 |        4
         7 |       6 |        1
         7 |      11 |        1
         7 |      16 |        2
         7 |      17 |        3
        10 |       6 |        1
        10 |       7 |        2
        10 |      11 |        1
        10 |      16 |        2
        10 |      17 |        3
        11 |       6 |        2
        11 |       7 |        1
        11 |      16 |        1
        11 |      17 |        2
        16 |       6 |        3
        16 |       7 |        2
        16 |      11 |        1
        16 |      17 |        1
        17 |       6 |        4
        17 |       7 |        3
        17 |      11 |        2
        17 |      16 |        1
(25 rows)

```

Cost value for \(17 \rightarrow 10\) do not exist on the matrix, but the value used is taken from\(10 \rightarrow 17\).

```
SELECT * FROM pgr_TSP(
  $$SELECT * FROM pgr_dijkstraCostMatrix(
  $q1$SELECT id, source, target, cost, reverse_cost FROM edges WHERE id IN (2, 4, 5, 8, 9, 15)$q1$,
  (SELECT ARRAY[6, 7, 10, 11, 16, 17]),
  directed => true)$$);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    6 |    0 |        0
   2 |    7 |    1 |        1
   3 |   11 |    1 |        2
   4 |   16 |    1 |        3
   5 |   17 |    1 |        4
   6 |   10 |    3 |        7
   7 |    6 |    1 |        8
(7 rows)

```

See Also

- **Traveling Sales Person - Family of functions**
- **Sample Data**
- **Boost's metric appro's metric approximation**
- **Wikipedia: Traveling Salesman Problem**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.5 2.4 2.3**

pgr_TSPeuclidean

- pgr_TSPeuclidean - Aproximation using *metric* algorithm.

Boost Graph Inside

Availability:

- Version 3.2.1
  - Metric Algorithm from **Boost library**
  - Simulated Annealing Algorithm no longer supported
    - The Simulated Annealing Algorithm related parameters are ignored: *max_processing_time*, *tries_per_temperature*, *max_changes_per_temperature*, *max_consecutive_non_changes*, *initial_temperature*, *final_temperature*, *cooling_factor*, *randomize*
- Version 3.0.0
  - Name change from pgr_eucledianTSP
- Version 2.3.0
  - New **Official** function

Description

Problem Definition

The travelling salesperson problem (TSP) 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?*

Characteristics

- This problem is an NP-hard optimization problem.
- Metric Algorithm is used
- Implementation generates solutions that *are twice as long as the optimal tour in the worst case* when:
  - Graph is undirected
  - Graph is fully connected
  - Graph where traveling costs on edges obey the triangle inequality.
- On an undirected graph:
  - The traveling costs are symmetric:
  - Traveling costs from `u` to `v` are just as much as traveling from `v` to `u`
- Any duplicated identifier will be ignored. The coordinates that will be kept is arbitrarly.
  - The coordinates are quite similar for the same identifier, for example

    ```
    1, 3.5, 1
    1, 3.499999999999 0.9999999
    ```

  - The coordinates are quite different for the same identifier, for example

    ```
    2, 3.5, 1.0
    2, 3.6, 1.1
    ```

Signatures

Summary

```
pgr_TSPeuclidean(Coordinates SQL, [start_id, end_id])

RETURNS SET OF (seq, node, cost, agg_cost)
OR EMTPY SET
```

Example:
With default values

```
SELECT * FROM pgr_TSPeuclidean(
  $$
  SELECT id, st_X(geom) AS x, st_Y(geom)AS y  FROM vertices
  $$);
 seq | node |      cost      |    agg_cost
-----+------+----------------+---------------
   1 |    1 |              0 |             0
   2 |    6 |   2.2360679775 |  2.2360679775
   3 |    5 |              1 |  3.2360679775
   4 |   10 |  1.41421356237 | 4.65028153987
   5 |    7 |  1.41421356237 | 6.06449510225
   6 |    2 |  2.12132034356 | 8.18581544581
   7 |    9 |  1.58113883008 | 9.76695427589
   8 |    4 |            0.5 | 10.2669542759
   9 |   14 |  1.58113883009 |  11.848093106
  10 |   17 |  1.11803398875 | 12.9661270947
  11 |   16 |              1 | 13.9661270947
  12 |   15 |              1 | 14.9661270947
  13 |   11 |  1.41421356237 | 16.3803406571
  14 |   13 | 0.583095189485 | 16.9634358466
  15 |   12 | 0.860232526704 | 17.8236683733
  16 |    8 |              1 | 18.8236683733
  17 |    3 |  1.41421356237 | 20.2378819357
  18 |    1 |              1 | 21.2378819357
(18 rows)
```

Parameters

Parameter	Type	Description
Coordinates SQL	TEXT	**Coordinates SQL** as described below

TSP optional parameters

Column	Type	Default	Description
start_id	**ANY-INTEGER**	0	The first visiting vertex • When *0* any vertex can become the first visiting vertex.
end_id	**ANY-INTEGER**	0	Last visiting vertex before returning to start_vid. • When 0 any vertex can become the last visiting vertex before returning to start_id. • When NOT 0 and start_id = 0 then it is the first and last vertex

Inner Queries

Coordinates SQL

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

Column	Type	Description
id	ANY-INTEGER	Identifier of the starting vertex.
x	ANY-NUMERICAL	X value of the coordinate.
y	ANY-NUMERICAL	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 tour sequence.
agg_cost	FLOAT	Aggregate cost from the node at seq = 1 to the current node. • 0 for the first row in the tour sequence.

Additional Examples

- **Test 29 cities of Western Sahara**

Test 29 cities of Western Sahara

This example shows how to make performance tests using University of Waterloo's **example data** using the 29 cities of **Western Sahara dataset**

Creating a table for the data and storing the data

```
CREATE TABLE wi29 (id BIGINT, x FLOAT, y FLOAT, geom geometry);
INSERT INTO wi29 (id, x, y) VALUES
(1,20833.3333,17100.0000),
(2,20900.0000,17066.6667),
(3,21300.0000,13016.6667),
(4,21600.0000,14150.0000),
(5,21600.0000,14966.6667),
(6,21600.0000,16500.0000),
(7,22183.3333,13133.3333),
(8,22583.3333,14300.0000),
(9,22683.3333,12716.6667),
(10,23616.6667,15866.6667),
(11,23700.0000,15933.3333),
(12,23883.3333,14533.3333),
(13,24166.6667,13250.0000),
(14,25149.1667,12365.8333),
(15,26133.3333,14500.0000),
(16,26150.0000,10550.0000),
(17,26283.3333,12766.6667),
(18,26433.3333,13433.3333),
(19,26550.0000,13850.0000),
(20,26733.3333,11683.3333),
(21,27026.1111,13051.9444),
(22,27096.1111,13415.8333),
(23,27153.6111,13203.3333),
(24,27166.6667,9833.3333),
(25,27233.3333,10450.0000),
(26,27233.3333,11783.3333),
(27,27266.6667,10383.3333),
(28,27433.3333,12400.0000),
(29,27462.5000,12992.2222);
```

Adding a geometry (for visual purposes)

```
UPDATE wi29 SET geom = ST_makePoint(x,y);
```

Total tour cost

Getting a total cost of the tour, compare the value with the length of an optimal tour is 27603, given on the dataset

```
SELECT *
FROM pgr_TSPeuclidean($$SELECT * FROM wi29$$)
WHERE seq = 30;
 seq | node |     cost      |   agg_cost
-----+------+---------------+---------------
  30 |    1 | 2266.91173136 | 28777.4854127
(1 row)
```

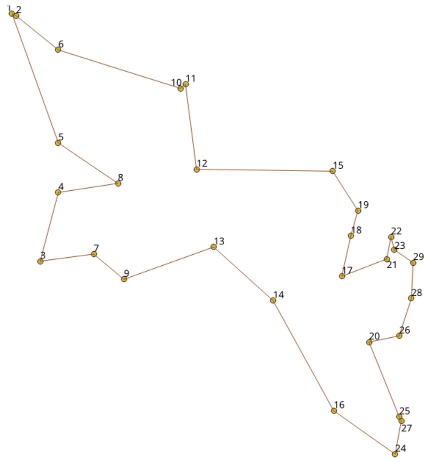
Getting a geometry of the tour

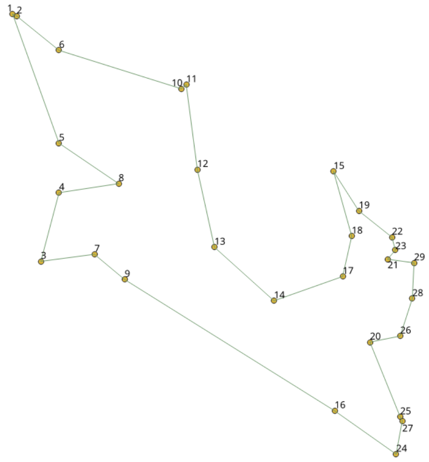
```
WITH
tsp_results AS (SELECT seq, geom FROM pgr_TSPeuclidean($$SELECT * FROM wi29$$) JOIN wi29 ON (node = id))
SELECT ST_MakeLine(ARRAY(SELECT geom FROM tsp_results ORDER BY seq));

-----------------------------------------------------------------------------------------------------------------------------------
 01020000001E000000F085C9545558D4400000000000B3D040000000000069D440107A36ABAAAAD040000000000018D54000000000001DD040107A36AB2A10D
(1 row)
```

Visual results

Visualy, The first image is the **optimal solution** and the second image is the solution obtained with `pgr_TSPeuclidean`.





See Also

- **Traveling Sales Person - Family of functions**
- **Sample Data** network.
- **Boost’s metric appro’s metric approximation**
- **University of Waterloo TSP**
- **Wikipedia: Traveling Salesman Problem**

Indices and tables

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

Table of Contents

General Information

Problem Definition

The travelling salesperson problem (TSP) 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)**

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 the travel costs do not depend on the direction taken around the tour:
  - this number by 2
  - \((n-1)!/2\).

Characteristics

- This problem is an NP-hard optimization problem.
- Metric Algorithm is used
- Implementation generates solutions that *are twice as long as the optimal tour in the worst case* when:
  - Graph is undirected
  - Graph is fully connected
  - Graph where traveling costs on edges obey the triangle inequality.
- On an undirected graph:
  - The traveling costs are symmetric:
  - Traveling costs from `u` to `v` are just as much as traveling from `v` to `u`

TSP optional parameters

Column	Type	Default	Description
`start_id`	**ANY-INTEGER**	`0`	The first visiting vertex • When *0* any vertex can become the first visiting vertex.
`end_id`	**ANY-INTEGER**	`0`	Last visiting vertex before returning to `start_vid`. • When `0` any vertex can become the last visiting vertex before returning to `start_id`. • When `NOT 0` and `start_id = 0` then it is the first and last vertex

See Also

References

- **Boost's metric appro's metric approximation**
- **University of Waterloo TSP**
- **Wikipedia: Traveling Salesman Problem**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

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.

Characteristics:

- 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.

See Also

- **Boost: Prim's algorithm**
- **Boost: Kruskal's algorithm**
- **Wikipedia: Prim's algorithm**
- **Wikipedia: Kruskal's algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**
- **Unsupported versions: 2.5 2.4 2.6**

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

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

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

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 edges',
        1, 17, false, false
    );
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   1 |   6 |    1
   1 |   3 |   7 |    1
   2 |   7 |   8 |    1
   3 |  11 |   9 |    1
   4 |  16 |  15 |    1
   5 |  17 |  -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 edges',
        6, 1, false, false,
        'SELECT to_cost, target_id::int4,
        from_edge || coalesce('','' || via_path, '''') AS via_path
        FROM restrictions'
    );
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   6 |   4 |    1
   1 |   7 |  10 |    1
   2 |   8 |  12 |    1
   3 |  12 |  11 |    1
   4 |  11 |   8 |    1
   5 |   7 |   7 |    1
   6 |   3 |   6 |    1
   7 |   1 |  -1 |    0
(8 rows)

SELECT * FROM pgr_trsp(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edges',
        1, 12, false, false,
        'SELECT to_cost, target_id::int4,
        from_edge || coalesce('','' || via_path, '''') AS via_path
        FROM restrictions'
    );
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   1 |   6 |    1
   1 |   3 |   7 |    1
   2 |   7 |   8 |    1
   3 |  11 |   9 |    1
   4 |  16 |  15 |    1
   5 |  17 |  13 |    1
   6 |  12 |  -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 edges',
        ARRAY[6,1,12]::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 |   6 |   4 |    1
   2 |   1 |   7 |  10 |    1
   3 |   1 |   8 |  12 |    1
   4 |   1 |  12 |  11 |    1
   5 |   1 |  11 |   8 |    1
   6 |   1 |   7 |   7 |    1
   7 |   1 |   3 |   6 |    1
   8 |   2 |   1 |   6 |    1
   9 |   2 |   3 |   7 |    1
  10 |   2 |   7 |   8 |    1
  11 |   2 |  11 |   9 |    1
  12 |   2 |  16 |  15 |    1
  13 |   2 |  17 |  13 |    1
  14 |   2 |  12 |  -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 edges',
    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 |   6 |   4 |    1
   3 |   1 |   7 |   8 |    1
   4 |   1 |  11 |   9 |    1
   5 |   1 |  16 |  16 |    1
   6 |   1 |  15 |   3 |    1
   7 |   1 |  10 |   5 |    1
   8 |   1 |  11 |   8 |    1
   9 |   1 |   7 |   7 |    1
  10 |   2 |   7 |   8 |    1
  11 |   2 |  11 |   9 |    1
  12 |   2 |  16 |  16 |    1
  13 |   2 |  15 |   3 |    1
  14 |   2 |  10 |   5 |    1
  15 |   2 |  11 |  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]);
```

1 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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    7, 4, 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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    7, 4
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
(0 rows)
```

Throwing EXCEPTION to represent no path found

```
SELECT * FROM pgr_trsp(
     $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
     7, 4, 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
CONTEXT:  PL/pgSQL function pgr_trsp(text,integer,integer,boolean,boolean,text) line 53 at RAISE
```

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.

1 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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
    7, 7,  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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
    2, 2, 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
CONTEXT:  PL/pgSQL function pgr_trsp(text,integer,integer,boolean,boolean,text) line 53 at RAISE
```

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(
   $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
   5, 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 |   5 |   1 |    1
   1 |   6 |   4 |    1
   2 |   7 |   8 |    1
   3 |  11 |   9 |    1
   4 |  16 |  16 |    1
   5 |  15 |   3 |    1
   6 |  10 |   2 |    1
   7 |   6 |   1 |    1
   8 |   5 |  -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.

1 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(
   $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edges$$,
   6, 10, 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
CONTEXT:  PL/pgSQL function pgr_trsp(text,integer,integer,boolean,boolean,text) line 24 at RAISE
```

An EXCEPTION is thrown.

When the reverse_cost exists, and the flag *has_rcost* is set to false

```
SELECT * FROM pgr_trsp(
   $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
   6, 10, false, false,
   $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   6 |   4 |    1
   1 |   7 |   8 |    1
   2 |  11 |   5 |    1
   3 |  10 |  -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]);
```

2 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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    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.

2 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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    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 \(-2\).

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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edges$$,
    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
CONTEXT:  PL/pgSQL function pgr_trsp(text,integer,double precision,integer,double precision,boolean,boolean,text) line 36 at RAISE
```

An EXCEPTION is thrown.

When the reverse_cost exists, and the flag *has_rcost* is set to false

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
    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 |                    geom                    |                  newpoint
-----+-----+-----+---------+------+----------+--------------------------------------------+--------------------------------------------
   1 | 1.8 | 0.4 |       1 | l    |      0.4 | 0101000000CDCCCCCCCCCCFC3F9A9999999999D93F | 010100000000000000000000409A9999999999D93F
   2 | 4.2 | 2.4 |      15 | r    |      0.4 | 0101000000CDCCCCCCCCCC10403333333333330340 | 010100000000000000000010403333333333330340
   3 | 2.6 | 3.2 |      12 | l    |      0.6 | 0101000000CDCCCCCCCCCC04409A99999999990940 | 0101000000CDCCCCCCCCCC04400000000000000840
   4 | 0.3 | 1.8 |       6 | r    |      0.3 | 0101000000333333333333D33FCDCCCCCCCCCCFC3F | 0101000000333333333333D33F0000000000000040
   5 | 2.9 | 1.8 |       5 | l    |      0.8 | 01010000003333333333330740CDCCCCCCCCCCFC3F | 01010000000000000000000840CDCCCCCCCCCCFC3F
   6 | 2.2 | 1.7 |       4 | b    |      0.7 | 01010000009A99999999990140333333333333FB3F | 01010000000000000000000040333333333333FB3F
(6 rows)
```

Using pgr_trsp

```
SELECT * FROM pgr_TRSP(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    (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 |   6 |   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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    $$SELECT pid, edge_id, fraction FROM pointsOfInterest$$,
    -1, -6
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   -1 |    1 |  0.6 |        0
   2 |        2 |    6 |    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(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
    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 |  11 |   8 |    1
   1 |   7 |   4 |    1
   2 |   6 |   1 |  0.6
(3 rows)
```

- Vertex 6 is on edge 8 at 1 fraction

```
SELECT * FROM pgr_trsp(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edges$$,
    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 |  11 |   8 |    1
   1 |   7 |   4 |    1
   2 |   6 |   1 |  0.6
(3 rows)
```

- Vertex 6 is also edge 11 at 0 fraction

Using pgr_withPoints - Proposed

```
SELECT * FROM pgr_withPoints(
    $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost  FROM edges$$,
    $$SELECT pid, edge_id, fraction FROM pointsOfInterest$$,
    11, -1
);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   11 |    8 |    1 |        0
   2 |        2 |    7 |    4 |    1 |        1
   3 |        3 |    6 |    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**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.5 2.4 2.6**

Cost - Category

- **pgr_aStarCost**
- **pgr_bdAstarCost**
- **pgr_dijkstraCost**
- **pgr_bdDijkstraCost**
- **pgr_dijkstraNearCost - Proposed**

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**

General Information

Characteristics

Each function works as part of the family it belongs to.

The main Characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path of each pair combination of nodes requested.
- 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 **aggregate cost** of \((u, v)\) is the same as for \((v, u)\).
- Any duplicated value in the start or end vertex identifiers are ignored.
- The returned values are ordered:
  - `start_vid` ascending
  - `end_vid` ascending

See Also

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4**

Cost Matrix - Category

- **pgr_aStarCostMatrix**
- **pgr_bdAstarCostMatrix**
- **pgr_bdDijkstraCostMatrix**
- **pgr_dijkstraCostMatrix**
- **pgr_bdDijkstraCostMatrix**

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**

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**.
  - Use 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.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The aggregate 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 aggregate cost in the non included values *(u, v)* is \(\infty\).
- Let be the case the values returned are stored in a table:
  - The unique index would be the pair: (start_vid, end_vid).
- Depending on the function and its parameters, the results can be symmetric.
  - The aggregate 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

Parameters

Used in:

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

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Used in:

- **pgr_withPointsCostMatrix - proposed**

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Optional parameters

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

Inner Queries

Edges SQL

Used in:

- **pgr_withPointsCostMatrix - proposed**

- **pgr_dijkstraCostMatrix**

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative**value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

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

Result Columns

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	Aggregate cost from start_vid to end_vid.

See Also

- **Traveling Sales Person - Family of functions**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**
- **Unsupported versions: 2.6 2.5 2.4**

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

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

pgr_alphaShape

pgr_alphaShape — Polygon part of an alpha shape.

Availability

- Version 3.0.0
  - Breaking change on signature
  - Old signature no longer supported
  - **Boost 1.54** & **Boost 1.55** are supported
  - **Boost 1.56+** is preferable
    - Boost Geometry is stable on Boost 1.56
- 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

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\(circumcenter\ radius < spoon_radius\)
- The alpha parameter is the **spoon radius**
- When the total number of points is less than 3, returns an EMPTY geometry

Signatures

Summary

```
pgr_alphaShape(geometry, [alpha])
```

```
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(geom)
     FROM vertices), 1.5));
 st_area
---------
    9.75
(1 row)
```

Parameters

Parameter	Type	Default	Description
geometry	`geometry`		Geometry with at least \(3\) points
`alpha`	`FLOAT`	0	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**

Calculate nodes that are within a distance.

- 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.
- Edge \((u, v)\) will not be included when:
  - The distance from the **root** to \(u\) > limit distance.
  - The distance from the **root** to \(v\) > limit distance.
  - No new nodes are created on the graph, so when is within the limit and is not within the limit, the edge is not included.

Parameters

Parameter	Type	Description
Edges SQL	`TEXT`	Edges SQL as described below.
Root vid	`BIGINT`	Identifier of the root vertex of the tree.
Root vids	`ARRAY[ANY-INTEGER]`	Array of identifiers of the root vertices. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.
distance	`FLOAT`	Upper limit for the inclusion of a node in the result.

Where:

ANY-INTEGER:
`SMALLINT`, `INTEGER`, `BIGINT`

ANY-NUMERIC:
`SMALLINT`, `INTEGER`, `BIGINT`, `REAL`, `FLOAT`

Inner Queries

Edges SQL

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)
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**

- **Supported versions: Latest (3.3)**

BFS - Category

- **pgr_kruskalBFS**
- **pgr_primBFS**

Traversal using breadth first search.

- 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.

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • 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. • 0 values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

BFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	9223372036854775807	Upper limit of the depth of the tree. • When negative throws an error.

Inner Queries

Edges SQL

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)
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)

Parameter	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.
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.

Where:

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

See Also

- **Boost: Prim's algorithm**
- **Boost: Kruskal's algorithm**
- **Wikipedia: Prim's algorithm**
- **Wikipedia: Kruskal's algorithm**

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

A* - 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**

Reference

- **pgr_version**
- **pgr_full_version**

Topology - Family of Functions

The following functions modify the database directly therefore the user must have special permissions given by the administrators to use them.

- **pgr_createTopology** - create a topology based on the geometry.
- **pgr_createVerticesTable** - 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_bdAstarCost**
- **pgr_dijkstraCost**
- **pgr_bdDijkstraCost**
- **pgr_dijkstraNearCost - Proposed**

Cost Matrix - Category

- **pgr_aStarCostMatrix**
- **pgr_bdAstarCostMatrix**
- **pgr_bdDijkstraCostMatrix**
- **pgr_dijkstraCostMatrix**
- **pgr_bdDijkstraCostMatrix**

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**

BFS - Category

- **pgr_kruskalBFS**
- **pgr_primBFS**

DFS - Category

- **pgr_kruskalDFS**
- **pgr_primDFS**

Available Functions but not official pgRouting functions

- **Proposed Functions**
- **Experimental Functions**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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.
- **pgr_dijkstraNear - Proposed** - Get the route to the nearest vertex.
- **pgr_dijkstraNearCost - Proposed** - Get the cost to the nearest vertex.

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.

Topology - Family of Functions

These proposed functions do not modify the database.

- **pgr_extractVertices – Proposed** - Extracts vertex information based on the edge table information.

Coloring - Family of functions

- **pgr_sequentialVertexColoring - Proposed** - Vertex coloring algorithm using greedy approach.

Traversal - Family of functions

- **pgr_depthFirstSearch - Proposed** - Depth first search traversal of the graph.

- **Supported versions: Latest** (**3.3**) **3.2**

Traversal - Family of functions

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_depthFirstSearch - Proposed** - Depth first search traversal of the 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_breadthFirstSearch - Experimental** - Breath first search traversal of the graph.
- **pgr_binaryBreadthFirstSearch - Experimental** - Breath first search traversal of the graph.

Aditionaly there are 2 categories under this family

- **BFS - Category**
- **DFS - Category**

- **Supported versions: Latest (3.3) 3.2**

pgr_depthFirstSearch - Proposed

pgr_depthFirstSearch — Returns a depth first search traversal of the graph. The graph can be directed or undirected.

Boost Graph Inside

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.

Availability

- Version 3.3.0
  - Promoted to **proposed** function
- Version 3.2.0
  - New **experimental** signatures:
    - pgr_depthFirstSearch (**Single Vertex**)
    - pgr_depthFirstSearch (**Multiple Vertices**)

Description

Depth First Search algorithm is a traversal algorithm which starts from a root vertex, goes as deep as possible, and backtracks once a vertex is reached with no adjacent vertices or with all visited adjacent vertices. The traversal continues until all the vertices reachable from the root vertex are visited.

The main Characteristics are:

- The implementation works for both **directed** and **undirected** graphs.
- Provides the Depth First Search traversal order from a root vertex or from a set of root vertices.
- An optional non-negative maximum depth parameter to limit the results up to a particular depth.

- For optimization purposes, any duplicated values in the *Root vids* are ignored.
- It does not produce the shortest path from a root vertex to a target vertex.
- The aggregate cost of traversal is not guaranteed to be minimal.
- The returned values are ordered in ascending order of *start_vid*.
- Depth First Search Running time: \(O(E + V)\)

Signatures

Summary

pgr_depthFirstSearch(**Edges SQL**, **root vid**, [**options**])
pgr_depthFirstSearch(**Edges SQL**, **root vids**, [**options**])
options: [directed, max_depth]

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Single vertex

pgr_depthFirstSearch(**Edges SQL**, **root vid**, [**options**])
options: [directed, max_depth]

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Example:
From root vertex \(6\) on a **directed** graph with edges in ascending order of `id`

```
SELECT * FROM pgr_depthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edges
  ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |    7 |    4 |    1 |        1
   4 |     2 |         6 |    3 |    7 |    1 |        2
   5 |     3 |         6 |    1 |    6 |    1 |        3
   6 |     2 |         6 |   11 |    8 |    1 |        2
   7 |     3 |         6 |   16 |    9 |    1 |        3
   8 |     4 |         6 |   17 |   15 |    1 |        4
   9 |     4 |         6 |   15 |   16 |    1 |        4
  10 |     5 |         6 |   10 |    3 |    1 |        5
  11 |     3 |         6 |   12 |   11 |    1 |        3
  12 |     2 |         6 |    8 |   10 |    1 |        2
  13 |     3 |         6 |    9 |   14 |    1 |        3
(13 rows)
```

Multiple vertices

pgr_depthFirstSearch(**Edges SQL**, **root vids**, [**options**])
options: [directed, max_depth]

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Example:
From root vertices \(\{12, 6\}\) on an **undirected** graph with **depth** \(<= 2\) and edges in ascending order of `id`

```
SELECT * FROM pgr_depthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edges
  ORDER BY id',
  ARRAY[12, 6], directed => false, max_depth => 2);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     2 |         6 |   15 |    3 |    1 |        2
   5 |     2 |         6 |   11 |    5 |    1 |        2
   6 |     1 |         6 |    7 |    4 |    1 |        1
   7 |     2 |         6 |    3 |    7 |    1 |        2
   8 |     2 |         6 |    8 |   10 |    1 |        2
   9 |     0 |        12 |   12 |   -1 |    0 |        0
  10 |     1 |        12 |   11 |   11 |    1 |        1
  11 |     2 |        12 |   10 |    5 |    1 |        2
  12 |     2 |        12 |    7 |    8 |    1 |        2
  13 |     2 |        12 |   16 |    9 |    1 |        2
  14 |     1 |        12 |    8 |   12 |    1 |        1
  15 |     2 |        12 |    9 |   14 |    1 |        2
  16 |     1 |        12 |   17 |   13 |    1 |        1
(16 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • 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. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

Optional parameters

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

DFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	\(9223372036854775807\)	Upper limit of the depth of the tree. • When negative throws an error.

Inner Queries

Edges SQL

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)
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, depth, start_vid, node, edge, cost, agg_cost)

Parameter	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.
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.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

ANY-NUMERIC:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT, NUMERIC

Additional Examples

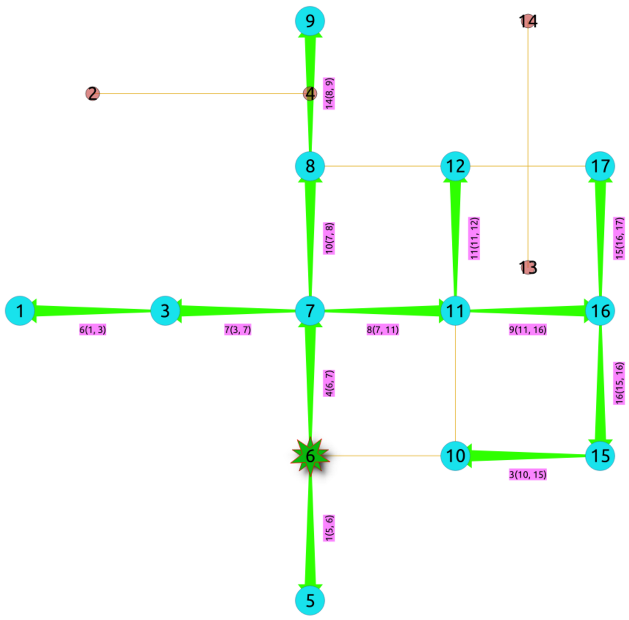
Example:

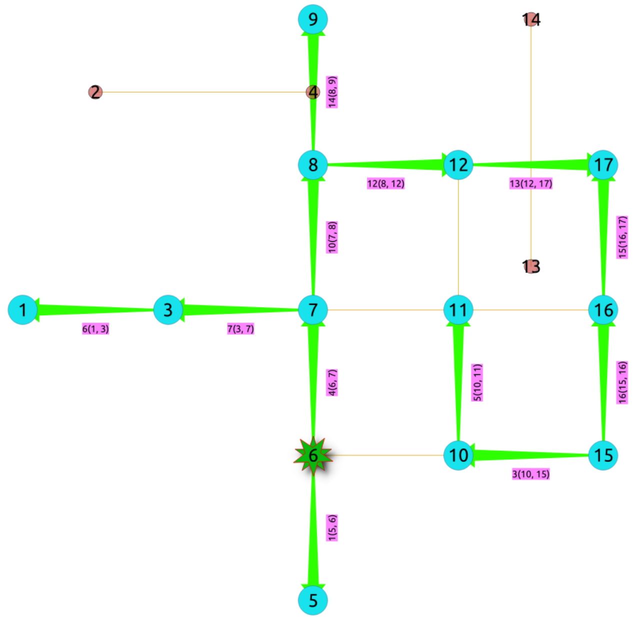
Same as **Single vertex** but with edges in descending order of id.

```
SELECT * FROM pgr_depthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edges
  ORDER BY id DESC',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    7 |    4 |    1 |        1
   3 |     2 |         6 |    8 |   10 |    1 |        2
   4 |     3 |         6 |    9 |   14 |    1 |        3
   5 |     3 |         6 |   12 |   12 |    1 |        3
   6 |     4 |         6 |   17 |   13 |    1 |        4
   7 |     5 |         6 |   16 |   15 |    1 |        5
   8 |     6 |         6 |   15 |   16 |    1 |        6
   9 |     7 |         6 |   10 |    3 |    1 |        7
  10 |     8 |         6 |   11 |    5 |    1 |        8
  11 |     2 |         6 |    3 |    7 |    1 |        2
  12 |     3 |         6 |    1 |    6 |    1 |        3
  13 |     1 |         6 |    5 |    1 |    1 |        1
(13 rows)
```

The resulting traversal is different.

The left image shows the result with ascending order of ids and the right image shows with descending order of the edge identifiers.





See Also

- **DFS - Category**
- **Sample Data**
- **Boost: Depth First Search algorithm documentation**
- **Boost: Undirected DFS algorithm documentation**
- **Wikipedia: Depth First Search algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

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

- Version 3.0.0
  - New **experimental** signature:
    - `pgr_breadthFirstSearch` (**Single Vertex**)
    - `pgr_breadthFirstSearch` (**Multiple Vertices**)

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.
- Running time: $O(E + V)$

Signatures

Summary

pgr_breadthFirstSearch(**Edges SQL**, **root vid**, [**options**])
pgr_breadthFirstSearch(**Edges SQL**, **root vids**, [**options**])
options: [max_depth, directed]

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Single vertex

pgr_breadthFirstSearch(**Edges SQL**, **root vid**, [**options**])
options: [max_depth, directed]

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Example:
From root vertex 6 on a **directed** graph with edges in ascending order of `id`

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |    7 |    4 |    1 |        1
   4 |     2 |         6 |    3 |    7 |    1 |        2
   5 |     2 |         6 |   11 |    8 |    1 |        2
   6 |     2 |         6 |    8 |   10 |    1 |        2
   7 |     3 |         6 |    1 |    6 |    1 |        3
   8 |     3 |         6 |   16 |    9 |    1 |        3
   9 |     3 |         6 |   12 |   11 |    1 |        3
  10 |     3 |         6 |    9 |   14 |    1 |        3
  11 |     4 |         6 |   17 |   15 |    1 |        4
  12 |     4 |         6 |   15 |   16 |    1 |        4
  13 |     5 |         6 |   10 |    3 |    1 |        5
(13 rows)
```

Multiple vertices

pgr_breadthFirstSearch(**Edges SQL**, **root vids**, [**options**])
options: [max_depth, directed]

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)

Example:
From root vertices \(\{12, 6\}\) on an **undirected** graph with **depth** \(<= 2\) and edges in ascending order of id

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges ORDER BY id',
  ARRAY[12, 6], directed => false, max_depth => 2);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |   10 |    2 |    1 |        1
   4 |     1 |         6 |    7 |    4 |    1 |        1
   5 |     2 |         6 |   15 |    3 |    1 |        2
   6 |     2 |         6 |   11 |    5 |    1 |        2
   7 |     2 |         6 |    3 |    7 |    1 |        2
   8 |     2 |         6 |    8 |   10 |    1 |        2
   9 |     0 |        12 |   12 |   -1 |    0 |        0
  10 |     1 |        12 |   11 |   11 |    1 |        1
  11 |     1 |        12 |    8 |   12 |    1 |        1
  12 |     1 |        12 |   17 |   13 |    1 |        1
  13 |     2 |        12 |   10 |    5 |    1 |        2
  14 |     2 |        12 |    7 |    8 |    1 |        2
  15 |     2 |        12 |   16 |    9 |    1 |        2
  16 |     2 |        12 |    9 |   14 |    1 |        2
(16 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.
root vid	BIGINT	Identifier of the root vertex of the tree. • 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. • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.

Where:

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

Optional parameters

Column	Type	Default	Description
directed	BOOLEAN	true	- When true the graph is considered *Directed* - When false the graph is considered as *Undirected*.

DFS optional parameters

Parameter	Type	Default	Description
max_depth	BIGINT	\(9223372036854775807\)	Upper limit of the depth of the tree. - When negative throws an error.

Inner Queries

Edges SQL

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)
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, depth, start_vid, node, edge, cost, agg_cost)

Parameter	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.
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.

Where:

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

Additional Examples

Example:
Same as **Single vertex** with edges in ascending order of id.

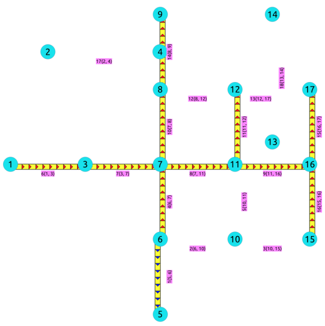
```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges ORDER BY id',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    5 |    1 |    1 |        1
   3 |     1 |         6 |    7 |    4 |    1 |        1
   4 |     2 |         6 |    3 |    7 |    1 |        2
   5 |     2 |         6 |   11 |    8 |    1 |        2
   6 |     2 |         6 |    8 |   10 |    1 |        2
   7 |     3 |         6 |    1 |    6 |    1 |        3
   8 |     3 |         6 |   16 |    9 |    1 |        3
   9 |     3 |         6 |   12 |   11 |    1 |        3
  10 |     3 |         6 |    9 |   14 |    1 |        3
  11 |     4 |         6 |   17 |   15 |    1 |        4
  12 |     4 |         6 |   15 |   16 |    1 |        4
  13 |     5 |         6 |   10 |    3 |    1 |        5
(13 rows)
```

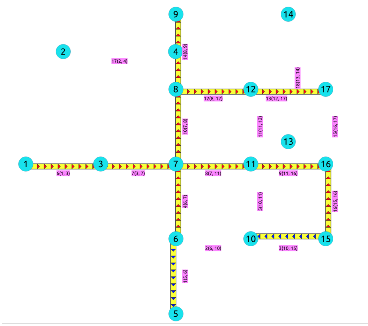
Example:
Same as **Single vertex** with edges in descending order of `id`.

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges ORDER BY id DESC',
  6);
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         6 |    6 |   -1 |    0 |        0
   2 |     1 |         6 |    7 |    4 |    1 |        1
   3 |     1 |         6 |    5 |    1 |    1 |        1
   4 |     2 |         6 |    8 |   10 |    1 |        2
   5 |     2 |         6 |   11 |    8 |    1 |        2
   6 |     2 |         6 |    3 |    7 |    1 |        2
   7 |     3 |         6 |    9 |   14 |    1 |        3
   8 |     3 |         6 |   12 |   12 |    1 |        3
   9 |     3 |         6 |   16 |    9 |    1 |        3
  10 |     3 |         6 |    1 |    6 |    1 |        3
  11 |     4 |         6 |   17 |   13 |    1 |        4
  12 |     4 |         6 |   15 |   16 |    1 |        4
  13 |     5 |         6 |   10 |    3 |    1 |        5
(13 rows)
```

The resulting traversal is different.

The left image shows the result with ascending order of ids and the right image shows with descending order of the edge identifiers.





See Also

- **BFS - Category**
- **Sample Data**
- **Boost: Breadth First Search algorithm documentation**
- **Wikipedia: Breadth First Search algorithm**

Indices and tables

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

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 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

- Version 3.2.0
  - New **experimental** signature:
    - pgr_binaryBreadthFirstSearch(**Combinations**)
- Version 3.0.0
  - New **experimental** signatures:
    - pgr_binaryBreadthFirstSearch(**One to One**)
    - pgr_binaryBreadthFirstSearch(**One to Many**)
    - pgr_binaryBreadthFirstSearch(**Many to One**)
    - pgr_binaryBreadthFirstSearch(**Many to Many**)

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 alledges 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

Summary

```
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])
pgr_binaryBreadthFirstSearch(Edges SQL, Combinations SQL, [directed])

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

Note: Using the **Sample Data** Network as all weights are same (i.e\(1`\))

One to One

```
pgr_binaryBreadthFirstSearch(Edges SQL, start vid, end vid, [directed])

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

Example:
From vertex \(6\) to vertex \(10\) on a **directed** graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost from edges',
  6, 10, true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 rows)
```

One to Many

```
pgr_binaryBreadthFirstSearch(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 \(6\) to vertices \(\{10, 17\}\) on a **directed** graph

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

Many to One

```
pgr_binaryBreadthFirstSearch(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 \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

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

Many to Many

```
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
```

Example:
From vertices \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

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

Combinations

```
pgr_binaryBreadthFirstSearch(Edges SQL, Combinations SQL, [directed])

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

Example:
Using a combinations table on an **undirected** graph

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    2 |    1 |        1
   5 |        3 |         5 |      10 |   10 |   -1 |    0 |        2
   6 |        1 |         6 |       5 |    6 |    1 |    1 |        0
   7 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
   8 |        1 |         6 |      15 |    6 |    2 |    1 |        0
   9 |        2 |         6 |      15 |   10 |    3 |    1 |        1
  10 |        3 |         6 |      15 |   15 |   -1 |    0 |        2
(10 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional Parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Result Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Path identifier. • Has value **1** for the first of a path from start_vid to end_vid.
path_seq	INTEGER	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** • **Combinations**
end_vid	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. • **One to Many** • **Many to Many** • **Combinations**
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_vid to node.

Additional Examples

Example:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

See Also

- **Sample Data**
- **https://cp-algorithms.com/graph/01_bfs.html**
- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Specialized_variants**

Indices and tables

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

See Also

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2**

Coloring - Family of functions

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_sequentialVertexColoring - Proposed** - Vertex coloring algorithm using greedy approach.

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_bipartite -Experimental** - Bipartite graph algorithm using a DFS-based coloring approach.
- **pgr_edgeColoring - Experimental** - Edge Coloring algorithm using Vizing’s theorem.

- **Supported versions: Latest (3.3) 3.2**

pgr_sequentialVertexColoring - Proposed

pgr_sequentialVertexColoring — Returns the vertex coloring of an undirected graph, using greedy approach.

Boost Graph Inside

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.

Availability

- Version 3.3.0
  - Promoted to **proposed** signature
- Version 3.2.0
  - New **experimental** signature

Description

Sequential vertex coloring algorithm is a graph coloring algorithm in which color identifiers are assigned to the vertices of a graph in a sequential manner, such that no edge connects two identically colored vertices.

The main Characteristics are:

- The implementation is applicable only for **undirected** graphs.
- Provides the color to be assigned to all the vertices present in the graph.
- Color identifiers values are in the Range $[1, |V|]$
- The algorithm tries to assign the least possible color to every vertex.
- Efficient graph coloring is an NP-Hard problem, and therefore, this algorithm does not always produce optimal coloring. It follows a greedy strategy by iterating through all the vertices sequentially, and assigning the smallest possible color that is not used by its neighbors, to each vertex.
- The returned rows are ordered in ascending order of the vertex value.
- Sequential Vertex Coloring Running Time: $O(|V|*(d + k))$
  - where $|V|$ is the number of vertices,
  - d is the maximum degree of the vertices in the graph,
  - k is the number of colors used.

Signatures

pgr_sequentialVertexColoring(**Edges SQL**)

RETURNS SET OF (vertex_id, color_id)
OR EMPTY SET

Example:
Graph coloring of pgRouting **Sample Data**

```
SELECT * FROM pgr_sequentialVertexColoring(
   'SELECT id, source, target, cost, reverse_cost FROM edges
   ORDER BY id'
);
 vertex_id | color_id
-----------+----------
         1 |        1
         2 |        1
         3 |        2
         4 |        2
         5 |        1
         6 |        2
         7 |        1
         8 |        2
         9 |        1
        10 |        1
        11 |        2
        12 |        1
        13 |        1
        14 |        2
        15 |        2
        16 |        1
        17 |        2
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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 (vertex_id, color_id)

Column	Type	Description
vertex_id	BIGINT	Identifier of the vertex.
color_id	BIGINT	Identifier of the color of the vertex. • The minimum value of color is 1.

See Also

- The queries use the **Sample Data** network.
- **Boost: Sequential Vertex Coloring algorithm documentation**
- **Wikipedia: Graph coloring**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2**

pgr_bipartite -Experimental

pgr_bipartite — Disjoint sets of vertices such that no two vertices within the same set are adjacent.

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 3.2.0
  - New **experimental** signature

Description

A bipartite graph is a graph with two sets of vertices which are connected to each other, but not within themselves. A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color.

The main Characteristics are:

- The algorithm works in undirected graph only.
- The returned values are not ordered.
- The algorithm checks graph is bipartite or not. If it is bipartite then it returns the node along with two colors *0* and *1* which represents two different sets.
- If graph is not bipartite then algorithm returns empty set.
- Running time: $O(V + E)$

Signatures

```
pgr_bipartite(Edges SQL)

RETURNS SET OF (vertex_id, color_id)
OR EMPTY SET
```

Example:
When the graph is bipartite

```
SELECT * FROM pgr_bipartite(
    $$SELECT id, source, target, cost, reverse_cost FROM edges$$
) ORDER BY vertex_id;
 vertex_id | color_id
-----------+----------
         1 |        0
         2 |        0
         3 |        1
         4 |        1
         5 |        0
         6 |        1
         7 |        0
         8 |        1
         9 |        0
        10 |        0
        11 |        1
        12 |        0
        13 |        0
        14 |        1
        15 |        1
        16 |        0
        17 |        1
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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 (vertex_id, color_id)

Column	Type	Description
vertex_id	BIGINT	Identifier of the vertex.
color_id	BIGINT	Identifier of the color of the vertex. • The minimum value of color is 1.

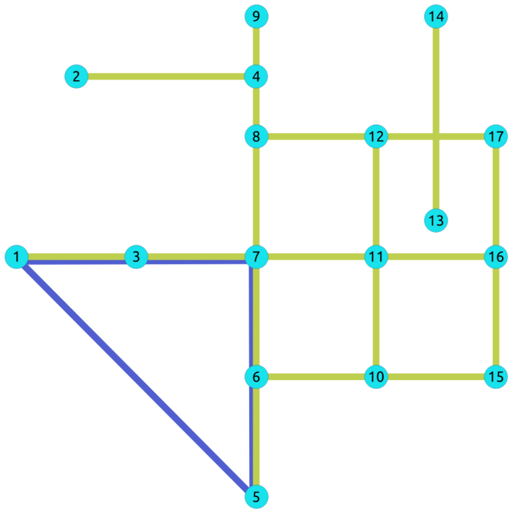
Additional Example

Example:
The odd length cyclic graph can not be bipartite.

The edge \(5 \rightarrow 1\) will make subgraph with vertices \(\{1, 3, 7, 6, 5\}\) an odd length cyclic graph, as the cycle has 5 vertices.

```
INSERT INTO edges (source, target, cost, reverse_cost) VALUES
(5, 1, 1, 1);
INSERT 0 1
```

Edges in blue represent odd length cycle subgraph.




```
SELECT * FROM pgr_bipartite(
   $$SELECT id, source, target, cost, reverse_cost FROM edges$$
);
 vertex_id | color_id
-----------+----------
(0 rows)
```

See Also

- **Boost: is_bipartite**
- **Wikipedia: bipartite graph**
- **Sample Data** network.

Indices and tables

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

- **Supported versions: Latest (3.3)**

pgr_edgeColoring - Experimental

pgr_edgeColoring — Returns the edge coloring of undirected and loop-free graphs

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 3.3.0
  - New **experimental** signature

Description

Edge Coloring is an algorithm used for coloring of the edges for the vertices in the graph. It is an assignment of colors to the edges of the graph so that no two adjacent edges have the same color.

The main Characteristics are:

- The implementation is for **undirected** and **loop-free** graphs
  - **loop free:**
    no self-loops and no parallel edges.
- Provides the color to be assigned to all the edges present in the graph.
- At most $\Delta + 1$ colors are used, where Δ is the degree of the graph.
  - This is optimal for some graphs, and by Vizing's theorem it uses at most one color more than the optimal for all others.
  - When the graph is bipartite
    - the chromatic number $x'(G)$ (minimum number of colors needed for proper edge coloring of graph) is equal to the degree $\Delta + 1$ of the graph, $x'(G) = \Delta$
- The algorithm tries to assign the least possible color to every edge.
  - Does not always produce optimal coloring.
- The returned rows are ordered in ascending order of the edge identifier.
- Efficient graph coloring is an NP-Hard problem, and therefore:
  - In this implelentation the running time: $O(|E|*|V|)$
    - where $|E|$ is the number of edges in the graph,
    - $|V|$ is the number of vertices in the graph.

Signatures

```
pgr_edgeColoring(Edges SQL)

RETURNS SET OF (edge_id, color_id)
OR EMPTY SET
```

Example:
Graph coloring of pgRouting **Sample Data**

```
SELECT * FROM pgr_edgeColoring(
    'SELECT id, source, target, cost, reverse_cost FROM edges
    ORDER BY id'
);
 edge_id | color_id
---------+----------
       1 |        3
       2 |        2
       3 |        3
       4 |        4
       5 |        4
       6 |        1
       7 |        2
       8 |        1
       9 |        2
      10 |        5
      11 |        5
      12 |        3
      13 |        2
      14 |        1
      15 |        3
      16 |        1
      17 |        1
      18 |        1
(18 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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_id, color_id)

Column	Type	Description
edge_id	BIGINT	Identifier of the edge.
color_id	BIGINT	Identifier of the color of the edge. • The minimum value of color is 1.

See Also

- The queries use the **Sample Data** network.
- **Boost: Edge Coloring Algorithm documentation**
- **Wikipedia: Graph Coloring**

Indices and tables

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

Result Columns

Returns SET OF (vertex_id, color_id)

Column	Type	Description
vertex_id	BIGINT	Identifier of the vertex.
color_id	BIGINT	Identifier of the color of the vertex. • The minimum value of color is 1.

Returns SET OF (edge_id, color_id)

Column	Type	Description
edge_id	BIGINT	Identifier of the edge.
color_id	BIGINT	Identifier of the color of the edge. • The minimum value of color is 1.

See Also

- **Boost: Sequential Vertex Coloring algorithm documentation**
- **Wikipedia: Graph coloring**

- **Boost: is_bipartite**
- **Wikipedia: bipartite graph**

- **Boost: Edge Coloring Algorithm documentation**
- **Wikipedia: Graph Coloring**

Indices and tables

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

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

Via - Category

- **pgr_dijkstraVia - Proposed**

withPoints - Category

- **withPoints - Family of functions** - Functions based on Dijkstra algorithm.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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.

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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 3.2.0
  - New **proposed** function:
    - pgr_withPoints(Combinations)
- Version 2.2.0
  - New **proposed** function

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**, **start vid**, **end vid**, [**options**])
pgr_withPoints(**Edges SQL**, **Points SQL**, **start vid**, **end vids**, [**options**])
pgr_withPoints(**Edges SQL**, **Points SQL**, **start vids**, **end vid**, [**options**])
pgr_withPoints(**Edges SQL**, **Points SQL**, **start vids**, **end vids**, [**options**])
pgr_withPoints(**Edges SQL**, **Points SQL**, **Combinations SQL**, [**options**])
options: [directed, driving_side, details])

RETURNS SET OF (seq, path_seq, [start_pid], [end_pid], node, edge, cost, agg_cost)
OR EMTPY SET

One to One

```
pgr_withPoints(Edges SQL, Points SQL, start vid, end vid, [options])
options: [directed, driving_side, details])

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

Example:
From point \(1\) to vertex \(10\) with details

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 10,
  details => true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   -1 |    1 |  0.6 |        0
   2 |        2 |    6 |    4 |  0.7 |      0.6
   3 |        3 |   -6 |    4 |  0.3 |      1.3
   4 |        4 |    7 |    8 |    1 |      1.6
   5 |        5 |   11 |    9 |    1 |      2.6
   6 |        6 |   16 |   16 |    1 |      3.6
   7 |        7 |   15 |    3 |    1 |      4.6
   8 |        8 |   10 |   -1 |    0 |      5.6
(8 rows)
```

One to Many

```
pgr_withPoints(Edges SQL, Points SQL, start vid, end vids, [options])
options: [directed, driving_side, details])

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

Example:
From point \(1\) to point \(3\) and vertex \(7\) on an undirected graph

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, ARRAY[-3, 7],
  directed => false);
 seq | path_seq | end_pid | node | edge | cost | agg_cost
-----+----------+---------+------+------+------+----------
   1 |        1 |      -3 |   -1 |    1 |  0.6 |        0
   2 |        2 |      -3 |    6 |    4 |    1 |      0.6
   3 |        3 |      -3 |    7 |   10 |    1 |      1.6
   4 |        4 |      -3 |    8 |   12 |  0.6 |      2.6
   5 |        5 |      -3 |   -3 |   -1 |    0 |      3.2
   6 |        1 |       7 |   -1 |    1 |  0.6 |        0
   7 |        2 |       7 |    6 |    4 |    1 |      0.6
   8 |        3 |       7 |    7 |   -1 |    0 |      1.6
(8 rows)
```

Many to One

```
pgr_withPoints(Edges SQL, Points SQL, start vids, end vid, [options])
options: [directed, driving_side, details])

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

Example:
From point \(1\) and vertex \(6\) to point \(3\)

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1, 6], -3);
 seq | path_seq | start_pid | node | edge | cost | agg_cost
-----+----------+-----------+------+------+------+----------
   1 |        1 |        -1 |   -1 |    1 |  0.6 |        0
   2 |        2 |        -1 |    6 |    4 |    1 |      0.6
   3 |        3 |        -1 |    7 |   10 |    1 |      1.6
   4 |        4 |        -1 |    8 |   12 |  0.6 |      2.6
   5 |        5 |        -1 |   -3 |   -1 |    0 |      3.2
   6 |        1 |         6 |    6 |    4 |    1 |        0
   7 |        2 |         6 |    7 |   10 |    1 |        1
   8 |        3 |         6 |    8 |   12 |  0.6 |        2
   9 |        4 |         6 |   -3 |   -1 |    0 |      2.6
(9 rows)
```

Many to Many

pgr_withPoints(**Edges SQL**, **Points SQL**, **start vids**, **end vids**, [**options**])
options: [directed, driving_side, details])

RETURNS SET OF (seq, path_seq, start_pid, end_pid, node, edge, cost, agg_cost)
OR EMTPY SET

Example:
From point \(1\) and vertex \(6\) to point \(3\) and vertex \(1\)

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1, 6], ARRAY[-3, 1]);
 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 |    6 |    4 |    1 |      0.6
   3 |        3 |        -1 |      -3 |    7 |   10 |    1 |      1.6
   4 |        4 |        -1 |      -3 |    8 |   12 |  0.6 |      2.6
   5 |        5 |        -1 |      -3 |   -3 |   -1 |    0 |      3.2
   6 |        1 |        -1 |       1 |   -1 |    1 |  0.6 |        0
   7 |        2 |        -1 |       1 |    6 |    4 |    1 |      0.6
   8 |        3 |        -1 |       1 |    7 |    7 |    1 |      1.6
   9 |        4 |        -1 |       1 |    3 |    6 |    1 |      2.6
  10 |        5 |        -1 |       1 |    1 |   -1 |    0 |      3.6
  11 |        1 |         6 |      -3 |    6 |    4 |    1 |        0
  12 |        2 |         6 |      -3 |    7 |   10 |    1 |        1
  13 |        3 |         6 |      -3 |    8 |   12 |  0.6 |        2
  14 |        4 |         6 |      -3 |   -3 |   -1 |    0 |      2.6
  15 |        1 |         6 |       1 |    6 |    4 |    1 |        0
  16 |        2 |         6 |       1 |    7 |    7 |    1 |        1
  17 |        3 |         6 |       1 |    3 |    6 |    1 |        2
  18 |        4 |         6 |       1 |    1 |   -1 |    0 |        3
(18 rows)
```

Combinations

pgr_withPoints(**Edges SQL**, **Points SQL**, **Combinations SQL**, [**options**])
options: [directed, driving_side, details])

RETURNS SET OF (seq, path_seq, start_pid, end_pid, node, edge, cost, agg_cost)
OR EMTPY SET

Example:
Two combinations

From point \(1\) to vertex \(10\), and from vertex \(6\) to point \(3\) with **right** side driving.

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  'SELECT * FROM (VALUES (-1, 10), (6, -3)) AS combinations(source, target)',
  driving_side => 'r', details => true);
 seq | path_seq | start_pid | end_pid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |        -1 |      10 |   -1 |    1 |  0.4 |        0
   2 |        2 |        -1 |      10 |    5 |    1 |    1 |      0.4
   3 |        3 |        -1 |      10 |    6 |    4 |  0.7 |      1.4
   4 |        4 |        -1 |      10 |   -6 |    4 |  0.3 |      2.1
   5 |        5 |        -1 |      10 |    7 |    8 |    1 |      2.4
   6 |        6 |        -1 |      10 |   11 |    9 |    1 |      3.4
   7 |        7 |        -1 |      10 |   16 |   16 |    1 |      4.4
   8 |        8 |        -1 |      10 |   15 |    3 |    1 |      5.4
   9 |        9 |        -1 |      10 |   10 |   -1 |    0 |      6.4
  10 |        1 |         6 |      -3 |    6 |    4 |  0.7 |        0
  11 |        2 |         6 |      -3 |   -6 |    4 |  0.3 |      0.7
  12 |        3 |         6 |      -3 |    7 |   10 |    1 |        1
  13 |        4 |         6 |      -3 |    8 |   12 |  0.6 |        2
  14 |        5 |         6 |      -3 |   -3 |   -1 |    0 |      2.6
(14 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** as described below
start vid	BIGINT	Identifier of the starting vertex of the path. Negative value is for point's identifier.
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices. Negative values are for point's identifiers.
end vid	BIGINT	Identifier of the ending vertex of the path. Negative value is for point's identifier.
end vids	ARRAY[BIGINT]	Array of identifiers of ending vertices. Negative values are for point's identifiers.

Optional parameters

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

With points optional parameters

Parameter	Type	Default	Description
driving_side	CHAR	b	Value in [r, l, b] indicating if the driving side is: • r for right driving side. • l for left driving side. • b for both.
details	BOOLEAN	false	• When true the results will include the points that are in the path. • When false the results will not include the points that are in the path.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative**value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

Result Columns

Returns set of (seq, path_seq [, start_pid] [, end_pid], node, edge, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	Relative position in the path. • **1** For the first row of the path.
start_pid	BIGINT	Identifier of a starting vertex/point of the path. • When positive is the identifier of the starting vertex. • When negative is the identifier of the starting point. • Returned on **Many to One** and **Many to Many**
end_pid	BIGINT	Identifier of an ending vertex/point of the path. • When positive is the identifier of the ending vertex. • When negative is the identifier of the ending point. • Returned on **One to Many** and **Many to Many**
node	BIGINT	Identifier of the node in the path fromstart_pid to end_pid. • When positive is the identifier of the a vertex. • When negative is the identifier of the a point.

Column	Type	Description
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. - **-1** for the last row of the path.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence. - **0** For the first row of the path.
agg_cost	FLOAT	Aggregate cost from start_vid to node. - **0** For the first row of the path.

Additional Examples

Usage variations

All the examples are about traveling from point \(1\) and vertex \(5\) to points \(\{2, 3, 6\}\) and vertices \(\{10, 11\}\)

```
SELECT *
FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[5, -1], ARRAY[-2, -3, -6, 10, 11],
  driving_side => 'r', details => true);
 seq | path_seq | start_pid | end_pid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |        -1 |      -6 |   -1 |    1 |  0.4 |        0
   2 |        2 |        -1 |      -6 |    5 |    1 |    1 |      0.4
   3 |        3 |        -1 |      -6 |    6 |    4 |  0.7 |      1.4
   4 |        4 |        -1 |      -6 |   -6 |   -1 |    0 |      2.1
   5 |        1 |        -1 |      -3 |   -1 |    1 |  0.4 |        0
   6 |        2 |        -1 |      -3 |    5 |    1 |    1 |      0.4
   7 |        3 |        -1 |      -3 |    6 |    4 |  0.7 |      1.4
   8 |        4 |        -1 |      -3 |   -6 |    4 |  0.3 |      2.1
   9 |        5 |        -1 |      -3 |    7 |   10 |    1 |      2.4
  10 |        6 |        -1 |      -3 |    8 |   12 |  0.6 |      3.4
  11 |        7 |        -1 |      -3 |   -3 |   -1 |    0 |        4
  12 |        1 |        -1 |      -2 |   -1 |    1 |  0.4 |        0
  13 |        2 |        -1 |      -2 |    5 |    1 |    1 |      0.4
  14 |        3 |        -1 |      -2 |    6 |    4 |  0.7 |      1.4
  15 |        4 |        -1 |      -2 |   -6 |    4 |  0.3 |      2.1
  16 |        5 |        -1 |      -2 |    7 |    8 |    1 |      2.4
  17 |        6 |        -1 |      -2 |   11 |    9 |    1 |      3.4
  18 |        7 |        -1 |      -2 |   16 |   15 |  0.4 |      4.4
  19 |        8 |        -1 |      -2 |   -2 |   -1 |    0 |      4.8
  20 |        1 |        -1 |      10 |   -1 |    1 |  0.4 |        0
  21 |        2 |        -1 |      10 |    5 |    1 |    1 |      0.4
  22 |        3 |        -1 |      10 |    6 |    4 |  0.7 |      1.4
  23 |        4 |        -1 |      10 |   -6 |    4 |  0.3 |      2.1
  24 |        5 |        -1 |      10 |    7 |    8 |    1 |      2.4
  25 |        6 |        -1 |      10 |   11 |    9 |    1 |      3.4
  26 |        7 |        -1 |      10 |   16 |   16 |    1 |      4.4
  27 |        8 |        -1 |      10 |   15 |    3 |    1 |      5.4
  28 |        9 |        -1 |      10 |   10 |   -1 |    0 |      6.4
  29 |        1 |        -1 |      11 |   -1 |    1 |  0.4 |        0
  30 |        2 |        -1 |      11 |    5 |    1 |    1 |      0.4
  31 |        3 |        -1 |      11 |    6 |    4 |  0.7 |      1.4
  32 |        4 |        -1 |      11 |   -6 |    4 |  0.3 |      2.1
  33 |        5 |        -1 |      11 |    7 |    8 |    1 |      2.4
  34 |        6 |        -1 |      11 |   11 |   -1 |    0 |      3.4
  35 |        1 |         5 |      -6 |    5 |    1 |    1 |        0
  36 |        2 |         5 |      -6 |    6 |    4 |  0.7 |        1
  37 |        3 |         5 |      -6 |   -6 |   -1 |    0 |      1.7
  38 |        1 |         5 |      -3 |    5 |    1 |    1 |        0
  39 |        2 |         5 |      -3 |    6 |    4 |  0.7 |        1
  40 |        3 |         5 |      -3 |   -6 |    4 |  0.3 |      1.7
  41 |        4 |         5 |      -3 |    7 |   10 |    1 |        2
  42 |        5 |         5 |      -3 |    8 |   12 |  0.6 |        3
  43 |        6 |         5 |      -3 |   -3 |   -1 |    0 |      3.6
  44 |        1 |         5 |      -2 |    5 |    1 |    1 |        0
  45 |        2 |         5 |      -2 |    6 |    4 |  0.7 |        1
  46 |        3 |         5 |      -2 |   -6 |    4 |  0.3 |      1.7
  47 |        4 |         5 |      -2 |    7 |    8 |    1 |        2
  48 |        5 |         5 |      -2 |   11 |    9 |    1 |        3
  49 |        6 |         5 |      -2 |   16 |   15 |  0.4 |        4
  50 |        7 |         5 |      -2 |   -2 |   -1 |    0 |      4.4
  51 |        1 |         5 |      10 |    5 |    1 |    1 |        0
  52 |        2 |         5 |      10 |    6 |    4 |  0.7 |        1
  53 |        3 |         5 |      10 |   -6 |    4 |  0.3 |      1.7
  54 |        4 |         5 |      10 |    7 |    8 |    1 |        2
  55 |        5 |         5 |      10 |   11 |    9 |    1 |        3
  56 |        6 |         5 |      10 |   16 |   16 |    1 |        4
  57 |        7 |         5 |      10 |   15 |    3 |    1 |        5
  58 |        8 |         5 |      10 |   10 |   -1 |    0 |        6
  59 |        1 |         5 |      11 |    5 |    1 |    1 |        0
  60 |        2 |         5 |      11 |    6 |    4 |  0.7 |        1
  61 |        3 |         5 |      11 |   -6 |    4 |  0.3 |      1.7
  62 |        4 |         5 |      11 |    7 |    8 |    1 |        2
  63 |        5 |         5 |      11 |   11 |   -1 |    0 |        3
(63 rows)
```

Passes in front or visits with right side driving.

For point \(6\) and vertex \(11\).

```
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 edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[5, -1], ARRAY[-2, -3, -6, 10, 11],
  driving_side => 'r', details => true)
WHERE node IN (-6, 11);
      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 | 11
 -1 -> 10 at 4th step |  passes in front of | Point  |  6
 -1 -> 10 at 6th step |  passes in front of | Vertex | 11
 -1 -> 11 at 4th step |  passes in front of | Point  |  6
 -1 -> 11 at 6th step |  visits             | Vertex | 11
 5 -> -6 at 3th step  |  visits             | Point  |  6
 5 -> -3 at 3th step  |  passes in front of | Point  |  6
 5 -> -2 at 3th step  |  passes in front of | Point  |  6
 5 -> -2 at 5th step  |  passes in front of | Vertex | 11
 5 -> 10 at 3th step  |  passes in front of | Point  |  6
 5 -> 10 at 5th step  |  passes in front of | Vertex | 11
 5 -> 11 at 3th step  |  passes in front of | Point  |  6
 5 -> 11 at 5th step  |  visits             | Vertex | 11
(16 rows)
```

Passes in front or visits with left side driving.

For point \(6\) and vertex \(11\).

```
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 edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[5, -1], ARRAY[-2, -3, -6, 10, 11],
  driving_side => 'l', details => true)
WHERE node IN (-6, 11);
      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 | 11
 -1 => 10 at 3th step |  passes in front of | Point  |  6
 -1 => 10 at 5th step |  passes in front of | Vertex | 11
 -1 => 11 at 3th step |  passes in front of | Point  |  6
 -1 => 11 at 5th step |  visits             | Vertex | 11
 5 => -6 at 4th step  |  visits             | Point  |  6
 5 => -3 at 4th step  |  passes in front of | Point  |  6
 5 => -2 at 4th step  |  passes in front of | Point  |  6
 5 => -2 at 6th step  |  passes in front of | Vertex | 11
 5 => 10 at 4th step  |  passes in front of | Point  |  6
 5 => 10 at 6th step  |  passes in front of | Vertex | 11
 5 => 11 at 4th step  |  passes in front of | Point  |  6
 5 => 11 at 6th step  |  visits             | Vertex | 11
(16 rows)
```

See Also

- **withPoints - Family of functions**
- **withPoints - Category**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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 3.2.0
  - New **proposed** function:
    - pgr_withPointsCost(Combinations)
- Version 2.2.0
  - New **proposed** function

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 ∞
- 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`_, **start vid**, **end vid**, **[options]**)
pgr_withPointsCost(**Edges SQL**, 'Points SQL`_, **start vid**, **end vids**, **[options]**)
pgr_withPointsCost(**Edges SQL**, 'Points SQL`_, **start vids**, **end vid**, **[options]**)
pgr_withPointsCost(**Edges SQL**, 'Points SQL`_, **start vids**, **end vids**, **[options]**)
pgr_withPointsCost(**Edges SQL**, 'Points SQL`_, **Combinations SQL**, **[options]**)
options: [directed, driving_side]

RETURNS SET OF (start_pid, end_pid, agg_cost)
OR EMPTY SET

Note

There is no **details** flag, unlike the other members of the withPoints family of functions.

One to One

pgr_withPointsCost(**Edges SQL**, 'Points SQL`_, **start vid**, **end vid**, **[options]**)
options: [directed, driving_side]

RETURNS SET OF (start_pid, end_pid, agg_cost)
OR EMPTY SET

Example:
From point \(1\) to vertex \(10\) with defaults

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 10);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      10 |      5.6
(1 row)
```

One to Many

pgr_withPointsCost(**Edges SQL**, **Points SQL**, **start vid**, **end vids**, **[options]**)
options: [directed, driving_side]

RETURNS SET OF (start_pid, end_pid, agg_cost)
OR EMPTY SET

Example:
From point \(1\) to point \(3\) and vertex \(7\) on an undirected graph

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, ARRAY[-3, 7],
  directed => false);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
        -1 |       7 |      1.6
(2 rows)
```

Many to One

pgr_withPointsCost(**Edges SQL**, **Points SQL**, **start vids**, **end vid**, **[options]**)
options: [directed, driving_side]

RETURNS SET OF (start_pid, end_pid, agg_cost)
OR EMPTY SET

Example:
From point \(1\) and vertex \(6\) to point \(3\)

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1, 6], -3);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
         6 |      -3 |      2.6
(2 rows)
```

Many to Many

pgr_withPointsCost(**Edges SQL**, **Points SQL**, **start vids**, **end vids**, **[options]**)
options: [directed, driving_side]

RETURNS SET OF (start_pid, end_pid, agg_cost)
OR EMPTY SET

Example:
From point \(15\) and vertex \(6\) to point \(3\) and vertex \(1\)

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1, 6], ARRAY[-3, 1]);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -3 |      3.2
        -1 |       1 |      3.6
         6 |      -3 |      2.6
         6 |       1 |        3
(4 rows)
```

Combinations

pgr_withPointsCost(**Edges SQL**, **Points SQL**, **Combinations SQL**, **[options]**)
options: [directed, driving_side]

RETURNS SET OF (start_pid, end_pid, agg_cost)
OR EMPTY SET

Example:
Two combinations

From point \(1\) to vertex \(10\), and from vertex \(6\) to point \(3\) with **right** side driving.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  'SELECT * FROM (VALUES (-1, 10), (6, -3)) AS combinations(source, target)',
  driving_side => 'r');
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      10 |      6.4
         6 |      -3 |      2.6
(2 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** as described below
start vid	BIGINT	Identifier of the starting vertex of the path. Negative value is for point's identifier.
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices. Negative values are for point's identifiers.
end vid	BIGINT	Identifier of the ending vertex of the path. Negative value is for point's identifier.

Column	Type	Description
end vids	ARRAY[BIGINT]	Array of identifiers of ending vertices. Negative values are for point's identifiers.

Optional parameters

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

With points optional parameters

Parameter	Type	Default	Description
driving_side	CHAR	b	Value in [r, l, b] indicating if the driving side is: • r for right driving side. • l for left driving side. • b for both.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative **value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Combinations SQL

Parameter	Type	Description
source	ANY-INTEGER	Identifier of the departure vertex.
target	ANY-INTEGER	Identifier of the arrival vertex.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

Result Columns

Column	Type	Description
start_pid	BIGINT	Identifier of the starting vertex or point. • When positive: is a vertex's identifier. • When negative: is a point's identifier.
end_pid	BIGINT	Identifier of the ending vertex or point. • When positive: is a vertex's identifier. • When negative: is a point's identifier.
agg_cost	FLOAT	Aggregate cost from start_vid to end_vid.

Additional Examples

- **Right side driving topology**
- **Left side driving topology**
- **Does not matter driving side driving topology**

Right side driving topology

Traveling from point \(1\) and vertex \(5\) to points \(\{2, 3, 6\}\) and vertices \(\{10, 11\}\)

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[5, -1], ARRAY[-2, -3, -6, 10, 11],
  driving_side => 'r');
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -6 |      2.1
        -1 |      -3 |        4
        -1 |      -2 |      4.8
        -1 |      10 |      6.4
        -1 |      11 |      3.4
         5 |      -6 |      1.7
         5 |      -3 |      3.6
         5 |      -2 |      4.4
         5 |      10 |        6
         5 |      11 |        3
(10 rows)
```

Left side driving topology

Traveling from point \(1\) and vertex \(5\) to points \(\{2, 3, 6\}\) and vertices \(\{10, 11\}\)

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[5, -1], ARRAY[-2, -3, -6, 10, 11],
  driving_side => 'l');
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -6 |      1.3
        -1 |      -3 |      3.2
        -1 |      -2 |      5.2
        -1 |      10 |      5.6
        -1 |      11 |      2.6
         5 |      -6 |      1.7
         5 |      -3 |      3.6
         5 |      -2 |      5.6
         5 |      10 |        6
         5 |      11 |        3
(10 rows)
```

Does not matter driving side driving topology

Traveling from point \(1\) and vertex \(5\) to points \(\{2, 3, 6\}\) and vertices \(\{10, 11\}\)

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[5, -1], ARRAY[-2, -3, -6, 10, 11]);
 start_pid | end_pid | agg_cost
-----------+---------+----------
        -1 |      -6 |      1.3
        -1 |      -3 |      3.2
        -1 |      -2 |        4
        -1 |      10 |      5.6
        -1 |      11 |      2.6
         5 |      -6 |      1.7
         5 |      -3 |      3.6
         5 |      -2 |      4.4
         5 |      10 |        6
         5 |      11 |        3
(10 rows)
```

The queries use the **Sample Data** network.

See Also

- **withPoints - Family of functions**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3**

pgr_withPointsCostMatrix - proposed

pgr_withPointsCostMatrix - Calculates a cost matrix using **pgr_withPoints - 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.

Boost Graph Inside

Availability

- Version 2.2.0
  - New **proposed** function

Description

Using Dijkstra algorithm, calculate and return a cost matrix.

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 to an ending vertex. This implementation can be used with a directed graph and an undirected graph.

The main Characteristics are:

- Can be used as input to **pgr_TSP**.
  - Use directly when the resulting matrix is symmetric and there is no ∞ 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 ∞ 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.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The aggregate 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 aggregate cost in the non included values *(u, v)* is ∞.
- Let be the case the values returned are stored in a table:
  - The unique index would be the pair: `(start_vid, end_vid)`.
- Depending on the function and its parameters, the results can be symmetric.
  - The aggregate 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

Signatures

Summary

pgr_withPointsCostMatrix(**Edges SQL**, **Points SQL**, **start vids**, [**options**])
options: [directed, driving_side]

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

Note

There is no **details** flag, unlike the other members of the withPoints family of functions.

Example:
Cost matrix for points $\{1, 6\}$ and vertices $\{10, 11\}$ 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 edges ORDER BY id',
  'SELECT pid, edge_id, fraction from pointsOfInterest',
  array[-1, 10, 11, -6], directed := false);
 start_vid | end_vid | agg_cost
-----------+---------+----------
        -6 |      -1 |      1.3
        -6 |      10 |      1.7
        -6 |      11 |      1.3
        -1 |      -6 |      1.3
        -1 |      10 |      1.6
        -1 |      11 |      2.6
        10 |      -6 |      1.7
        10 |      -1 |      1.6
        10 |      11 |        1
        11 |      -6 |      1.3
        11 |      -1 |      2.6
        11 |      10 |        1
(12 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices.

Optional parameters

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

With points optional parameters

Parameter	Type	Default	Description
driving_side	CHAR	b	Value in [r, l, b] indicating if the driving side is: • r for right driving side. • l for left driving side. • b for both.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative**value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Result Columns

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	Aggregate cost from start_vid to end_vid.

Note

When start_vid or end_vid columns have negative values, the identifier is for a Point.

Additional Examples

- **Use with pgr_TSP**.

Use with pgr_TSP.

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_withPointsCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
    'SELECT pid, edge_id, fraction from pointsOfInterest',
    array[-1, 10, 11, -6], directed := false);
  $$
);
NOTICE:  pgr_TSP no longer solving with simulated annaeling
HINT:  Ignoring annaeling parameters
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |   -6 |    0 |        0
   2 |   -1 |  1.3 |      1.3
   3 |   10 |  1.6 |      2.9
   4 |   11 |    1 |      3.9
   5 |   -6 |  1.3 |      5.2
(5 rows)
```

See Also

- **withPoints - Family of functions**
- **Cost Matrix - Category**
- **Traveling Sales Person - Family of functions**
- **Sample Data**

Indices and tables

pgr_withPointsKSP - Proposed

`pgr_withPointsKSP` — Yen's algorithm for K shortest paths using Dijkstra.

> **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

Description

Modifies the graph to include the points defined in the **Points SQL** and using Yen algorithm, finds the K shortest paths.

Signatures

```
pgr_withPointsKSP(Edges SQL, Points SQL start vid, end vid, K, [options])
options: [directed, heap_paths, driving_side, details]

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

Example:
Get 2 paths from Point 1 to point 2 on a directed graph.

- 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 edges 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 |    6 |    4 |    1 |      0.6
   3 |       1 |        3 |    7 |    8 |    1 |      1.6
   4 |       1 |        4 |   11 |    9 |    1 |      2.6
   5 |       1 |        5 |   16 |   15 |  0.4 |      3.6
   6 |       1 |        6 |   -2 |   -1 |    0 |        4
   7 |       2 |        1 |   -1 |    1 |  0.6 |        0
   8 |       2 |        2 |    6 |    4 |    1 |      0.6
   9 |       2 |        3 |    7 |    8 |    1 |      1.6
  10 |       2 |        4 |   11 |   11 |    1 |      2.6
  11 |       2 |        5 |   12 |   13 |    1 |      3.6
  12 |       2 |        6 |   17 |   15 |  0.6 |      4.6
  13 |       2 |        7 |   -2 |   -1 |    0 |      5.2
(13 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** query as described.
Points SQL	TEXT	**Points SQL** query as described.
start vid	**ANY-INTEGER**	Identifier of the departure vertex. • Negative values represent a point
end vid	**ANY-INTEGER**	Identifier of the destination vertex. • Negative values represent a point
K	**ANY-INTEGER**	Number of required paths

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Optional parameters

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

KSP Optional parameters

Column	Type	Default	Description
heap_paths	BOOLEAN	false	• When false Returns at most K paths • When true all the calculated paths while processing are returned. • Roughly, when the shortest path has N edges, the heap will contain about than N * K paths for small value of K and K > 5.

With points optional parameters

Parameter	Type	Default	Description
driving_side	CHAR	b	Value in [r, l, b] indicating if the driving side is: • r for right driving side. • l for left driving side. • b for both.
details	BOOLEAN	false	• When true the results will include the points that are in the path. • When false the results will not include the points that are in the path.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative**value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

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	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Path identifier. • Has value **1** for the first of a path from**start vid** to **end_vid**
path_seq	INTEGER	Relative position in the path. Has value**1** for the beginning of a 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 fromnode 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. • \(0\) for the last node of the path.
agg_cost	FLOAT	Aggregate cost from **start vid** to node.

Additional Examples

- **Left driving side**
- **Right driving side**

Left driving side

Get \(2\) paths using left side driving topology, from point\(1\) to point \(2\) with details.

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edges 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 |    6 |    4 |  0.7 |      0.6
   3 |       1 |        3 |   -6 |    4 |  0.3 |      1.3
   4 |       1 |        4 |    7 |    8 |    1 |      1.6
   5 |       1 |        5 |   11 |   11 |    1 |      2.6
   6 |       1 |        6 |   12 |   13 |    1 |      3.6
   7 |       1 |        7 |   17 |   15 |  0.6 |      4.6
   8 |       1 |        8 |   -2 |   -1 |    0 |      5.2
   9 |       2 |        1 |   -1 |    1 |  0.6 |        0
  10 |       2 |        2 |    6 |    4 |  0.7 |      0.6
  11 |       2 |        3 |   -6 |    4 |  0.3 |      1.3
  12 |       2 |        4 |    7 |    8 |    1 |      1.6
  13 |       2 |        5 |   11 |    9 |    1 |      2.6
  14 |       2 |        6 |   16 |   15 |    1 |      3.6
  15 |       2 |        7 |   17 |   15 |  0.6 |      4.6
  16 |       2 |        8 |   -2 |   -1 |    0 |      5.2
(16 rows)
```

Right driving side

Get \(2\) paths using right side driving topology from, point\(1\) to point \(2\) with heap paths and details.

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edges 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 |    5 |    1 |    1 |      0.4
   3 |       1 |        3 |    6 |    4 |  0.7 |      1.4
   4 |       1 |        4 |   -6 |    4 |  0.3 |      2.1
   5 |       1 |        5 |    7 |    8 |    1 |      2.4
   6 |       1 |        6 |   11 |    9 |    1 |      3.4
   7 |       1 |        7 |   16 |   15 |  0.4 |      4.4
   8 |       1 |        8 |   -2 |   -1 |    0 |      4.8
   9 |       2 |        1 |   -1 |    1 |  0.4 |        0
  10 |       2 |        2 |    5 |    1 |    1 |      0.4
  11 |       2 |        3 |    6 |    4 |  0.7 |      1.4
  12 |       2 |        4 |   -6 |    4 |  0.3 |      2.1
  13 |       2 |        5 |    7 |    8 |    1 |      2.4
  14 |       2 |        6 |   11 |   11 |    1 |      3.4
  15 |       2 |        7 |   12 |   13 |    1 |      4.4
  16 |       2 |        8 |   17 |   15 |    1 |      5.4
  17 |       2 |        9 |   16 |   15 |  0.4 |      6.4
  18 |       2 |       10 |   -2 |   -1 |    0 |      6.8
  19 |       3 |        1 |   -1 |    1 |  0.4 |        0
  20 |       3 |        2 |    5 |    1 |    1 |      0.4
  21 |       3 |        3 |    6 |    4 |  0.7 |      1.4
  22 |       3 |        4 |   -6 |    4 |  0.3 |      2.1
  23 |       3 |        5 |    7 |   10 |    1 |      2.4
  24 |       3 |        6 |    8 |   12 |  0.6 |      3.4
  25 |       3 |        7 |   -3 |   12 |  0.4 |        4
  26 |       3 |        8 |   12 |   13 |    1 |      4.4
  27 |       3 |        9 |   17 |   15 |    1 |      5.4
  28 |       3 |       10 |   16 |   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**
- **K shortest paths - Category**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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

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

pgr_withPointsDD(**Edges SQL**, **Points SQL**, **root vid**, **distance**, [**options A**])
pgr_withPointsDD(**Edges SQL**, **Points SQL**, **root vids**, **distance**, [**options B**])
options A: [directed, driving_side, details]
options B: [directed, driving_side, details, equicost]

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

Single vertex

pgr_withPointsDD(**Edges SQL**, **Points SQL**, **root vid**, **distance**, [**options**])
options: [directed, driving_side, details]

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

Example:
Right side driving topology, from point 1 within a distance of 3.3 with details.

```
SELECT * FROM pgr_withPointsDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 3.3,
  driving_side => 'r',
  details => true);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |    0 |        0
   2 |    5 |    1 |  0.4 |      0.4
   3 |    6 |    1 |    1 |      1.4
   4 |   -6 |    4 |  0.7 |      2.1
   5 |    7 |    4 |  0.3 |      2.4
(5 rows)
```

Multiple vertices

pgr_withPointsDD(**Edges SQL**, **Points SQL**, **root vids**, **distance**, [**options**])
options: [directed, driving_side, details, equicost]

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

Example:
From point \(1\) and vertex \(16\) within a distance of\(3.3\) with equicost on a directed graph

```
SELECT * FROM pgr_withPointsDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1, 16], 3.3,
  driving_side => 'l',
  equicost => true);
 seq | start_vid | node | edge | cost | agg_cost
-----+-----------+------+------+------+----------
   1 |        -1 |   -1 |   -1 |    0 |        0
   2 |        -1 |    6 |    1 |  0.6 |      0.6
   3 |        -1 |    7 |    4 |    1 |      1.6
   4 |        -1 |    5 |    1 |    1 |      1.6
   5 |        -1 |    3 |    7 |    1 |      2.6
   6 |        -1 |    8 |   10 |    1 |      2.6
   7 |        16 |   16 |   -1 |    0 |        0
   8 |        16 |   11 |    9 |    1 |        1
   9 |        16 |   15 |   16 |    1 |        1
  10 |        16 |   17 |   15 |    1 |        1
  11 |        16 |   10 |    3 |    1 |        2
  12 |        16 |   12 |   11 |    1 |        2
(12 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
Root vid	BIGINT	Identifier of the root vertex of the tree. • Negative values represent a point
Root vids	ARRAY [**ANY-INTEGER**]	Array of identifiers of the root vertices. • Negative values represent a point • \(0\) values are ignored • For optimization purposes, any duplicated value is ignored.
distance	FLOAT	Upper limit for the inclusion of a node in the result.

Optional parameters

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

With points optional parameters

Parameter	Type	Default	Description

Parameter	Type	Default	Description
driving_side	CHAR	b	Value in [r, l, b] indicating if the driving side is: • r for right driving side. • l for left driving side. • b for both.
details	BOOLEAN	false	• When true the results will include the points that are in the path. • When false the results will not include the points that are in the path.

Driving distance optional parameters

Column	Type	Default	Description
equicost	BOOLEAN	true	• When true the node will only appear in the closest from_v list. • When false which resembles several calls using the single starting point signatures. Tie brakes are arbitrary.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative **value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

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

Result Columns

RETURNS SET OF (seq, [start_vid], node, edge, cost, agg_cost)

Parameter	Type	Description

Parameter	Type	Description
seq	BIGINT	Sequential value starting from \(1\).
[start_vid]	BIGINT	Identifier of the root vertex.
node	BIGINT	Identifier of node within the limits from from_v.
edge	BIGINT	Identifier of the edge used to arrive to node. • \(0\) when node = from_v.
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from from_v to node.

Where:

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

Additional Examples

- **Driving side does not matter**

Driving side does not matter

From point \(1\) within a distance of\(3.3\), does not matter driving side, with details.

```
SELECT * FROM pgr_withPointsDD(
  'SELECT id, source, target, cost, reverse_cost FROM edges ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 3.3,
  driving_side => 'b',
  details => true);
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |   -1 |   -1 |    0 |        0
   2 |    5 |    1 |  0.4 |      0.4
   3 |    6 |    1 |  0.6 |      0.6
   4 |   -6 |    4 |  0.7 |      1.3
   5 |    7 |    4 |  0.3 |      1.6
   6 |    3 |    7 |    1 |      2.6
   7 |    8 |   10 |    1 |      2.6
   8 |   11 |    8 |    1 |      2.6
   9 |   -3 |   12 |  0.6 |      3.2
  10 |   -4 |    6 |  0.7 |      3.3
(10 rows)
```

See Also

- **pgr_drivingDistance**
- **pgr_alphaShape**
- **Sample Data**

Indices and tables

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

Introduction

This family of functions belongs to the **withPoints - Category** and the functions that compose them are based one way or another on dijkstra algorithm.

Depending on the name:

- pgr_withPoints is pgr_dijkstra **with points**
- pgr_withPointsCost is pgr_dijkstraCost **with points**
- pgr_withPointsCostMatrix is pgr_dijkstraCostMatrix **with points**
- pgr_withPointsKSP is pgr_ksp **with points**
- pgr_withPointsDD is pgr_drivingDistance **with points**

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** as described below
start vid	BIGINT	Identifier of the starting vertex of the path. Negative value is for point's identifier.
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices. Negative values are for point's identifiers.
end vid	BIGINT	Identifier of the ending vertex of the path. Negative value is for point's identifier.
end vids	ARRAY[BIGINT]	Array of identifiers of ending vertices. Negative values are for point's identifiers.

Optional parameters

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

With points optional parameters

Parameter	Type	Default	Description
driving_side	CHAR	b	Value in [r, l, b] indicating if the driving side is: • r for right driving side. • l for left driving side. • b for both.
details	BOOLEAN	false	• When true the results will include the points that are in the path. • When false the results will not include the points that are in the path.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative **value** 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.

Parameter	Type	Default	Description
side	CHAR	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

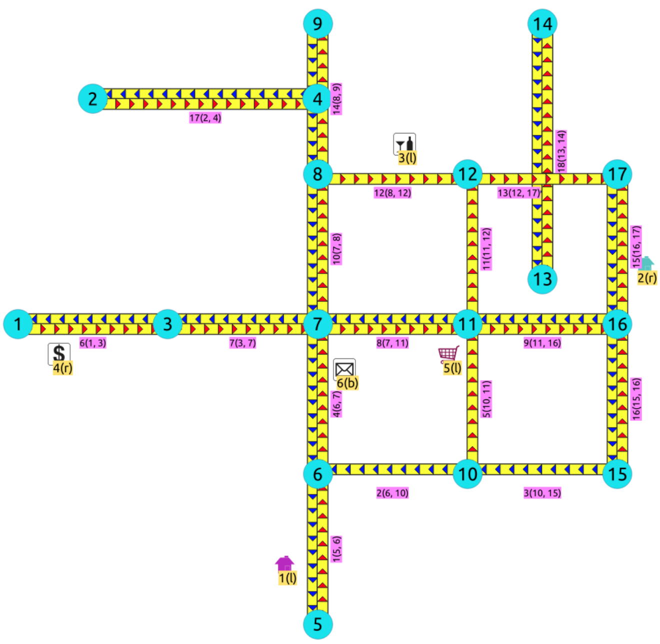
ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Advanced Documentation

Contents

About points

For this section the following city (see **Sample Data**) some interesing points such as restaurant, supermarket, post office, etc. will be used as example.



- The graph is **directed**
- Red arrows show the `(source, target)` of the edge on the edge table
- Blue arrows show the `(target, source)` of the edge on the edge table
- Each point location shows where it is located with relation of the edge `(source, target)`
  - On the right for points **2** and **4**.
  - On the left for points **1**, **3** and **5**.
  - On both sides for point **6**.

The representation on the data base follows the **Points SQL** description, and for this example:

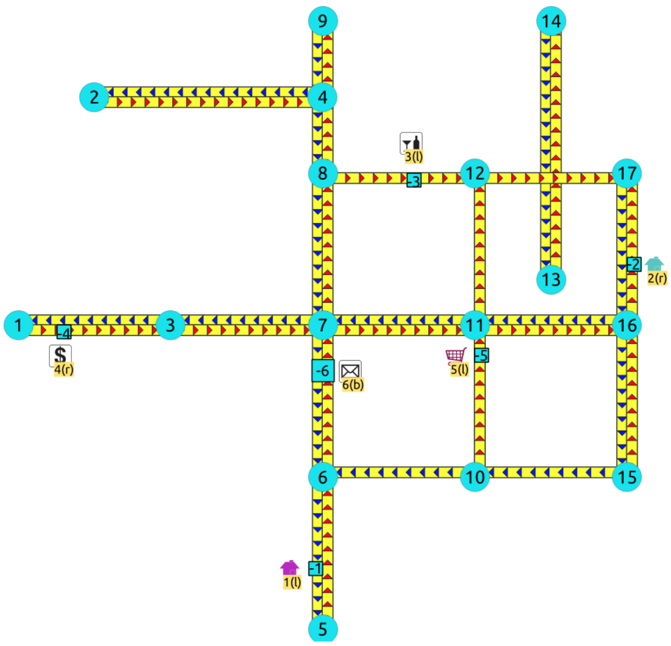
```
SELECT pid, edge_id, fraction, side FROM pointsOfInterest;
 pid | edge_id | fraction | side
-----+---------+----------+------
   1 |       1 |      0.4 | l
   2 |      15 |      0.4 | r
   3 |      12 |      0.6 | l
   4 |       6 |      0.3 | r
   5 |       5 |      0.8 | l
   6 |       4 |      0.7 | b
(6 rows)
```

Driving side

In the the folowwing images:

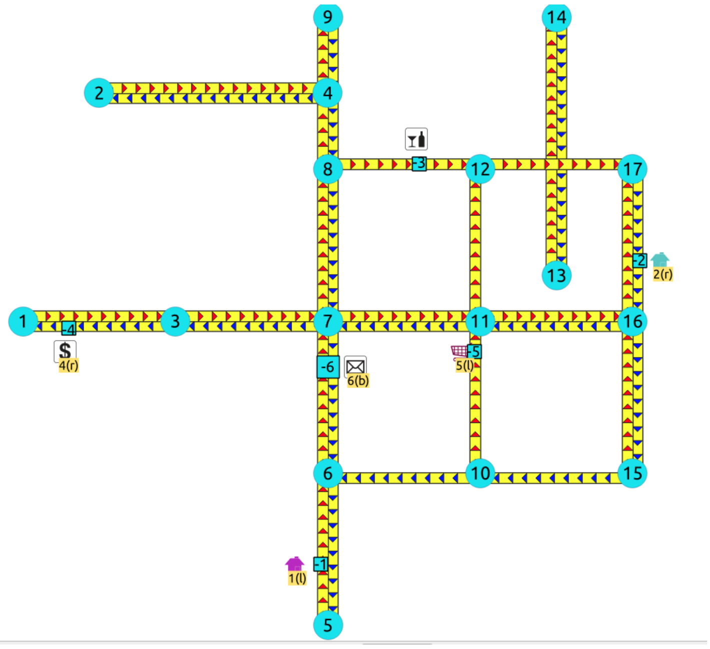
- The squared vertices are the temporary vertices,
- The temporary vertices are added according to the driving side,
- visually showing the differences on how depending on the driving side the data is interpreted.

Right driving side



- Point **1** located on edge `(6, 5)`
- Point **2** located on edge `(16, 17)`
- Point **3** located on edge `(8, 12)`
- Point **4** located on edge `(1, 3)`
- Point **5** located on edge `(10, 11)`
- Point **6** located on edges `(6, 7)` and `(7, 6)`

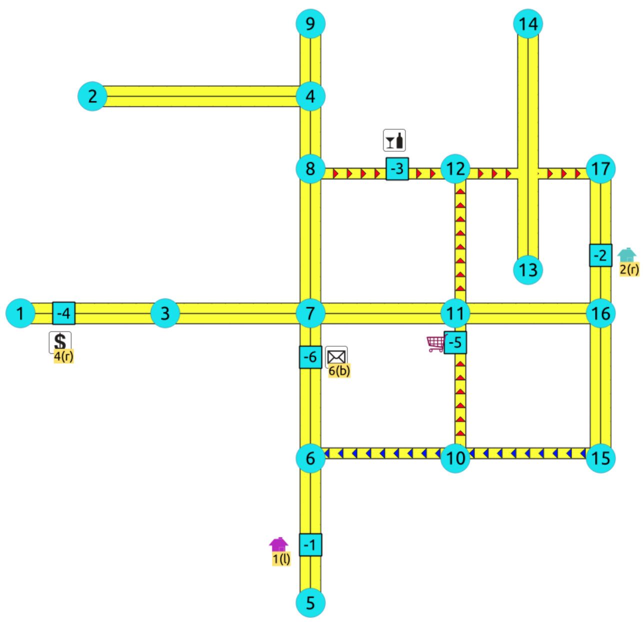
Left driving side



- Point **1** located on edge (5, 6)
- Point **2** located on edge (17, 16)
- Point **3** located on edge (8, 12)
- Point **4** located on edge (3, 1)
- Point **5** located on edge (10, 11)
- Point **6** located on edges (6, 7) and (7, 6)

Driving side does not matter

- Like having all points to be considered in both sidesb
- Prefered usage on **undirected** graphs



- Point **1** located on edge (5, 6) and (6, 5)
- Point **2** located on edge (17, 16)``and``16, 17
- Point **3** located on edge (8, 12)
- Point **4** located on edge (3, 1) and (1, 3)
- Point **5** located on edge (10, 11)
- Point **6** located on edges (6, 7) and (7, 6)

Creating temporary vertices

This section will demonstrate how a temporary vertex is created internally on the graph.

Problem

For edge:

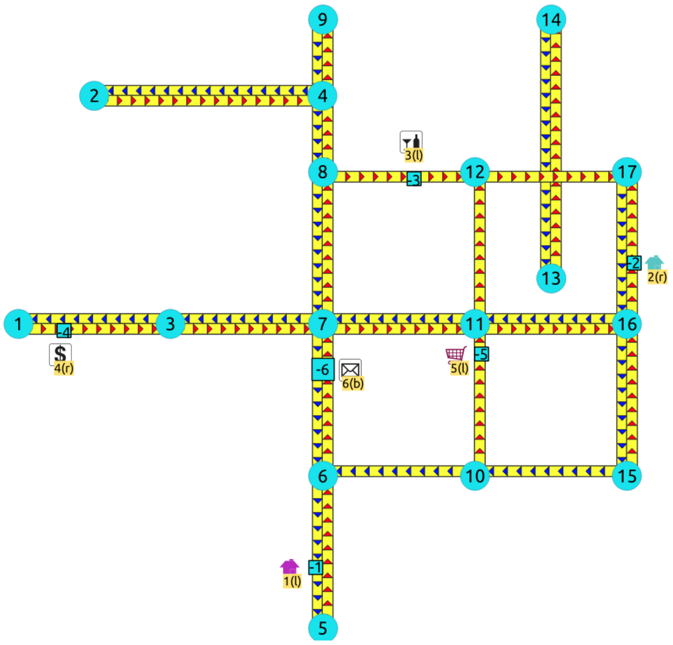
```
SELECT id, source, target, cost, reverse_cost
FROM edges WHERE id = 15;
 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
 15 |     16 |     17 |    1 |            1
(1 row)
```

insert point:

```
SELECT pid, edge_id, fraction, side
FROM pointsOfInterest WHERE pid = 2;
 pid | edge_id | fraction | side
-----+---------+----------+------
   2 |      15 |      0.4 | r
(1 row)
```

On a right hand side driving network

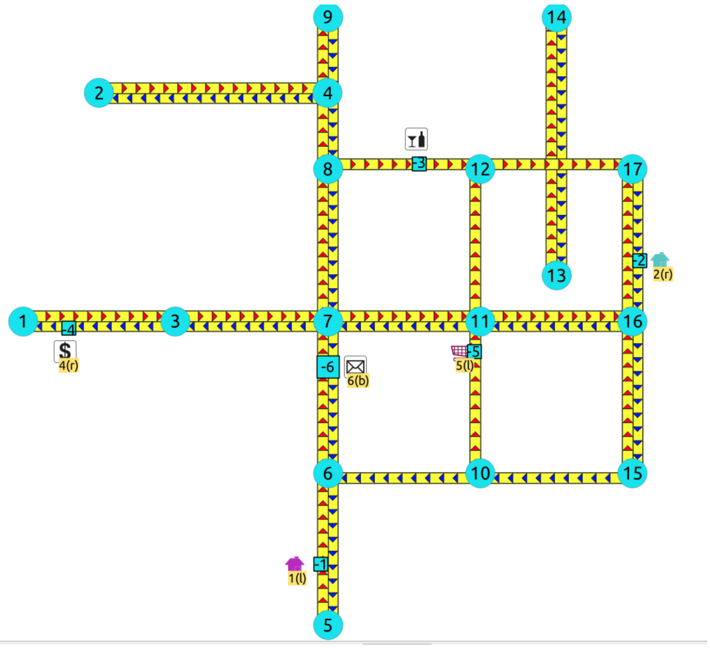
Right driving side



- Arrival to point `-2` can be achived only via vertex **16**.
- Does not affects edge `(17, 16)`, therefore the edge is kept.
- It only affects the edge `(16, 17)`, therefore the edge is removed.
- Create two new edges:
  - Edge `(16, -2)` with cost `0.4` (original cost * fraction == \(1 * 0.4\))
  - Edge `(-2, 17)` with cost `0.6` (the remaing cost)
- The total cost of the additional edges is equal to the original cost.
- If more points are on the same edge, the process is repeated recursevly.

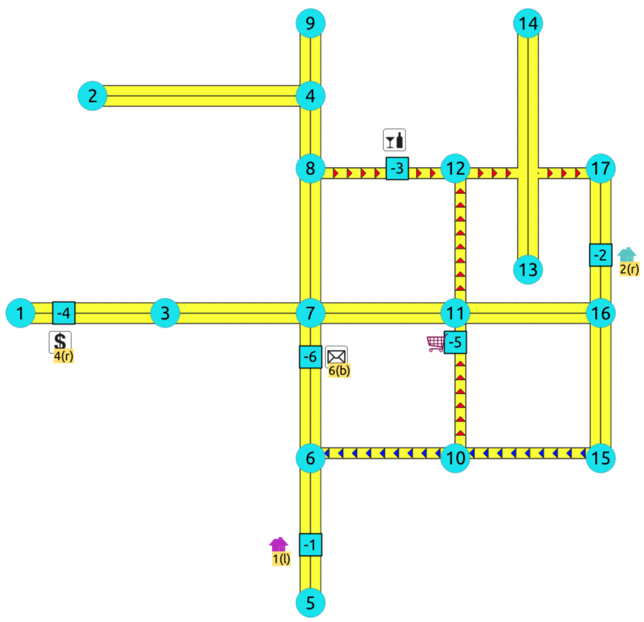
On a left hand side driving network

Left driving side



- Arrival to point -2 can be achived only via vertex **17**.
- Does not affects edge (16, 17), therefore the edge is kept.
- It only affects the edge (17, 16), therefore the edge is removed.
- Create two new edges:
  - Work with the original edge (16, 17) as the fraction is a fraction of the original:
    - Edge (16, -2) with cost 0.4 (original cost * fraction ==\(1 * 0.4\))
    - Edge (-2, 17) with cost 0.6 (the remaing cost)
    - If more points are on the same edge, the process is repeated recursevly.
  - Flip the Edges and add them to the graph:
    - Edge (17, -2) becomes (-2, 16) with cost 0.4 and is added to the graph.
    - Edge (-2, 16) becomes (17, -2) with cost 0.6 and is added to the graph.
- The total cost of the additional edges is equal to the original cost.

When driving side does not matter



- Arrival to point -2 can be achived via vertices **16** or **17**.
- Affects the edges (16, 17) and (17, 16), therefore the edges are removed.
- Create four new edges:
  - Work with the original edge (16, 17) as the fraction is a fraction of the original:
    - Edge (16, -2) with cost 0.4 (original cost * fraction ==\(1 * 0.4\))
    - Edge (-2, 17) with cost 0.6 (the remaing cost)
    - If more points are on the same edge, the process is repeated recursevly.
  - Flip the Edges and add all the edges to the graph:
    - Edge (16, -2) is added to the graph.
    - Edge (-2, 17) is added to the graph.
    - Edge (16, -2) becomes (-2, 16) with cost 0.4 and is added to the graph.

- Edge (-2, 17) becomes (17, -2) with cost 0.6 and is added to the graph.

See Also

- **withPoints - Category**

Indices and tables

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

- **Supported versions: Latest (3.3)**

Via - Category

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**

General Information

This category intends to solve the general problem:

> Given a graph and a list of vertices, find the shortest path between\(vertex_i\) and \(vertex_{i+1}\) for all vertices

In other words, find a continuos route that visits all the vertices in the order given.

path:
represents a section of a **route**.
route:
is a sequence of **paths**

Parameters

Used on:

- **pgr_dijkstraVia - Proposed**

Parameter	Type	Default	Description
Edges SQL	TEXT		SQL query as described.
via vertices	ARRAY [**ANY-INTEGER**]		Array of ordered vertices identifiers that are going to be visited.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Besides the compulsory parameters each function has, there are optional parameters that exist due to the kind of function.

Via optional parameters

Used on all Via functions

Parameter	Type	Default	Description

Parameter	Type	Default	Description
strict	BOOLEAN	false	- When true if a path is missing stops and returns **EMPTY SET** - When false ignores missing paths returning all paths found
U_turn_on_edge	BOOLEAN	true	- When true departing from a visited vertex will not try to avoid

Inner Queries

Depending on the function one or more inner queries are needed.

Edges SQL

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)
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

Restrictions SQL

Column	Type	Description
path	ARRAY [**ANY-INTEGER**]	Sequence of edge identifiers that form a path that is not allowed to be taken. - Empty arrays or NULL arrays are ignored. - Arrays that have a NULL element will raise an exception.
Cost	**ANY-NUMERICAL**	Cost of taking the forbidden path.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

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

Result Columns

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Identifier of a path. Has value **1** for the first path.
path_seq	INTEGER	Relative position in the path. Has value **1** for the beginning of a 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 path sequence.
agg_cost	FLOAT	Aggregate cost from start_vid to node.
route_agg_cost	FLOAT	Total cost from start_vid of seq = 1 to end_vid of the current seq.

Note

When start_vid, end_vid and node columns have negative values, the identifier is for a Point.

See Also

- **pgr_dijkstraVia - Proposed**

Indices and tables

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

- **Supported versions: Latest (3.3)**

withPoints - Category

When points are added to the graph.

> **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.

- **withPoints - Family of functions** - Functions based on Dijkstra algorithm.

Introduction

The **with points** category modifies the graph on the fly by adding points on edges as required by the **Points SQL** query.

The functions within this category give the ability to process between arbitrary points located outside the original graph.

This category of functions was thought for routing vehicles, but might as well work for some other application not involving vehicles.

When given a point identifier `pid` that its being mapped to an edge with an identifier `edge_id`, with a fraction from the source to the target along the edge `fraction` and some additional information about which side of the edge the point is on `side`, then processing from arbitrary points can be done on fixed networks.

All this functions consider as many traits from the "real world" as possible:

- Kind of graph:
  - **directed** graph
  - **undirected** graph
- Arriving at the point:
  - Compulsory arrival on the side of the segment where the point is located.
  - On either side of the segment.
- 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.
    - The graph has been modified to permanently have those points as vertices.
    - There is a table on the database that describes the points
  - **Temporal**: for example points given through a web application
- The numbering of the points are handled with negative sign.
  - This sign change is to avoid confusion when there is a vertex with the same identifier as the point identifier.
  - Original point identifiers are to be positive.
  - Transformation to negative is done internally.
  - Interpretation of the sign on the node information of the output
    - positive sign is a vertex of the original graph
    - negative sign is a point of the **Points SQL**

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Points SQL	TEXT	**Points SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** as described below
start vid	BIGINT	Identifier of the starting vertex of the path. Negative value is for point's identifier.
start vids	ARRAY[BIGINT]	Array of identifiers of starting vertices. Negative values are for point's identifiers.
end vid	BIGINT	Identifier of the ending vertex of the path. Negative value is for point's identifier.
end vids	ARRAY[BIGINT]	Array of identifiers of ending vertices. Negative values are for point's identifiers.

Optional parameters

Parameter	Type	Default	Description
driving_side	CHAR	r	Value in [r, l] indicating if the driving side is: • r for right driving side • l for left driving side • Any other value will be considered as r
details	BOOLEAN	false	• When true the results will include the points that are in the path. • When false the results will not include the points that are in the path.

Inner Queries

Edges SQL

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)
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

Points SQL

Parameter	Type	Default	Description
pid	**ANY-INTEGER**	**value**	Identifier of the point. • Use with positive value, as internally will be converted to negative value • If column is present, it can not be NULL. • If column is not present, a sequential negative **value** 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	b	Value in [b, r, l, NULL] indicating if the point is: • In the right r, • In the left l, • In both sides b, NULL

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:

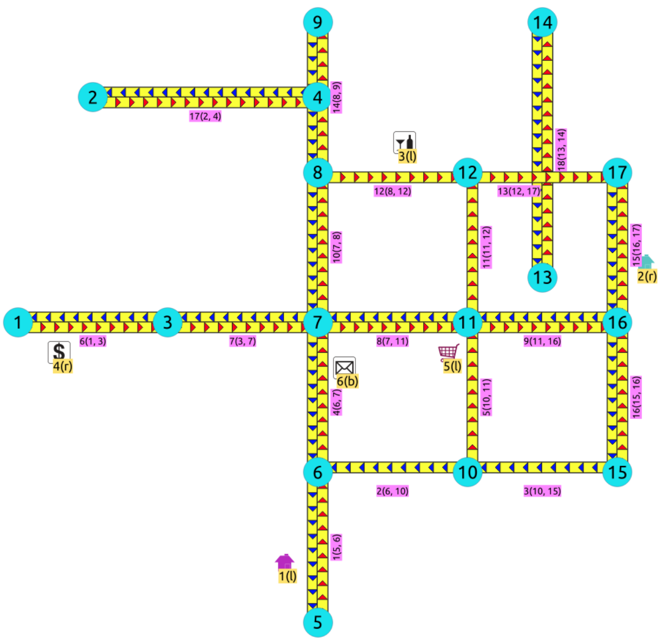
SMALLINT, INTEGER, BIGINT

Advanced documentation

Contents

About points

For this section the following city (see **Sample Data**) some interesing points such as restaurant, supermarket, post office, etc. will be used as example.



- The graph is **directed**
- Red arrows show the (source, target) of the edge on the edge table
- Blue arrows show the (target, source) of the edge on the edge table
- Each point location shows where it is located with relation of the edge (source, target)
  - On the right for points **2** and **4**.
  - On the left for points **1**, **3** and **5**.
  - On both sides for point **6**.

The representation on the data base follows the **Points SQL** description, and for this example:

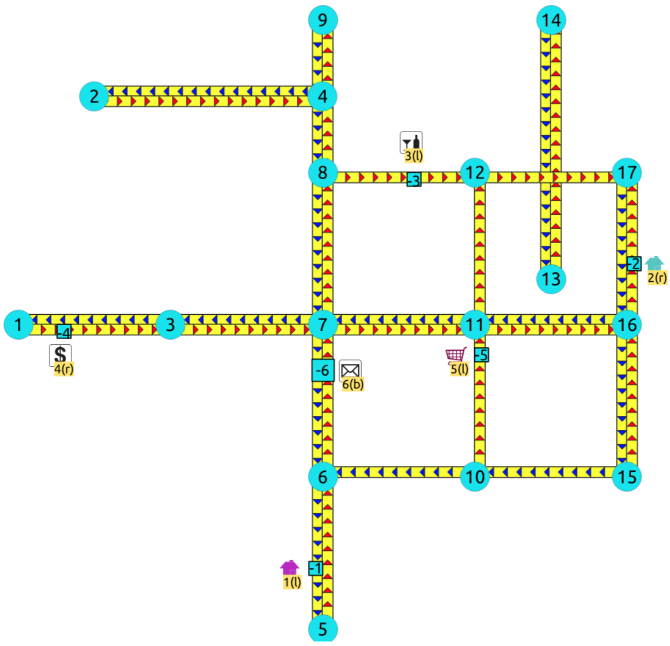
```
SELECT pid, edge_id, fraction, side FROM pointsOfInterest;
 pid | edge_id | fraction | side
-----+---------+----------+------
   1 |       1 |      0.4 | l
   2 |      15 |      0.4 | r
   3 |      12 |      0.6 | l
   4 |       6 |      0.3 | r
   5 |       5 |      0.8 | l
   6 |       4 |      0.7 | b
(6 rows)
```

Driving side

In the the folowwing images:

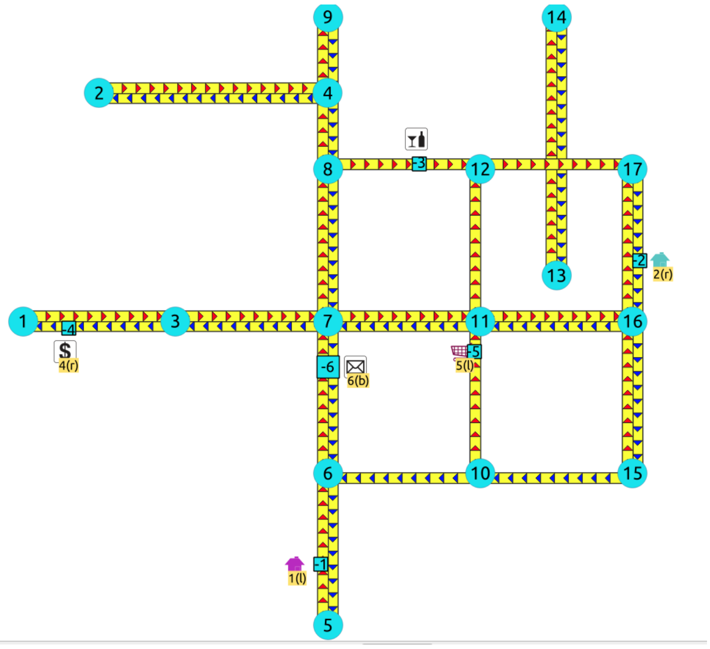
- The squared vertices are the temporary vertices,
- The temporary vertices are added according to the driving side,
- visually showing the differences on how depending on the driving side the data is interpreted.

Right driving side



- Point **1** located on edge (6, 5)
- Point **2** located on edge (16, 17)
- Point **3** located on edge (8, 12)
- Point **4** located on edge (1, 3)
- Point **5** located on edge (10, 11)
- Point **6** located on edges (6, 7) and (7, 6)

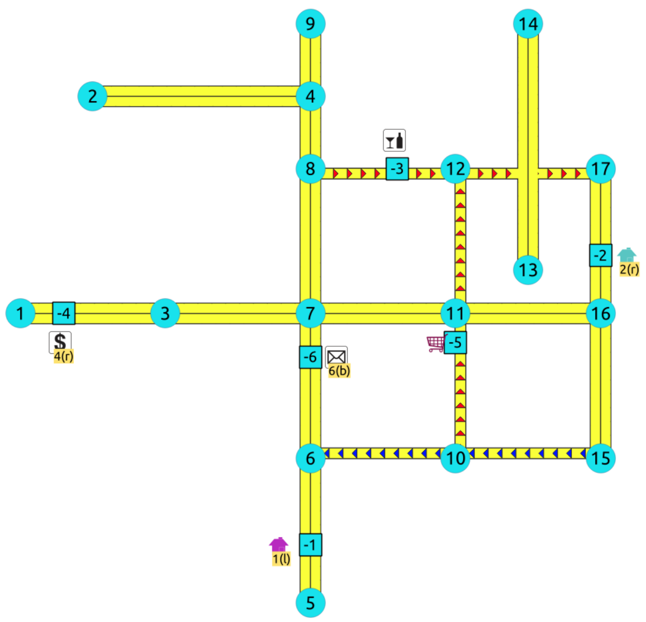
Left driving side



- Point **1** located on edge `(5, 6)`
- Point **2** located on edge `(17, 16)`
- Point **3** located on edge `(8, 12)`
- Point **4** located on edge `(3, 1)`
- Point **5** located on edge `(10, 11)`
- Point **6** located on edges `(6, 7)` and `(7, 6)`

Driving side does not matter

- Like having all points to be considered in both sides`b`
- Prefered usage on **undirected** graphs



- Point **1** located on edge `(5, 6)` and `(6, 5)`
- Point **2** located on edge `(17, 16)``and ``16, 17`
- Point **3** located on edge `(8, 12)`
- Point **4** located on edge `(3, 1)` and `(1, 3)`
- Point **5** located on edge `(10, 11)`
- Point **6** located on edges `(6, 7)` and `(7, 6)`

Creating temporary vertices

This section will demonstrate how a temporary vertex is created internally on the graph.

Problem

For edge:

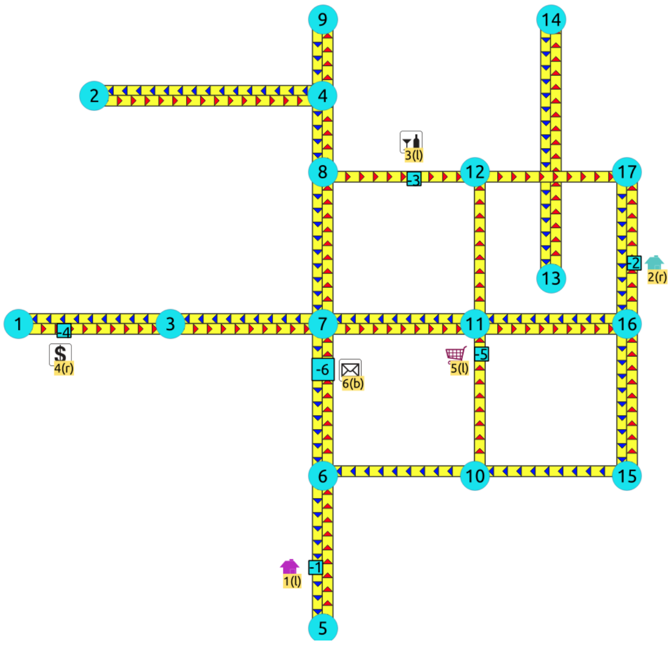
```
SELECT id, source, target, cost, reverse_cost
FROM edges WHERE id = 15;
 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
 15 |     16 |     17 |    1 |            1
(1 row)
```

insert point:

```
SELECT pid, edge_id, fraction, side
FROM pointsOfInterest WHERE pid = 2;
 pid | edge_id | fraction | side
-----+---------+----------+------
   2 |      15 |      0.4 | r
(1 row)
```

On a right hand side driving network

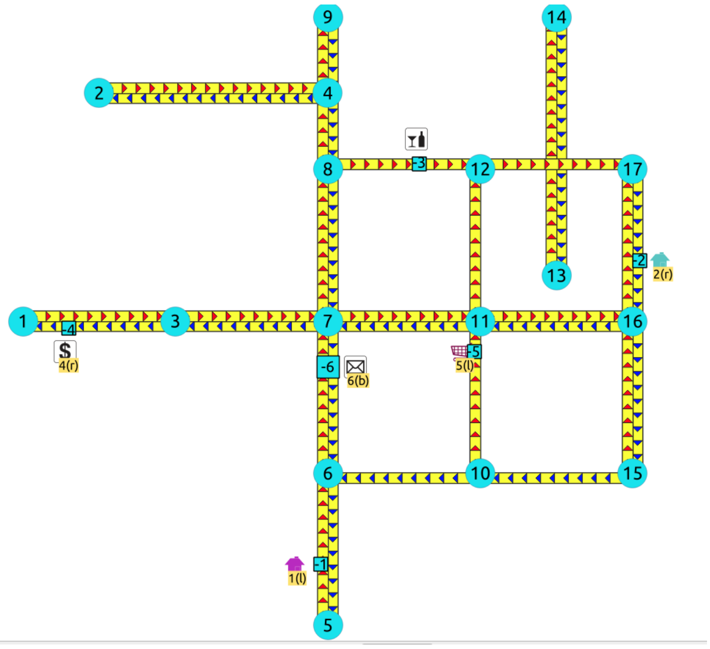
Right driving side



- Arrival to point -2 can be achived only via vertex **16**.
- Does not affects edge (17, 16), therefore the edge is kept.
- It only affects the edge (16, 17), therefore the edge is removed.
- Create two new edges:
  - Edge (16, -2) with cost 0.4 (original cost * fraction == \(1 * 0.4\))
  - Edge (-2, 17) with cost 0.6 (the remaing cost)
- The total cost of the additional edges is equal to the original cost.
- If more points are on the same edge, the process is repeated recursevly.

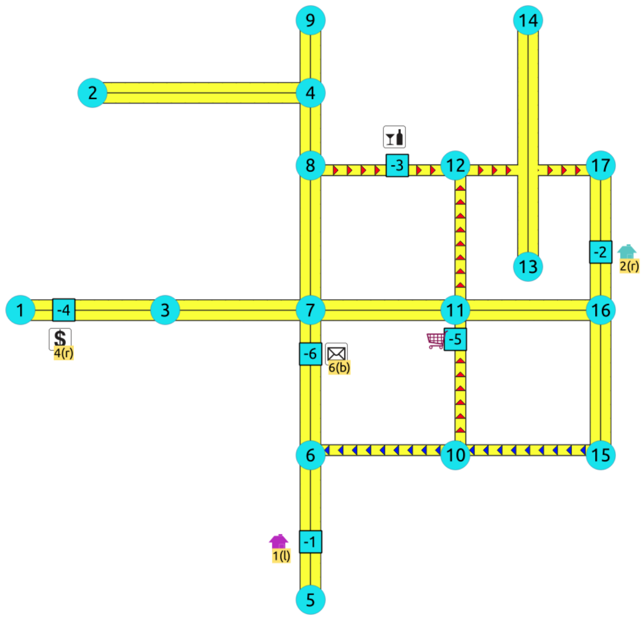
On a left hand side driving network

Left driving side



- Arrival to point `-2` can be achived only via vertex **17**.
- Does not affects edge `(16, 17)`, therefore the edge is kept.
- It only affects the edge `(17, 16)`, therefore the edge is removed.
- Create two new edges:
  - Work with the original edge `(16, 17)` as the fraction is a fraction of the original:
    - Edge `(16, -2)` with cost `0.4` (original cost * fraction ==\(1 * 0.4\))
    - Edge `(-2, 17)` with cost `0.6` (the remaing cost)
    - If more points are on the same edge, the process is repeated recursevly.
  - Flip the Edges and add them to the graph:
    - Edge `(17, -2)` becomes `(-2, 16)` with cost `0.4` and is added to the graph.
    - Edge `(-2, 16)` becomes `(17, -2)` with cost `0.6` and is added to the graph.
- The total cost of the additional edges is equal to the original cost.

When driving side does not matter



- Arrival to point `-2` can be achived via vertices **16** or **17**.
- Affects the edges `(16, 17)` and `(17, 16)`, therefore the edges are removed.
- Create four new edges:
  - Work with the original edge `(16, 17)` as the fraction is a fraction of the original:
    - Edge `(16, -2)` with cost `0.4` (original cost * fraction ==\(1 * 0.4\))
    - Edge `(-2, 17)` with cost `0.6` (the remaing cost)
    - If more points are on the same edge, the process is repeated recursevly.
  - Flip the Edges and add all the edges to the graph:
    - Edge `(16, -2)` is added to the graph.
    - Edge `(-2, 17)` is added to the graph.
    - Edge `(16, -2)` becomes `(-2, 16)` with cost `0.4` and is added to the graph.

- Edge (-2, 17) becomes (17, -2) with cost 0.6 and is added to the graph.

See Also

- **withPoints - Family of functions**

Indices and tables

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

See Also

- **Experimental Functions**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2**

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**

Coloring - Family of functions

- **pgr_bipartite -Experimental** - Bipartite graph algorithm using a DFS-based coloring approach.
- **pgr_edgeColoring - Experimental** - Edge Coloring algorithm using Vizing's theorem.

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.

Traversal - Family of functions

- **pgr_breadthFirstSearch - Experimental** - Breath first search traversal of the graph.
- **pgr_binaryBreadthFirstSearch - Experimental** - Breath first search traversal of the graph.

Components - Family of functions

- **pgr_makeConnected - Experimental** - Details of edges to make graph connected.

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**

Chinese Postman Problem - Family of functions (Experimental)

- **pgr_chinesePostman - Experimental**
- **pgr_chinesePostmanCost - Experimental**

- **Supported versions Latest** (**3.3**) **3.2 3.1 3.0**

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** signature

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 edges WHERE id < 17');
 seq | node | edge | cost | agg_cost
-----+------+------+------+----------
   1 |    1 |    6 |    1 |        0
   2 |    3 |    7 |    1 |        1
   3 |    7 |    4 |    1 |        2
   4 |    6 |    4 |    1 |        3
   5 |    7 |    8 |    1 |        4
   6 |   11 |    8 |    1 |        5
   7 |    7 |   10 |    1 |        6
   8 |    8 |   12 |    1 |        7
   9 |   12 |   13 |    1 |        8
  10 |   17 |   15 |    1 |        9
  11 |   16 |   15 |    1 |       10
  12 |   17 |   15 |    1 |       11
  13 |   16 |   16 |    1 |       12
  14 |   15 |   16 |    1 |       13
  15 |   16 |    9 |    1 |       14
  16 |   11 |   11 |    1 |       15
  17 |   12 |   13 |    1 |       16
  18 |   17 |   15 |    1 |       17
  19 |   16 |   16 |    1 |       18
  20 |   15 |    3 |    1 |       19
  21 |   10 |    5 |    1 |       20
  22 |   11 |    9 |    1 |       21
  23 |   16 |   16 |    1 |       22
  24 |   15 |    3 |    1 |       23
  25 |   10 |    2 |    1 |       24
  26 |    6 |    1 |    1 |       25
  27 |    5 |    1 |    1 |       26
  28 |    6 |    4 |    1 |       27
  29 |    7 |   10 |    1 |       28
  30 |    8 |   14 |    1 |       29
  31 |    9 |   14 |    1 |       30
  32 |    8 |   10 |    1 |       31
  33 |    7 |    7 |    1 |       32
  34 |    3 |    6 |    1 |       33
  35 |    1 |   -1 |    0 |       34
(35 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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`
`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)**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

`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** signature

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.

- 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 edges WHERE id < 17');
 pgr_chinesepostmancost
------------------------
                     34
(1 row)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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

Column	Type	Description
pgr_chinesepostmancost	FLOAT	Minimum costs of a circuit path.

See Also

- **Chinese Postman Problem - Family of functions (Experimental)**
- **Sample Data**

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

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

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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**

- **Supported versions: Latest** (**3.3**) **3.2 3.1**) **3.0**
- **Unsupported versions: 2.6**

Transformation - Family of functions (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_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.

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**
- **Unsupported versions: 2.6 2.5**

pgr_lineGraph - Experimental

pgr_lineGraph — Transforms the given graph into its corresponding edge-based 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 2.5.0
  - New **Experimental** function

Description

Given a graph G, its line graph L(G) is a graph such that:

- Each vertex of L(G) represents an edge of G
- Two vertices of L(G) are adjacent if and only if their corresponding edges share a common endpoint in G.

Signatures

```
pgr_lineGraph(Edges SQL, [directed])

RETURNS SET OF (seq, source, target, cost, reverse_cost)
OR EMPTY SET
```

Example:
For a **directed** graph

```
SELECT * FROM pgr_lineGraph(
  'SELECT id, source, target, cost, reverse_cost FROM edges'
);
 seq | source | target | cost | reverse_cost
-----+--------+--------+------+--------------
   1 |    -18 |     18 |    1 |            1
   2 |    -17 |     17 |    1 |            1
   3 |    -16 |     -3 |    1 |           -1
   4 |    -14 |    -10 |    1 |            1
   5 |    -14 |     12 |    1 |           -1
   6 |    -14 |     14 |    1 |            1
   7 |    -10 |     -7 |    1 |            1
   8 |    -10 |     -4 |    1 |            1
   9 |    -10 |      8 |    1 |            1
  10 |    -10 |     10 |    1 |            1
  11 |     -9 |     -8 |    1 |            1
  12 |     -9 |      9 |    1 |            1
  13 |     -9 |     11 |    1 |           -1
  14 |     -8 |     -7 |    1 |            1
  15 |     -8 |     -4 |    1 |            1
  16 |     -8 |      8 |    1 |            1
  17 |     -7 |     -6 |    1 |            1
  18 |     -7 |      7 |    1 |            1
  19 |     -6 |      6 |    1 |            1
  20 |     -3 |     -2 |    1 |           -1
  21 |     -3 |      5 |    1 |           -1
  22 |     -2 |     -1 |    1 |           -1
  23 |     -2 |      4 |    1 |           -1
  24 |      1 |     -1 |    1 |            1
  25 |      1 |      4 |    1 |            1
  26 |      4 |     -7 |    1 |            1
  27 |      4 |     -4 |    1 |            1
  28 |      5 |     -8 |    1 |           -1
  29 |      5 |      9 |    1 |           -1
  30 |      5 |     11 |    1 |           -1
  31 |      8 |     11 |    1 |           -1
  32 |      9 |    -16 |    1 |            1
  33 |      9 |     15 |    1 |            1
  34 |     10 |     12 |    1 |           -1
  35 |     11 |     13 |    1 |           -1
  36 |     12 |     13 |    1 |           -1
  37 |     13 |    -15 |    1 |           -1
  38 |     15 |    -15 |    1 |            1
  39 |     16 |    -16 |    1 |            1
  40 |     16 |     15 |    1 |            1
(40 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Optional parameters

Column	Type	Default	Description

Column	Type	Default	Description
directed	BOOLEAN	true	- When true the graph is considered *Directed* - When false the graph is considered as *Undirected*.

Inner Queries

Edges SQL

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)
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, source, target, cost, reverse_cost)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**. - Gives a local identifier for the edge
source	BIGINT	Identifier of the source vertex of the current edge. - When *negative*: the source is the reverse edge in the original graph.
target	BIGINT	Identifier of the target vertex of the current edge. - When *negative*: the target is the reverse edge in the original graph.
cost	FLOAT	Weight of the edge (source, target). - When *negative*: edge (source, target) does not exist, therefore it's not part of the graph.
reverse_cost	FLOAT	Weight of the edge (target, source). - When *negative*: edge (target, source) does not exist, therefore it's not part of the graph.

See Also

- **https://en.wikipedia.org/wiki/Line_graph**
- The queries use the **Sample Data** network.

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**
- **Unsupported versions: 2.6**

pgr_lineGraphFull - Experimental

pgr_lineGraphFull — Transforms a given graph into a new graph where all of the vertices from the original graph are converted to line graphs.

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 2.6.0
  - New **Experimental** function

Description

`pgr_lineGraphFull`, converts original directed graph to a directed line graph by converting each vertex to a complete graph and keeping all the original edges. The new connecting edges have a cost 0 and go between the adjacent original edges, respecting the directionality.

A possible application of the resulting graph is **"routing with two edge restrictions"**:

- Setting a cost of using the vertex when routing between edges on the connecting edge
- Forbid the routing between two edges by removing the connecting edge

This is possible because each of the intersections (vertices) in the original graph are now complete graphs that have a new edge for each possible turn across that intersection.

The main characteristics are:

- This function is for **directed** graphs.
- Results are undefined when a negative vertex id is used in the input graph.
- Results are undefined when a duplicated edge id is used in the input graph.
- Running time: TBD

Signatures

Summary

pgr_lineGraphFull(**Edges SQL**)

RETURNS SET OF `(seq, source, target, cost, edge)`
OR EMPTY SET

Example:
Full line graph of subgraph of edges\(\{4, 7, 8, 10\}\)

```
SELECT * FROM pgr_lineGraphFull(
  $$SELECT id, source, target, cost, reverse_cost
  FROM edges
  WHERE id IN (4, 7, 8, 10)$$);
 seq | source | target | cost | edge
-----+--------+--------+------+------
   1 |     -1 |      7 |    1 |    4
   2 |      6 |     -1 |    0 |    0
   3 |     -2 |      6 |    1 |   -4
   4 |     -3 |      3 |    1 |   -7
   5 |     -4 |     11 |    1 |    8
   6 |     -5 |      8 |    1 |   10
   7 |      7 |     -2 |    0 |    0
   8 |      7 |     -3 |    0 |    0
   9 |      7 |     -4 |    0 |    0
  10 |      7 |     -5 |    0 |    0
  11 |     -6 |     -2 |    0 |    0
  12 |     -6 |     -3 |    0 |    0
  13 |     -6 |     -4 |    0 |    0
  14 |     -6 |     -5 |    0 |    0
  15 |     -7 |     -2 |    0 |    0
  16 |     -7 |     -3 |    0 |    0
  17 |     -7 |     -4 |    0 |    0
  18 |     -7 |     -5 |    0 |    0
  19 |     -8 |     -2 |    0 |    0
  20 |     -8 |     -3 |    0 |    0
  21 |     -8 |     -4 |    0 |    0
  22 |     -8 |     -5 |    0 |    0
  23 |     -9 |     -6 |    1 |    7
  24 |      3 |     -9 |    0 |    0
  25 |    -10 |     -7 |    1 |   -8
  26 |     11 |    -10 |    0 |    0
  27 |    -11 |     -8 |    1 |  -10
  28 |      8 |    -11 |    0 |    0
(28 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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, source, target, cost, edge)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**. • Gives a local identifier for the edge
source	BIGINT	Identifier of the source vertex of the current edge. • When *negative*: the source is the reverse edge in the original graph.

Column	Type	Description
target	BIGINT	Identifier of the target vertex of the current edge. • When *negative*: the target is the reverse edge in the original graph.
cost	FLOAT	Weight of the edge (source, target). • When *negative*: edge (source, target) does not exist, therefore it's not part of the graph.
reverse_cost	FLOAT	Weight of the edge (target, source). • When *negative*: edge (target, source) does not exist, therefore it's not part of the graph.

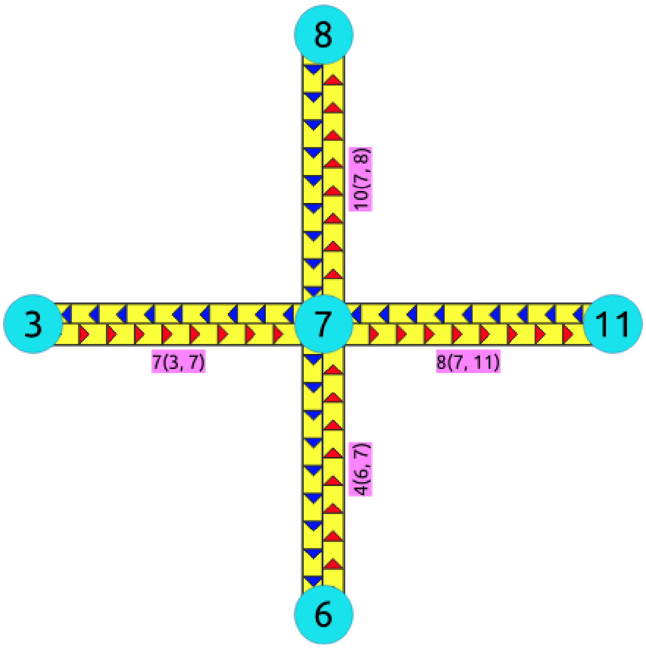
Additional Examples

The examples of this section are based on the **Sample Data** network. The examples include the subgraph including edges 4, 7, 8, and 10 with reverse_cost.

The data

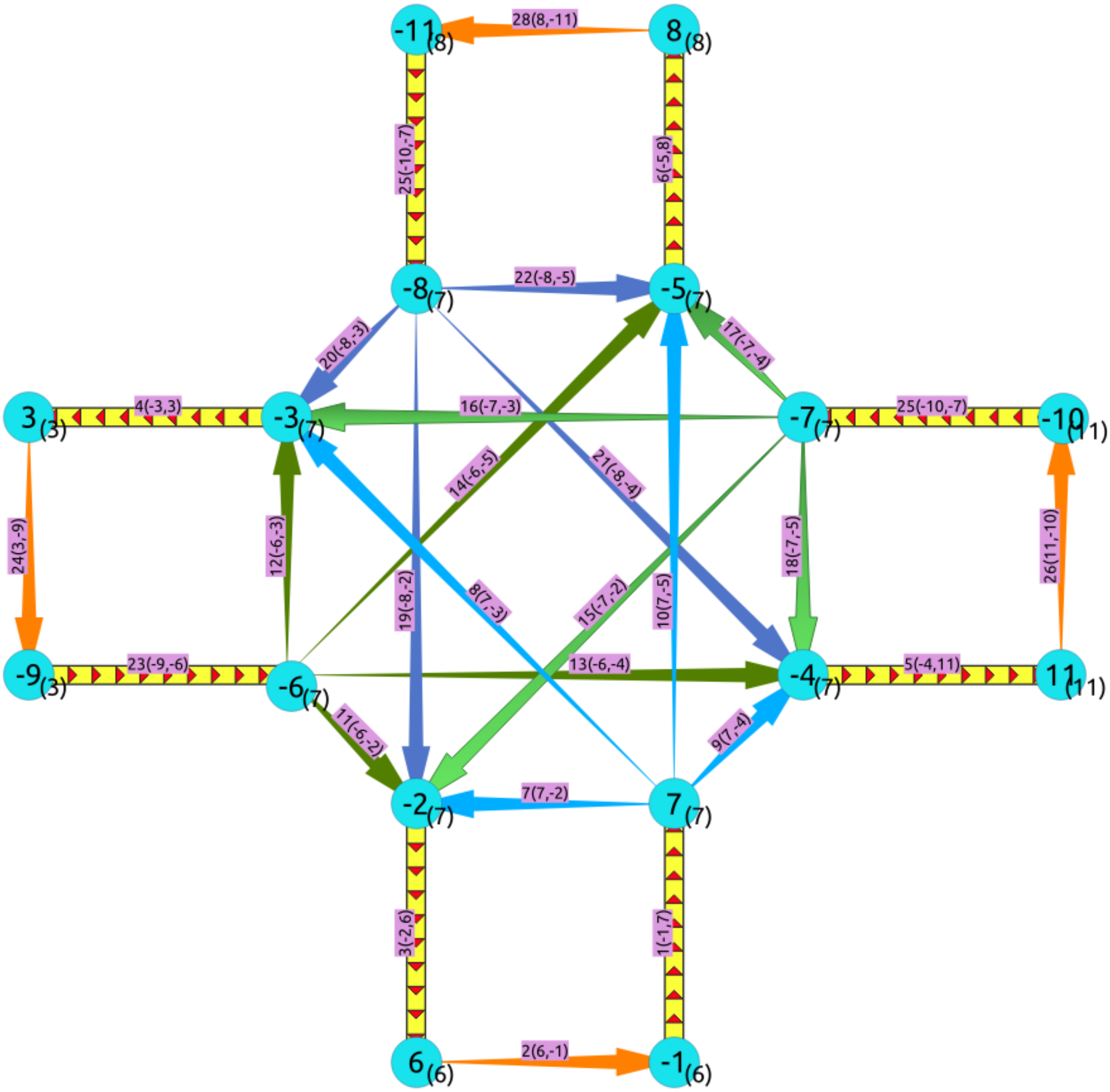
This example displays how this graph transformation works to create additional edges for each possible turn in a graph.

```
SELECT id, source, target, cost, reverse_cost
FROM edges
WHERE id IN (4, 7, 8, 10);
 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
  4 |      6 |      7 |    1 |            1
  7 |      3 |      7 |    1 |            1
  8 |      7 |     11 |    1 |            1
 10 |      7 |      8 |    1 |            1
(4 rows)
```



The transformation

```
SELECT * FROM pgr_lineGraphFull(
  $$SELECT id, source, target, cost, reverse_cost
  FROM edges
  WHERE id IN (4, 7, 8, 10)$$);
 seq | source | target | cost | edge
-----+--------+--------+------+------
   1 |     -1 |      7 |    1 |    4
   2 |      6 |     -1 |    0 |    0
   3 |     -2 |      6 |    1 |   -4
   4 |     -3 |      3 |    1 |   -7
   5 |     -4 |     11 |    1 |    8
   6 |     -5 |      8 |    1 |   10
   7 |      7 |     -2 |    0 |    0
   8 |      7 |     -3 |    0 |    0
   9 |      7 |     -4 |    0 |    0
  10 |      7 |     -5 |    0 |    0
  11 |     -6 |     -2 |    0 |    0
  12 |     -6 |     -3 |    0 |    0
  13 |     -6 |     -4 |    0 |    0
  14 |     -6 |     -5 |    0 |    0
  15 |     -7 |     -2 |    0 |    0
  16 |     -7 |     -3 |    0 |    0
  17 |     -7 |     -4 |    0 |    0
  18 |     -7 |     -5 |    0 |    0
  19 |     -8 |     -2 |    0 |    0
  20 |     -8 |     -3 |    0 |    0
  21 |     -8 |     -4 |    0 |    0
  22 |     -8 |     -5 |    0 |    0
  23 |     -9 |     -6 |    1 |    7
  24 |      3 |     -9 |    0 |    0
  25 |    -10 |     -7 |    1 |   -8
  26 |     11 |    -10 |    0 |    0
  27 |    -11 |     -8 |    1 |  -10
  28 |      8 |    -11 |    0 |    0
(28 rows)
```



In the transformed graph, all of the edges from the original graph are still present (yellow), but we now have additional edges for every turn that could be made across vertex 7 (orange).

Creating table that identifies transformed vertices

The vertices in the transformed graph are each created by splitting up the vertices in the original graph. Unless a vertex in the original graph is a leaf vertex, it will generate more than one vertex in the transformed graph. One of the newly created vertices in the transformed graph will be given the same vertex identifier as the vertex that it was created from in the original graph, but the rest of the newly created vertices will have negative vertex ids.

Following is an example of how to generate a table that maps the ids of the newly created vertices with the original vertex that they were created from

Store edge results

The first step is to store the results of the `pgr_lineGraphFull` call into a table

```
SELECT seq AS id, source, target, cost, edge
INTO lineGraph_edges
FROM pgr_lineGraphFull(
  $$SELECT id, source, target, cost, reverse_cost
  FROM edges
  WHERE id IN (4, 7, 8, 10)$$);
SELECT 28
```

Create the mapping table

From the original graph’s vertex information

```
SELECT id, NULL::BIGINT original_id
INTO vertex_map
FROM vertices;
SELECT 17
```

Add the new vertices

```
INSERT INTO vertex_map (id)
(SELECT id
FROM pgr_extractVertices(
  $$SELECT id, source, target FROM lineGraph_edges$$) WHERE id < 0);
INSERT 0 11
```

Filling the mapping table

The positive vertex identifiers are the original identifiers

```
UPDATE vertex_map
SET original_id = id
WHERE id > 0;
UPDATE 17
```

Inspecting the vertices map

```
SELECT *
FROM vertex_map ORDER BY id DESC;
 id  | original_id
-----+-------------
  17 |          17
  16 |          16
  15 |          15
  14 |          14
  13 |          13
  12 |          12
  11 |          11
  10 |          10
   9 |           9
   8 |           8
   7 |           7
   6 |           6
   5 |           5
   4 |           4
   3 |           3
   2 |           2
   1 |           1
  -1 |
  -2 |
  -3 |
  -4 |
  -5 |
  -6 |
  -7 |
  -8 |
  -9 |
 -10 |
 -11 |
(28 rows)
```

The self loops happen when there is no cost traveling to the `target` and the source has an original value.

```
SELECT *, source AS targets_original_id
  FROM lineGraph_edges
  WHERE cost = 0 and source > 0;
 id | source | target | cost | edge | targets_original_id
----+--------+--------+------+------+---------------------
  2 |      6 |     -1 |    0 |    0 |                   6
  7 |      7 |     -2 |    0 |    0 |                   7
  8 |      7 |     -3 |    0 |    0 |                   7
  9 |      7 |     -4 |    0 |    0 |                   7
 10 |      7 |     -5 |    0 |    0 |                   7
 24 |      3 |     -9 |    0 |    0 |                   3
 26 |     11 |    -10 |    0 |    0 |                  11
 28 |      8 |    -11 |    0 |    0 |                   8
(8 rows)
```

Updating values from self loops

```
WITH
self_loops AS (
  SELECT DISTINCT source, target, source AS targets_original_id
  FROM lineGraph_edges
  WHERE cost = 0 and source > 0)
UPDATE vertex_map SET original_id = targets_original_id
FROM self_loops WHERE target = id;
UPDATE 8
```

Inspecting the vertices table

```
SELECT *
FROM vertex_map WHERE id < 0
ORDER BY id DESC;
 id  | original_id
-----+-------------
  -1 |           6
  -2 |           7
  -3 |           7
  -4 |           7
  -5 |           7
  -6 |
  -7 |
  -8 |
  -9 |           3
 -10 |          11
 -11 |           8
(11 rows)
```

Updating from inner self loops

```
WITH
assigned_vertices
AS (SELECT id, original_id
  FROM vertex_map
  WHERE original_id IS NOT NULL),
cross_edges
AS (SELECT DISTINCT e.source, v.original_id AS source_original_id
  FROM lineGraph_edges AS e
  JOIN vertex_map AS v ON (e.target = v.id)
  WHERE source NOT IN (SELECT id FROM assigned_vertices)
)
UPDATE vertex_map SET original_id = source_original_id
FROM cross_edges WHERE source = id;
UPDATE 3
```

Inspecting the vertices map

```
SELECT *
FROM vertex_map WHERE id < 0
ORDER BY id DESC;
 id  | original_id
-----+-------------
  -1 |           6
  -2 |           7
  -3 |           7
  -4 |           7
  -5 |           7
  -6 |           7
  -7 |           7
  -8 |           7
  -9 |           3
 -10 |          11
 -11 |           8
(11 rows)
```

Adding a soft restriction

A soft restriction going from vertex 6 to vertex 3 using edges 4 -> 7 is wanted.

Idenifying the restriction

Running a **pgr_dijkstraNear - Proposed** the edge with cost 0, edge 8, is where the cost will be increased

```
SELECT seq, path_seq, start_vid, end_vid, node, original_id, edge, cost, agg_cost
FROM (SELECT * FROM pgr_dijkstraNear(
  $$SELECT * FROM lineGraph_edges$$,
  (SELECT array_agg(id) FROM vertex_map where original_id = 6),
  (SELECT array_agg(id) FROM vertex_map where original_id = 3))) dn
JOIN vertex_map AS v1 ON (node = v1.id);
 seq | path_seq | start_vid | end_vid | node | original_id | edge | cost | agg_cost
-----+----------+-----------+---------+------+-------------+------+------+----------
   3 |        3 |        -1 |       3 |   -3 |           7 |    4 |    1 |        1
   1 |        1 |        -1 |       3 |   -1 |           6 |    1 |    1 |        0
   4 |        4 |        -1 |       3 |    3 |           3 |   -1 |    0 |        2
   2 |        2 |        -1 |       3 |    7 |           7 |    8 |    0 |        1
(4 rows)
```

The edge to be altered is `WHERE cost = 0 AND seq != 1 AND edge != -1` from the previus query:

```
SELECT edge FROM pgr_dijkstraNear(
  $$SELECT * FROM lineGraph_edges$$,
  (SELECT array_agg(id) FROM vertex_map where original_id = 6),
  (SELECT array_agg(id) FROM vertex_map where original_id = 3))
WHERE cost = 0 AND seq != 1 AND edge != -1;
 edge
------
    8
(1 row)
```

Adding a value to the restriction

Updating the cost to the edge:

```
UPDATE lineGraph_edges
SET cost = 100
WHERE id IN (
SELECT edge FROM pgr_dijkstraNear(
  $$SELECT * FROM lineGraph_edges$$,
  (SELECT array_agg(id) FROM vertex_map where original_id = 6),
  (SELECT array_agg(id) FROM vertex_map where original_id = 3))
WHERE cost = 0 AND seq != 1 AND edge != -1);
UPDATE 1
```

Example:

Routing from \(6\) to \(3\)

Now the route does not use edge 8 and does a U turn on a leaf vertex.

```
WITH
results AS (
  SELECT * FROM pgr_dijkstraNear(
    $$SELECT * FROM lineGraph_edges$$,
    (SELECT array_agg(id) FROM vertex_map where original_id = 6),
    (SELECT array_agg(id) FROM vertex_map where original_id = 3)))
SELECT seq, path_seq, start_vid, end_vid, node, original_id, edge, cost, agg_cost
FROM results
LEFT JOIN vertex_map AS v1 ON (node = v1.id) ORDER BY seq;
 seq | path_seq | start_vid | end_vid | node | original_id | edge | cost | agg_cost
-----+----------+-----------+---------+------+-------------+------+------+----------
   1 |        1 |        -1 |       3 |   -1 |           6 |    1 |    1 |        0
   2 |        2 |        -1 |       3 |    7 |           7 |   10 |    0 |        1
   3 |        3 |        -1 |       3 |   -5 |           7 |    6 |    1 |        1
   4 |        4 |        -1 |       3 |    8 |           8 |   28 |    0 |        2
   5 |        5 |        -1 |       3 |  -11 |           8 |   27 |    1 |        2
   6 |        6 |        -1 |       3 |   -8 |           7 |   20 |    0 |        3
   7 |        7 |        -1 |       3 |   -3 |           7 |    4 |    1 |        3
   8 |        8 |        -1 |       3 |    3 |           3 |   -1 |    0 |        4
(8 rows)
```

Simplifying leaf vertices

In this example, there is no additional cost for traversing a leaf vertex.

Using the vertex map give the leaf verices their original value.

On the source column

```
WITH
u_turns AS (
SELECT e.id AS eid, v1.original_id
FROM linegraph_edges as e
JOIN vertex_map AS v1 ON (source = v1.id)
AND v1.original_id IN (3, 6, 8, 11))
UPDATE lineGraph_edges
SET source = original_id
FROM u_turns
WHERE id = eid;
UPDATE 8
```

On the target column

```
WITH
u_turns AS (
SELECT e.id AS eid, v1.original_id
FROM linegraph_edges as e
JOIN vertex_map AS v1 ON (target = v1.id)
AND v1.original_id IN (3, 6, 8, 11))
UPDATE lineGraph_edges
SET target = original_id
FROM u_turns
WHERE id = eid;
UPDATE 8
```

Removing self loops on leaf nodes

The self loops of the leaf nodes are

```
SELECT * FROM linegraph_edges
WHERE source = target
ORDER BY id;
 id | source | target | cost | edge
----+--------+--------+------+------
  2 |      6 |      6 |    0 |    0
 24 |      3 |      3 |    0 |    0
 26 |     11 |     11 |    0 |    0
 28 |      8 |      8 |    0 |    0
(4 rows)
```

Which can be removed

```
DELETE FROM linegraph_edges
WHERE source = target;
DELETE 4
```

Example:

Routing from \(6\) to \(3\)

Routing can be done now using the original vertices id using**pgr_dijkstra**

```
WITH
results AS (
  SELECT * FROM pgr_dijkstra(
    $$SELECT * FROM lineGraph_edges$$, 6, 3))
SELECT seq, path_seq, node, original_id, edge, cost, agg_cost
FROM results
LEFT JOIN vertex_map AS v1 ON (node = v1.id) ORDER BY seq;
 seq | path_seq | node | original_id | edge | cost | agg_cost
-----+----------+------+-------------+------+------+----------
   1 |        1 |    6 |           6 |    1 |    1 |        0
   2 |        2 |    7 |           7 |    9 |    0 |        1
   3 |        3 |   -4 |           7 |    5 |    1 |        1
   4 |        4 |   11 |          11 |   25 |    1 |        2
   5 |        5 |   -7 |           7 |   16 |    0 |        3
   6 |        6 |   -3 |           7 |    4 |    1 |        3
   7 |        7 |    3 |           3 |   -1 |    0 |        4
(7 rows)
```

Complete routing graph

Add edges from the original graph

Add all the edges that are not involved in the line graph process to the new table

```
SELECT id, source, target, cost, reverse_cost
INTO new_graph from edges
WHERE id NOT IN (4, 7, 8, 10);
SELECT 14
```

Some administrative tasks to get new identifiers for the edges

```
CREATE SEQUENCE new_graph_id_seq;
CREATE SEQUENCE
ALTER TABLE new_graph ALTER COLUMN id SET DEFAULT nextval('new_graph_id_seq');
ALTER TABLE
ALTER TABLE new_graph ALTER COLUMN id SET NOT NULL;
ALTER TABLE
ALTER SEQUENCE new_graph_id_seq OWNED BY new_graph.id;
ALTER SEQUENCE
SELECT setval('new_graph_id_seq', (SELECT max(id) FROM new_graph));
 setval
--------
     18
(1 row)
```

Add the newly calculated edges

```
INSERT INTO new_graph (source, target, cost, reverse_cost)
SELECT source, target, cost, -1 FROM lineGraph_edges;
INSERT 0 24
```

Using the routing graph

When using this method for routing with soft restrictions there will be uturns

Example:
Routing from \(6\) to \(3\)

```
WITH
results AS (
  SELECT * FROM pgr_dijkstra(
    $$SELECT * FROM new_graph$$, 6, 3))
SELECT seq, path_seq, node, original_id, edge, cost, agg_cost
FROM results
LEFT JOIN vertex_map AS v1 ON (node = v1.id) ORDER BY seq;
 seq | path_seq | node | original_id | edge | cost | agg_cost
-----+----------+------+-------------+------+------+----------
   1 |        1 |    6 |           6 |   35 |    1 |        0
   2 |        2 |    7 |           7 |   20 |    0 |        1
   3 |        3 |   -4 |           7 |   41 |    1 |        1
   4 |        4 |   11 |          11 |   37 |    1 |        2
   5 |        5 |   -7 |           7 |   27 |    0 |        3
   6 |        6 |   -3 |           7 |   40 |    1 |        3
   7 |        7 |    3 |           3 |   -1 |    0 |        4
(7 rows)
```

Example:
Routing from \(5\) to \(1\)

```
WITH
results AS (
  SELECT * FROM pgr_dijkstra(
    $$SELECT * FROM new_graph$$, 5, 1))
SELECT seq, path_seq, node, original_id, edge, cost, agg_cost
FROM results
LEFT JOIN vertex_map AS v1 ON (node = v1.id) ORDER BY seq;
 seq | path_seq | node | original_id | edge | cost | agg_cost
-----+----------+------+-------------+------+------+----------
   1 |        1 |    5 |           5 |    1 |    1 |        0
   2 |        2 |    6 |           6 |   35 |    1 |        1
   3 |        3 |    7 |           7 |   20 |    0 |        2
   4 |        4 |   -4 |           7 |   41 |    1 |        2
   5 |        5 |   11 |          11 |   37 |    1 |        3
   6 |        6 |   -7 |           7 |   27 |    0 |        4
   7 |        7 |   -3 |           7 |   40 |    1 |        4
   8 |        8 |    3 |           3 |    6 |    1 |        5
   9 |        9 |    1 |           1 |   -1 |    0 |        6
(9 rows)
```

See Also

- **https://en.wikipedia.org/wiki/Line_graph**
- **https://en.wikipedia.org/wiki/Complete_graph**

Indices and tables

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

Introduction

This family of functions is used for transforming a given input graph\(G(V,E)\) into a new graph \(G'(V',E')\).

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

- **Supported versions: Latest** (**3.2 3.1**) **3.0**

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**

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

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, [options])
options: [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)
```

Example:

Solve the following problem

Given the vehicles:

```
SELECT id, capacity, start_node_id, start_open, start_close
FROM vehicles;
 id | capacity | start_node_id | start_open | start_close
----+----------+---------------+------------+-------------
  1 |       50 |            11 |          0 |          50
  2 |       50 |            11 |          0 |          50
(2 rows)
```

and the orders:

```
SELECT id, demand,
      p_node_id, p_open, p_close, p_service,
      d_node_id, d_open, d_close, d_service
FROM orders;
 id | demand | p_node_id | p_open | p_close | p_service | d_node_id | d_open | d_close | d_service
----+--------+-----------+--------+---------+-----------+-----------+--------+---------+-----------
  1 |     10 |        10 |      2 |      10 |         3 |         3 |      6 |      15 |         3
  2 |     20 |        16 |      4 |      15 |         2 |        15 |      6 |      20 |         3
  3 |     30 |         7 |      2 |      10 |         3 |        12 |      3 |      20 |         3
(3 rows)
```

The query:

```
SELECT * FROM pgr_pickDeliver(
  $$SELECT id, demand,
      p_node_id, p_open, p_close, p_service,
      d_node_id, d_open, d_close, d_service
    FROM orders$$,
  $$SELECT id, capacity, start_node_id, start_open, start_close
    FROM vehicles$$,
  $$SELECT * from pgr_dijkstraCostMatrix(
    'SELECT * FROM edges ',
    (SELECT array_agg(id) FROM (SELECT p_node_id AS id FROM orders
      UNION
      SELECT d_node_id FROM orders
      UNION
      SELECT start_node_id FROM vehicles) a))
  $$);
 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 |      11 |       -1 |     0 |           0 |            0 |         0 |            0 |              0
   2 |           1 |          1 |        2 |         2 |       7 |        3 |    30 |           1 |            1 |         1 |            3 |              5
   3 |           1 |          1 |        3 |         3 |      12 |        3 |     0 |           2 |            7 |         0 |            3 |             10
   4 |           1 |          1 |        4 |         2 |      16 |        2 |    20 |           2 |           12 |         0 |            2 |             14
   5 |           1 |          1 |        5 |         3 |      15 |        2 |     0 |           1 |           15 |         0 |            3 |             18
   6 |           1 |          1 |        6 |         6 |      11 |       -1 |     0 |           2 |           20 |         0 |            0 |             20
   7 |           2 |          2 |        1 |         1 |      11 |       -1 |     0 |           0 |            0 |         0 |            0 |              0
   8 |           2 |          2 |        2 |         2 |      10 |        1 |    10 |           3 |            3 |         0 |            3 |              6
   9 |           2 |          2 |        3 |         3 |       3 |        1 |     0 |           3 |            9 |         0 |            3 |             12
  10 |           2 |          2 |        4 |         6 |      11 |       -1 |     0 |           2 |           14 |         0 |            0 |             14
  11 |          -2 |          0 |        0 |        -1 |      -1 |       -1 |    -1 |          16 |           -1 |         1 |           17 |             34
(11 rows)
```

Parameters

The parameters are:

Column	Type	Description
Orders SQL	TEXT	**Orders SQL** as described below.
Vehicles SQL	TEXT	**Vehicles SQL** as described below.
Matrix SQL	TEXT	**Matrix SQL** as described below.

Pick-Deliver optional parameters

Column	Type	Default	Description
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

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	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.
[p_service]	ANY-NUMERICAL	The duration of the loading at the pickup location. • When missing: 0 time units are used
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	The duration of the unloading at the delivery location. • When missing: 0 time units are used

Where:

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

Column	Type	Description
p_node_id	ANY-INTEGER	The node identifier of the pickup, must match a vertex identifier in the**Matrix SQL**.
d_node_id	ANY-INTEGER	The node identifier of the delivery, must match a vertex identifier in the**Matrix SQL**.

Where:

ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

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	Description
id	ANY-NUMERICAL	Identifier of the vehicle.
capacity	ANY-NUMERICAL	Maiximum capacity units
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	The duration of the loading at the starting location. • When missing: A duration of \(0\) time units is used.
[end_open]	ANY-NUMERICAL	The time, relative to 0, when the ending location opens. • When missing: The value of start_open is used
[end_close]	ANY-NUMERICAL	The time, relative to 0, when the ending location closes. • When missing: The value of start_close is used
[end_service]	ANY-NUMERICAL	The duration of the loading at the ending location. • When missing: A duration in start_service is used.

Column	Type	Description
start_node_id	ANY-INTEGER	The node identifier of the start location, must match a vertex identifier in the **Matrix SQL**.
[end_node_id]	ANY-INTEGER	The node identifier of the end location, must match a vertex identifier in the **Matrix SQL**. • When missing: end_node_id is used.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Matrix SQL

Where:

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

Return columns

```
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. • Value \(-2\) indicates it is the summary row.

Column	Type	Description
vehicle_id	BIGINT	Current vehicle identifier. - Sumary row has the **total capacity violations**. - A capacity violation happens when overloading or underloading a vehicle.
stop_seq	INTEGER	Sequential value starting from **1** for the stops made by the current vehicle. The m_{th} stop of the current vehicle. - Sumary row has the **total time windows violations**. - A time window violation happens when arriving after the location has closed.

stop_type	INTEGER	- Kind of stop location the vehicle is at
		  - -1: at the solution summary row
		  - 1: Starting location
		  - 2: Pickup location
		  - 3: Delivery location
		  - 6: Ending location and indicates the vehicle's summary row

order_id	BIGINT	Pickup-Delivery order pair identifier. - Value -1: When no order is involved on the current stop location.
cargo	FLOAT	Cargo units of the vehicle when leaving the stop. - Value -1 on solution summary row.
travel_time	FLOAT	Travel time from previous stop_seq to current stop_seq. - Summary has the **total traveling time**: - The sum of all the travel_time.
arrival_time	FLOAT	Time spent waiting for current location to open. - -1: at the solution summary row. - 0: at the starting location.
wait_time	FLOAT	Time spent waiting for current location to open. - Summary row has the **total waiting time**: - The sum of all the wait_time.
service_time	FLOAT	Service duration at current location. - Summary row has the **total service time**: - The sum of all the service_time.

departure_time	FLOAT	- The time at which the vehicle departs from the stop.
		  - $arrival\_time + wait\_time + service\_time$.
		- The ending location has the **total time** used by the current vehicle.
		- Summary row has the **total solution time**:
		  - $total\ traveling\ time + total\ waiting\ time + total\ service\ time$.

See Also

- **Vehicle Routing Functions - Category (Experimental)**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1**

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

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**, [**options**])
options: [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)

Example:
Solve the following problem

Given the vehicles:

```
SELECT id, capacity, start_x, start_y, start_open, start_close
FROM vehicles;
 id | capacity | start_x | start_y | start_open | start_close
----+----------+---------+---------+------------+-------------
  1 |       50 |       3 |       2 |          0 |          50
  2 |       50 |       3 |       2 |          0 |          50
(2 rows)
```

and the orders:

```
SELECT id, demand,
       p_x, p_y, p_open, p_close, p_service,
       d_x, d_y, d_open, d_close, d_service
FROM orders;
 id | demand | p_x | p_y | p_open | p_close | p_service | d_x | d_y | d_open | d_close | d_service
----+--------+-----+-----+--------+---------+-----------+-----+-----+--------+---------+-----------
  1 |     10 |   3 |   1 |      2 |      10 |         3 |   1 |   2 |      6 |      15 |         3
  2 |     20 |   4 |   2 |      4 |      15 |         2 |   4 |   1 |      6 |      20 |         3
  3 |     30 |   2 |   2 |      2 |      10 |         3 |   3 |   3 |      3 |      20 |         3
(3 rows)
```

The query:

```
SELECT * FROM pgr_pickDeliverEuclidean(
  $$SELECT id, demand,
       p_x, p_y, p_open, p_close, p_service,
       d_x, d_y, d_open, d_close, d_service
    FROM orders$$,
  $$SELECT id, capacity, start_x, start_y, start_open, start_close
    FROM vehicles$$);
 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 |          2 |        1 |         1 |       -1 |     0 |             0 |             0 |         0 |            0 |              0
   8 |           2 |          2 |        2 |         2 |        1 |    10 |             1 |             1 |         1 |            3 |              5
   9 |           2 |          2 |        3 |         3 |        1 |     0 |  2.2360679775 |  7.2360679775 |         0 |            3 |  10.2360679775
  10 |           2 |          2 |        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)
```

Parameters

Column	Type	Description
Orders SQL	TEXT	**Orders SQL** as described below.
Vehicles SQL	TEXT	**Vehicles SQL** as described below.

Pick-Deliver optional parameters

Column	Type	Default	Description
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

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	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.
[p_service]	ANY-NUMERICAL	The duration of the loading at the pickup location. • When missing: 0 time units are used
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	The duration of the unloading at the delivery location. • When missing: 0 time units are used

Where:

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

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

Where:

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

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	Description
id	ANY-NUMERICAL	Identifier of the vehicle.
capacity	ANY-NUMERICAL	Maiximum capacity units
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	The duration of the loading at the starting location. • When missing: A duration of\(0\) time units is used.
[end_open]	ANY-NUMERICAL	The time, relative to 0, when the ending location opens. • When missing: The value ofstart_open is used
[end_close]	ANY-NUMERICAL	The time, relative to 0, when the ending location closes. • When missing: The value ofstart_close is used

Column	Type	Description
[end_service]	ANY-NUMERICAL	The duration of the loading at the ending location. • When missing: A duration in start_service is used.

Column	Type	Description
start_x	ANY-NUMERICAL	x value of the starting location
start_y	ANY-NUMERICAL	y value of the starting location
[end_x]	ANY-NUMERICAL	x value of the ending location • When missing: start_x is used.
[end_y]	ANY-NUMERICAL	y value of the ending location • When missing: start_y is used.

Where:

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

Return columns

```
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. • Value -2 indicates it is the summary row.
vehicle_id	BIGINT	Current vehicle identifier. • Sumary row has the **total capacity violations**. ◦ A capacity violation happens when overloading or underloading a vehicle.
stop_seq	INTEGER	Sequential value starting from **1** for the stops made by the current vehicle. The m_{th} stop of the current vehicle. • Sumary row has the **total time windows violations**. ◦ A time window violation happens when arriving after the location has closed.

stop_type	INTEGER	• Kind of stop location the vehicle is at
		  ◦ -1: at the solution summary row
		  ◦ 1: Starting location
		  ◦ 2: Pickup location
		  ◦ 3: Delivery location
		  ◦ 6: Ending location and indicates the vehicle's summary row

order_id	BIGINT	Pickup-Delivery order pair identifier. • Value -1: When no order is involved on the current stop location.
cargo	FLOAT	Cargo units of the vehicle when leaving the stop. • Value -1 on solution summary row.
travel_time	FLOAT	Travel time from previous stop_seq to current stop_seq. • Summary has the **total traveling time**: ◦ The sum of all the travel_time.
arrival_time	FLOAT	Time spent waiting for current location to open. • -1: at the solution summary row. • 0: at the starting location.
wait_time	FLOAT	Time spent waiting for current location to open. • Summary row has the **total waiting time**: ◦ The sum of all the wait_time.

Column	Type	Description
service_time	FLOAT	Service duration at current location. • Summary row has the **total service time**: ◦ The sum of all the service_time.

departure_time	FLOAT	• The time at which the vehicle departs from the stop.
		  ◦ $arrival\_time + wait\_time + service\_time$.
		• The ending location has the **total time** used by the current vehicle.
		• Summary row has the **total solution time**:
		  ◦ $total\ traveling\ time + total\ waiting\ time + total\ service\ time$.

Example

- **The vehicles**
- **The original orders**
- **The orders**
- **The query**

This data example **lc101** is from data published at **https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/**

The vehicles

There are 25 vehciles in the problem all with the same characteristics.

```
CREATE TABLE v_lc101(
  id BIGINT NOT NULL primary key,
  capacity BIGINT DEFAULT 200,
  start_x FLOAT DEFAULT 30,
  start_y FLOAT DEFAULT 50,
  start_open INTEGER DEFAULT 0,
  start_close INTEGER DEFAULT 1236);
CREATE TABLE
/* create 25 vehciles */
INSERT INTO v_lc101 (id)
(SELECT * FROM generate_series(1, 25));
INSERT 0 25
```

The original orders

The data comes in different rows for the pickup and the delivery of the same order.

```
CREATE table lc101_c(
  id BIGINT not null primary key,
  x DOUBLE PRECISION,
  y DOUBLE PRECISION,
  demand INTEGER,
  open INTEGER,
  close INTEGER,
  service INTEGER,
  pindex BIGINT,
  dindex BIGINT
);
CREATE TABLE
/* the original data */
INSERT INTO lc101_c(
  id,   x,   y, demand, open, close, service, pindex, dindex) VALUES
( 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,  -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);
INSERT 0 106
```

The orders

The original data needs to be converted to an appropiate table:

```
WITH deliveries AS (SELECT * FROM lc101_c WHERE dindex = 0)
SELECT
  row_number() over() AS id, p.demand,
  p.id as p_node_id, p.x AS p_x, p.y AS p_y, p.open AS p_open, p.close as p_close, p.service as p_service,
  d.id as d_node_id, d.x AS d_x, d.y AS d_y, d.open AS d_open, d.close as d_close, d.service as d_service
INTO c_lc101
FROM deliveries as d JOIN lc101_c as p ON (d.pindex = p.id);
SELECT 53
SELECT * FROM c_lc101 LIMIT 1;
 id | demand | p_node_id | p_x | p_y | p_open | p_close | p_service | d_node_id | d_x | d_y | d_open | d_close | d_service
----+--------+-----------+-----+-----+--------+---------+-----------+-----------+-----+-----+--------+---------+-----------
  1 |     10 |         3 |  42 |  66 |     65 |     146 |        90 |        75 |  45 |  65 |    997 |    1068 |        90
(1 row)
```

The query

Showing only the relevant information to compare with the best solution information published on **https://www.sintef.no/projectweb/top/pdptw/100-customers/**

- The best solution found for **lc101** is a travel time: 828.94
- This implementation's travel time: 854.54

```
SELECT travel_time, 828.94 AS best
FROM pgr_pickDeliverEuclidean(
  $$SELECT * FROM c_lc101 $$,
  $$SELECT * FROM v_lc101 $$,
  max_cycles => 2, initial_sol => 4) WHERE vehicle_seq = -2;
   travel_time    |  best
------------------+--------
 854.5412705652799 | 828.94
(1 row)
```

See Also

- **Vehicle Routing Functions - Category (Experimental)**
- The queries use the **Sample Data** network.

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1**

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
- **TBD**

Description

- TBD

Signatures

- TBD

Parameters

- TBD

Inner Queries

- 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
   7 |    2 |   1 |        0 |       0
   9 |    3 |   1 |        0 |       0
   8 |    4 |   1 |        0 |       0
   6 |    5 |   1 |        0 |       0
   5 |    6 |   1 |        0 |       0
   4 |    7 |   1 |        0 |       0
   2 |    8 |   1 |        0 |       0
   6 |    9 |   1 |       40 |      51
   8 |   10 |   1 |       62 |      89
   9 |   11 |   1 |       94 |     104
   7 |   12 |   1 |      110 |     120
   4 |   13 |   1 |      131 |     141
   2 |   14 |   1 |      144 |     155
   5 |   15 |   1 |      162 |     172
  -1 |   16 |   1 |      208 |     208
  -1 |    1 |   2 |        0 |       0
  10 |    2 |   2 |        0 |       0
  11 |    3 |   2 |        0 |       0
  10 |    4 |   2 |       34 |     101
  11 |    5 |   2 |      106 |     129
  -1 |    6 |   2 |      161 |     161
  -1 |    1 |   3 |        0 |       0
   3 |    2 |   3 |        0 |       0
   3 |    3 |   3 |       31 |      60
  -1 |    4 |   3 |       91 |      91
  -1 |    0 |   0 |       -1 |     460
(27 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
);

INSERT INTO solomon_100_RC_101 (id, x, y, order_unit, open_time, close_time, service_time) VALUES
(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);

DROP TABLE IF EXISTS vrp_vehicles cascade;
CREATE TABLE vrp_vehicles (
    vehicle_id integer not null primary key,
    capacity integer,
    case_no integer
);

INSERT INTO vrp_vehicles (vehicle_id, capacity, case_no) VALUES
(1, 200, 5),
(2, 200, 5),
(3, 200, 5);

DROP TABLE IF EXISTS vrp_distance cascade;
WITH
the_matrix_info AS (
    SELECT A.id AS src_id, B.id AS dest_id, sqrt( (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y)) AS cost
    FROM solomon_100_rc_101 AS A, solomon_100_rc_101 AS B WHERE A.id != B.id
)
SELECT src_id, dest_id, cost, cost AS distance, cost AS traveltime
INTO vrp_distance
FROM the_matrix_info;
```

See Also

- https://en.wikipedia.org/wiki/Vehicle_routing_problem

Indices and tables

- Index
- Search Page

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

Used in **pgr_pickDeliverEuclidean - Experimental**

Column	Type	Description
Orders SQL	TEXT	**Orders SQL** as described below.
Vehicles SQL	TEXT	**Vehicles SQL** as described below.

Used in **pgr_pickDeliver - Experimental**

Column	Type	Description
Orders SQL	TEXT	**Orders SQL** as described below.
Vehicles SQL	TEXT	**Vehicles SQL** as described below.
Matrix SQL	TEXT	**Matrix SQL** as described below.

Pick-Deliver optional parameters

Column	Type	Default	Description
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

Inner Queries

Orders SQL

Common columns for the orders SQL in both implementations:

Column	Type	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.
[p_service]	ANY-NUMERICAL	The duration of the loading at the pickup location. • When missing: 0 time units are used
d_open	ANY-NUMERICAL	The time, relative to 0, when the delivery location opens.

Column	Type	Description
d_close	ANY-NUMERICAL	The time, relative to 0, when the delivery location closes.
[d_service]	ANY-NUMERICAL	The duration of the unloading at the delivery location. • When missing: 0 time units are used

Where:

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

For **pgr_pickDeliver - Experimental** the pickup and delivery identifiers of the locations are needed:

Column	Type	Description
p_node_id	ANY-INTEGER	The node identifier of the pickup, must match a vertex identifier in the**Matrix SQL**.
d_node_id	ANY-INTEGER	The node identifier of the delivery, must match a vertex identifier in the**Matrix SQL**.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

For **pgr_pickDeliverEuclidean - Experimental** the \((x, y)\) values of the 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

Where:

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

Vehicles SQL

Common columns for the vehicles SQL in both implementations:

Column	Type	Description
id	ANY-NUMERICAL	Identifier of the vehicle.
capacity	ANY-NUMERICAL	Maiximum capacity units
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	The duration of the loading at the starting location. • When missing: A duration of\(0\) time units is used.
[end_open]	ANY-NUMERICAL	The time, relative to 0, when the ending location opens. • When missing: The value ofstart_open is used
[end_close]	ANY-NUMERICAL	The time, relative to 0, when the ending location closes. • When missing: The value ofstart_close is used
[end_service]	ANY-NUMERICAL	The duration of the loading at the ending location. • When missing: A duration in start_service is used.

For **pgr_pickDeliver - Experimental** the starting and ending identifiers of the locations are needed:

Column	Type	Description

Column	Type	Description
start_node_id	ANY-INTEGER	The node identifier of the start location, must match a vertex identifier in the **Matrix SQL**.
[end_node_id]	ANY-INTEGER	The node identifier of the end location, must match a vertex identifier in the **Matrix SQL**. • When missing: end_node_id is used.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

For **pgr_pickDeliverEuclidean - Experimental** the (x, y) values of the locations are needed:

Column	Type	Description
start_x	ANY-NUMERICAL	x value of the starting location
start_y	ANY-NUMERICAL	y value of the starting location
[end_x]	ANY-NUMERICAL	x value of the ending location • When missing: start_x is used.
[end_y]	ANY-NUMERICAL	y value of the ending location • When missing: start_y is used.

Where:

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

Matrix SQL

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	Aggregate cost from start_vid to end_vid.

Return columns

```
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. • Value -2 indicates it is the summary row.
vehicle_id	BIGINT	Current vehicle identifier. • Sumary row has the **total capacity violations**. ◦ A capacity violation happens when overloading or underloading a vehicle.
stop_seq	INTEGER	Sequential value starting from **1** for the stops made by the current vehicle. The m_{th} stop of the current vehicle. • Sumary row has the **total time windows violations**. ◦ A time window violation happens when arriving after the location has closed.

stop_type	INTEGER	• Kind of stop location the vehicle is at
		  ◦ -1: at the solution summary row
		  ◦ 1: Starting location
		  ◦ 2: Pickup location
		  ◦ 3: Delivery location
		  ◦ 6: Ending location and indicates the vehicle's summary row

Column	Type	Description
order_id	BIGINT	Pickup-Delivery order pair identifier. • Value \(-1\): When no order is involved on the current stop location.
cargo	FLOAT	Cargo units of the vehicle when leaving the stop. • Value \(-1\) on solution summary row.
travel_time	FLOAT	Travel time from previous stop_seq to current stop_seq. • Summary has the **total traveling time**: ◦ The sum of all the travel_time.
arrival_time	FLOAT	Time spent waiting for current location to open. • \(-1\): at the solution summary row. • \(0\): at the starting location.
wait_time	FLOAT	Time spent waiting for current location to open. • Summary row has the **total waiting time**: ◦ The sum of all the wait_time.
service_time	FLOAT	Service duration at current location. • Summary row has the **total service time**: ◦ The sum of all the service_time.

departure_time	FLOAT	• The time at which the vehicle departs from the stop.
		  ◦ \(arrival_time + wait_time + service_time\).
		• The ending location has the **total time** used by the current vehicle.
		• Summary row has the **total solution time**:
		  ◦ \(total\ traveling\ time + total\ waiting\ time + total\ service\ time\).

Summary Row

Column	Type	Description
seq	INTEGER	Continues the sequence
vehicle_seq	INTEGER	Value \(-2\) indicates it is the summary row.
vehicle_id	BIGINT	**total capacity violations**: • A capacity violation happens when overloading or underloading a vehicle.
stop_seq	INTEGER	**total time windows violations**: • A time window violation happens when arriving after the location has closed.
stop_type	INTEGER	\(-1\)
order_id	BIGINT	\(-1\)
cargo	FLOAT	\(-1\)
travel_time	FLOAT	**total traveling 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	Summary row has the **total solution time**: • \(total\ traveling\ time + total\ waiting\ time + total\ service\ time\).

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 in *boxes*, a conversion of *kg of feathers* to *number of boxes* is needed.
- If the vehicle's **capacity** is measured in *kg*, a conversion of *box of apples* to *kg* is needed.

Showing how the 2 possible conversions can be done

Let: - \(f_boxes\): number of boxes needed 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 **pgr_pickDeliverEuclidean - Experimental**:
  - The vehicles have \((x, y)\) pairs for start and ending locations.
  - The orders Have \((x, y)\) pairs for pickup and delivery locations.
- When using **pgr_pickDeliver - Experimental**:
  - 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**. All time units have to be converted to a **0** reference and the same time units.

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.

Meaning of 0	time units	9:00 am	4:30 pm	10 min 30 secs
0:00 am	hours	9	16.5	\(10.5 / 60 = 0.175\)
0:00 am	minutes	\(9*60 = 54\)	\(16.5*60 = 990\)	10.5
9:00 am	hours	0	7.5	\(10.5 / 60 = 0.175\)
9:00 am	minutes	0	\(7.5*60 = 540\)	10.5

Factor handling

`factor` acts as a multiplier to convert from distance values to time units the matrix values or the euclidean values.

- When the values are already in the desired time units
  - `factor` should be **1**
  - When `factor` > 1 the travel times are faster
  - When `factor` < 1 the travel times are slower

For the **pgr_pickDeliverEuclidean - Experimental**:

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

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

For the **pgr_pickDeliver - Experimental**:

Given \(v = d / t\) therefore \(t = d / v\) And the `factor` becomes \(1 / v\)

Where:

v:
Velocity
d:
Distance
t:
Time

For the following equivalences $10m/s \approx 600m/min \approx 36 km/hr$

Working with time units in seconds and the matrix been in meters: For a 1000m lenght value on the matrix:

Units	velocity	Conversion	Factor	Result
seconds	$10 m/s$	$\frac{1}{10m/s}$	$0.1s/m$	$1000m * 0.1s/m = 100s$
minutes	$600 m/min$	$\frac{1}{600m/min}$	$0.0016min/m$	$1000m * 0.0016min/m = 1.6min$
Hours	$36 km/hr$	$\frac{1}{36 km/hr}$	$0.0277hr/km$	$1km * 0.0277hr/km = 0.0277hr$

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_dagShortestPath - Experimental**
- **pgr_edwardMoore - Experimental**
- **pgr_isPlanar - Experimental**
- **pgr_stoerWagner - Experimental**
- **pgr_topologicalSort - Experimental**
- **pgr_transitiveClosure - Experimental**
- **pgr_turnRestrictedPath - Experimental**
- **pgr_lengauerTarjanDominatorTree -Experimental**

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_bellmanFord - Experimental

pgr_bellmanFord — Shortest path(s) using Bellman-Ford 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

- Version 3.2.0
  - New **experimental** signature:
    - pgr_bellmanFord (**Combinations**)
- Version 3.0.0
  - New **experimental** signatures:
    - pgr_bellmanFord (**One to One**)
    - pgr_bellmanFord (**One to Many**)
    - pgr_bellmanFord (**Many to One**)
    - pgr_bellmanFord (**Many to Many**)

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. 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 \(\infty\).
- 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_bellmanFord(**Edges SQL**, **start vid**, **end vid**, [directed])
pgr_bellmanFord(**Edges SQL**, **start vid**, **end vids**, [directed])
pgr_bellmanFord(**Edges SQL**, **start vids**, **end vid** , [directed])
pgr_bellmanFord(**Edges SQL**, **start vids**, **end vids**, [directed])
pgr_bellmanFord(**Edges SQL**, **Combinations SQL**, [directed])

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

One to One

pgr_bellmanFord(**Edges SQL**, **start vid**, **end vid**, [directed])

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

Example:

From vertex \(6\) to vertex \(10\) on a **directed** graph

```
SELECT * FROM pgr_bellmanFord(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  6, 10, true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 rows)
```

One to Many

pgr_bellmanFord(**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 \(6\) to vertices \(\{ 10, 17\}\) on a **directed** graph

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

Many to One

pgr_bellmanFord(**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 \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

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

Many to Many

pgr_bellmanFord(**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 \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

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

Combinations

```
pgr_bellmanFord(Edges SQL, Combinations SQL, [directed])

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

Example:
Using a combinations table on an **undirected** graph.

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_bellmanFord(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    2 |    1 |        1
   5 |        3 |         5 |      10 |   10 |   -1 |    0 |        2
   6 |        1 |         6 |       5 |    6 |    1 |    1 |        0
   7 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
   8 |        1 |         6 |      15 |    6 |    2 |    1 |        0
   9 |        2 |         6 |      15 |   10 |    3 |    1 |        1
  10 |        3 |         6 |      15 |   15 |   -1 |    0 |        2
(10 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below

Column	Type	Description
Combinations SQL	TEXT	**Combinations SQL** 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.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Return columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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.

Column	Type	Description
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_vid to node.

Additional Examples

Example 1:
Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:
Making **start vids** the same as **end vids**.

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

Example 3:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_bellmanFord(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

See Also

- https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
- Sample Data

Indices and tables

- Index
- Search Page

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

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 3.2.0
  - New **experimental** function:
    - pgr_dagShortestPath(Combinations)
- Version 3.0.0
  - New **experimental** function

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**, **start vid**, **end vid**)
pgr_dagShortestPath(**Edges SQL**, **start vid**, **end vids**)
pgr_dagShortestPath(**Edges SQL**, **start vids**, **end vid**)
pgr_dagShortestPath(**Edges SQL**, **start vids**, **end vids**)
pgr_dagShortestPath(**Edges SQL**, **Combinations SQL**)

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

One to One

pgr_dagShortestPath(**Edges SQL**, **start vid**, **end vid**)

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

Example:
From vertex \(5\) to vertex \(11\) on a **directed** graph

```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edges',
  5, 11);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    5 |    1 |    1 |        0
   2 |        2 |    6 |    4 |    1 |        1
   3 |        3 |    7 |    8 |    1 |        2
   4 |        4 |   11 |   -1 |    0 |        3
(4 rows)
```

One to Many

pgr_dagShortestPath(**Edges SQL**, **start vid**, **end vids**)

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

```
OR EMPTY SET
```

Example:
From vertex \(5\) to vertices \(\{7, 11\}\)

```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edges',
  5, ARRAY[7, 11]);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    5 |    1 |    1 |        0
   2 |        2 |    6 |    4 |    1 |        1
   3 |        3 |    7 |   -1 |    0 |        2
   4 |        1 |    5 |    1 |    1 |        0
   5 |        2 |    6 |    4 |    1 |        1
   6 |        3 |    7 |    8 |    1 |        2
   7 |        4 |   11 |   -1 |    0 |        3
(7 rows)
```

Many to One

```
pgr_dagShortestPath(Edges SQL, start vids, end vid)

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

Example:
From vertices \(\{5, 10\}\) to vertex \(11\)

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

Many to Many

```
pgr_dagShortestPath(Edges SQL, start vids, end vids)

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

Example:
From vertices \(\{5, 15\}\) to vertices \(\{11, 17\}\) on an **undirected** graph

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

Combinations

```
pgr_dagShortestPath(Edges SQL, Combinations)

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

Example:
Using a combinations table on an**undirected** graph

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edges',
  'SELECT source, target FROM combinations');
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    5 |    1 |    1 |        0
   2 |        2 |    6 |   -1 |    0 |        1
(2 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Resturn Columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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_vid to node.

Additional Examples

Example 1:
Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:
Making **start_vids** the same as **end_vids**

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

Example 3:
Manually assigned vertex combinations.

```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |   -1 |    0 |        1
(2 rows)
```

See Also

- **Sample Data**
- **https://en.wikipedia.org/wiki/Topological_sorting**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2**

pgr_edwardMoore - Experimental

pgr_edwardMoore — Returns the shortest path using Edward-Moore 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 3.2.0
  - New **experimental** signature:
    - pgr_edwardMoore (**Combinations**)
- Version 3.0.0
  - New **experimental** signatures:
    - pgr_edwardMoore (**One to One**)
    - pgr_edwardMoore (**One to Many**)
    - pgr_edwardMoore (**Many to One**)
    - pgr_edwardMoore (**Many to Many**)

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

Summary

```
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])
pgr_edwardMoore(Edges SQL, Combinations SQL, [directed])

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

One to One

```
pgr_edwardMoore(Edges SQL, start vid, end vid, [directed])

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

Example:
From vertex \(6\) to vertex \(10\) on a **directed** graph

```
SELECT * FROM pgr_edwardMoore(
 'SELECT id, source, target, cost, reverse_cost FROM edges',
 6, 10, true);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    6 |    4 |    1 |        0
   2 |        2 |    7 |    8 |    1 |        1
   3 |        3 |   11 |    9 |    1 |        2
   4 |        4 |   16 |   16 |    1 |        3
   5 |        5 |   15 |    3 |    1 |        4
   6 |        6 |   10 |   -1 |    0 |        5
(6 rows)
```

One to Many

```
pgr_edwardMoore(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 \(6\) to vertices \(\{ 10, 17\}\) on a **directed** graph

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

Many to One

pgr_edwardMoore(**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 \(\{6, 1\}\) to vertex \(17\) on a **directed** graph

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

Many to Many

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

Example:
From vertices \(\{6, 1\}\) to vertices \(\{10, 17\}\) on an **undirected** graph

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

Combinations

pgr_edwardMoore(**Edges SQL**, **Combinations SQL**, [directed])

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

Example:
Using a combinations table on an**undirected** graph.

The combinations table:

```
SELECT source, target FROM combinations;
 source | target
--------+--------
      5 |      6
      5 |     10
      6 |      5
      6 |     15
      6 |     14
(5 rows)
```

The query:

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT source, target FROM combinations',
  false);
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         5 |       6 |    5 |    1 |    1 |        0
   2 |        2 |         5 |       6 |    6 |   -1 |    0 |        1
   3 |        1 |         5 |      10 |    5 |    1 |    1 |        0
   4 |        2 |         5 |      10 |    6 |    2 |    1 |        1
   5 |        3 |         5 |      10 |   10 |   -1 |    0 |        2
   6 |        1 |         6 |       5 |    6 |    1 |    1 |        0
   7 |        2 |         6 |       5 |    5 |   -1 |    0 |        1
   8 |        1 |         6 |      15 |    6 |    2 |    1 |        0
   9 |        2 |         6 |      15 |   10 |    3 |    1 |        1
  10 |        3 |         6 |      15 |   15 |   -1 |    0 |        2
(10 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below
Combinations SQL	TEXT	**Combinations SQL** 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.

Column	Type	Description
end vids	ARRAY[BIGINT]	Array of identifiers of ending vertices.

Optional parameters

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

Inner Queries

Edges SQL

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)
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

Combinations SQL

Parameter	Type	Description
source	**ANY-INTEGER**	Identifier of the departure vertex.
target	**ANY-INTEGER**	Identifier of the arrival vertex.

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Return columns

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

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
path_seq	INTEGER	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_vid to node.

Additional Examples

Example 1:

Demonstration of repeated values are ignored, and result is sorted.

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

Example 2:

Making **start vids** the same as **end vids**.

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

Example 3:

Manually assigned vertex combinations.

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edges',
  'SELECT * FROM (VALUES (6, 10), (6, 7), (12, 10)) AS combinations (source, target)');
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
   1 |        1 |         6 |       7 |    6 |    4 |    1 |        0
   2 |        2 |         6 |       7 |    7 |   -1 |    0 |        1
   3 |        1 |         6 |      10 |    6 |    4 |    1 |        0
   4 |        2 |         6 |      10 |    7 |    8 |    1 |        1
   5 |        3 |         6 |      10 |   11 |    9 |    1 |        2
   6 |        4 |         6 |      10 |   16 |   16 |    1 |        3
   7 |        5 |         6 |      10 |   15 |    3 |    1 |        4
   8 |        6 |         6 |      10 |   10 |   -1 |    0 |        5
   9 |        1 |        12 |      10 |   12 |   13 |    1 |        0
  10 |        2 |        12 |      10 |   17 |   15 |    1 |        1
  11 |        3 |        12 |      10 |   16 |   16 |    1 |        2
  12 |        4 |        12 |      10 |   15 |    3 |    1 |        3
  13 |        5 |        12 |      10 |   10 |   -1 |    0 |        4
(13 rows)
```

See Also

- **Sample Data**
- **https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2**

`pgr_isPlanar` - Experimental

`pgr_isPlanar` — Returns a boolean depending upon the planarity of the 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 3.2.0
  - New **experimental** function

Description

A graph is planar if it can be drawn in two-dimensional space with no two of its edges crossing. Such a drawing of a planar graph is called a plane drawing. Every planar graph also admits a straight-line drawing, which is a plane drawing where each edge is represented by a line segment. When a graph has \(K_5\) or \(K_{3, 3}\) as subgraph then the graph is not planar.

The main characteristics are:

- This implementation use the Boyer-Myrvold Planarity Testing.
- It will return a boolean value depending upon the planarity of the graph.
- Applicable only for **undirected** graphs.
- The algorithm does not considers traversal costs in the calculations.
- Running time: \(O(|V|)\)

Signatures

Summary

```
pgr_isPlanar(Edges SQL)

RETURNS BOOLEAN
```

```
SELECT * FROM pgr_isPlanar(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges'
);
 pgr_isplanar
--------------
 t
(1 row)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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 a boolean (pgr_isplanar)

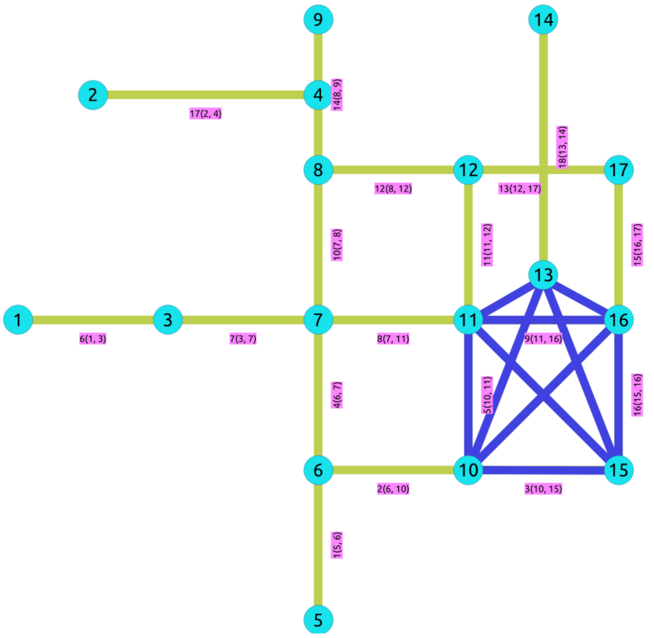
Column	Type	Description
pgr_isplanar	BOOLEAN	• *true* when the graph is planar. • *false* when the graph is not planar.

Additional Examples

The following edges will make the subgraph with vertices {10, 15, 11, 16, 13} a\(K_1\) graph.

```
INSERT INTO edges (source, target, cost, reverse_cost) VALUES
  (10, 16, 1, 1), (10, 13, 1, 1),
  (15, 11, 1, 1), (15, 13, 1, 1),
  (11, 13, 1, 1), (16, 13, 1, 1);
INSERT 0 6
```

The new graph is not planar because it has a\(K_5\) subgraph. Edges in blue represent\(K_5\) subgraph.




```
SELECT * FROM pgr_isPlanar(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges');
 pgr_isplanar
--------------
 f
(1 row)
```

See Also

- **Sample Data**
- **https://www.boost.org/libs/graph/doc/boyer_myrvold.html**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1) 3.0**

pgr_stoerWagner - Experimental

`pgr_stoerWagner` — The min-cut of graph using stoerWagner 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

- Version 3.0
  - New **Experimental** function

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:
min cut of the main subgraph

```
SELECT * FROM pgr_stoerWagner(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges WHERE id < 17');
 seq | edge | cost | mincut
-----+------+------+--------
   1 |    6 |    1 |      1
(1 row)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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:

Example:
min cut of an edge

```
SELECT * FROM pgr_stoerWagner(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges WHERE id = 18');
 seq | edge | cost | mincut
-----+------+------+--------
   1 |   18 |    1 |      1
(1 row)
```

Example:
Using **pgr_connectedComponents**

```
SELECT * FROM pgr_stoerWagner(
  $$
  SELECT id, source, target, cost, reverse_cost FROM edges
  WHERE source IN (
      SELECT node FROM pgr_connectedComponents(
      'SELECT id, source, target, cost, reverse_cost FROM edges ')
      WHERE component = 2)
  $$
);
 seq | edge | cost | mincut
-----+------+------+--------
   1 |   17 |    1 |      1
(1 row)
```

See Also

- **Sample Data**
- **https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner_algorithm**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_topologicalSort - Experimental

`pgr_topologicalSort` — Linear ordering of the vertices for directed acyclic graphs (DAG).

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

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.

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:
Topologically sorting the graph

```
SELECT * FROM pgr_topologicalsort(
  $$SELECT id, source, target, cost
  FROM edges WHERE cost >= 0
  UNION
  SELECT id, target, source, reverse_cost
  FROM edges WHERE cost < 0$$);
 seq | sorted_v
-----+----------
   1 |        1
   2 |        5
   3 |        2
   4 |        4
   5 |        3
   6 |       13
   7 |       14
   8 |       15
   9 |       10
  10 |        6
  11 |        7
  12 |        8
  13 |        9
  14 |       11
  15 |       16
  16 |       12
  17 |       17
(17 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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	INTEGER	Sequential value starting from \(1\)
sorted_v	BIGINT	Linear topological ordering of the vertices

Additional examples

Example:
Topologically sorting the one way segments

```
SELECT * FROM pgr_topologicalsort(
  $$SELECT id, source, target, cost, -1 AS reverse_cost
  FROM edges WHERE cost >= 0
  UNION
  SELECT id, source, target, -1, reverse_cost
  FROM edges WHERE cost < 0$$);
 seq | sorted_v
-----+----------
   1 |        5
   2 |        2
   3 |        4
   4 |       13
   5 |       14
   6 |        1
   7 |        3
   8 |       15
   9 |       10
  10 |        6
  11 |        7
  12 |        8
  13 |        9
  14 |       11
  15 |       12
  16 |       16
  17 |       17
(17 rows)
```

Example:
Graph is not a DAG

```
SELECT * FROM pgr_topologicalsort(
  $$SELECT id, source, target, cost, reverse_cost FROM edges$$);
ERROR:  The graph must be a DAG.
HINT:  Working with Directed Graph

CONTEXT:  SQL function "pgr_topologicalsort" statement 1
```

See Also

- **Sample Data**
- **https://en.wikipedia.org/wiki/Topological_sorting**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

`pgr_transitiveClosure` - Experimental

`pgr_transitiveClosure` — Transitive closure graph of a directed 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

Description

Transforms the input directed graph into the transitive closure of the graph.

The main characteristics are:

- Process is valid for directed graphs.
  - The transitive closure of an undirected graph produces a cluster graph
  - Reachability between vertices on an undirected graph happens when they belong to the same connected component. (see **pgr_connectedComponents**)
- The returned values are not ordered
- The returned graph is compresed
- Running time: \(O(|V||E|)\)

Signatures

Summary

The pgr_transitiveClosure function has the following signature:

```
pgr_transitiveClosure(Edges SQL)

RETURNS SET OF (seq, vid, target_array)
```

Example:
Rechability of a subgraph

```
SELECT * FROM pgr_transitiveclosure(
  'SELECT id, source, target, cost, reverse_cost
  FROM edges WHERE id IN (2, 3, 5, 11, 12, 13, 15)')
ORDER BY vid;
 seq | vid |    target_array
-----+-----+--------------------
   1 |   6 | {}
   6 |   8 | {12,17,16}
   2 |  10 | {12,17,16,11,6}
   4 |  11 | {12,17,16}
   5 |  12 | {17,16}
   3 |  15 | {12,17,16,10,11,6}
   8 |  16 | {17,16}
   7 |  17 | {17,16}
(8 rows)
```

Parameters

Parameter	Type	Description
Edges SQL	TEXT	**Edges SQL** as described below.

Inner Queries

Edges SQL

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)
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, vid, target_array)

Column	Type	Description
seq	INTEGER	Sequential value starting from \(1\)
vid	BIGINT	Identifier of the source of the edges
target_array	BIGINT	Identifiers of the targets of the edges • Identifiers of the vertices that are reachable from vertex v.

See Also

- **Sample Data**
- **https://en.wikipedia.org/wiki/Transitive_closure**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2 3.1 3.0**

pgr_turnRestrictedPath - Experimental

pgr_turnRestrictedPath Using Yen's algorithm Vertex -Vertex routing with restrictions

> **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

Using Yen's algorithm to obtain K shortest paths and analyze the paths to select the paths that do not use the restrictions

Signatures

pgr_turnRestrictedPath(**Edges SQL**, **Restrictions SQL**, **start vid**, **end vid**, **K**, [**options**])
options: [directed, heap_paths, stop_on_first, strict]

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

Example:
From vertex \(3\) to vertex \(8\) on a directed graph

```
SELECT * FROM pgr_turnRestrictedPath(
  $$SELECT id, source, target, cost, reverse_cost FROM edges$$,
  $$SELECT path, cost FROM new_restrictions$$,
  3, 8, 3);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    3 |    7 |    1 | Infinity
   2 |       1 |        2 |    7 |   10 |    1 |        1
   3 |       1 |        3 |    8 |   -1 |    0 |        2
(3 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	SQL query as described.
start vid	**ANY-INTEGER**	Identifier of the departure vertex.
end vid	**ANY-INTEGER**	Identifier of the departure vertex.
K	**ANY-INTEGER**	Number of required paths

Where:

ANY-INTEGER:
SMALLINT, INTEGER, BIGINT

Optional parameters

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

KSP Optional parameters

Column	Type	Default	Description
heap_paths	BOOLEAN	false	• When false Returns at most K paths • When true all the calculated paths while processing are returned. • Roughly, when the shortest path has N edges, the heap will contain about than N * K paths for small value of K and K > 5.

Special optional parameters

Column	Type	Default	Description
stop_on_first	BOOLEAN	true	• When true stops on first path found that dos not violate restrictions • When false returns at most K paths
strict	BOOLEAN	false	• When true returns only paths that do not violate restrictions • When false returns the paths found

Inner Queries

Edges SQL

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)
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

Restrictions SQL

Column	Type	Description
path	ARRAY [**ANY-INTEGER**]	Sequence of edge identifiers that form a path that is not allowed to be taken. - Empty arrays or NULL arrays are ignored. - Arrays that have aNULL element will raise an exception.
Cost	**ANY-NUMERICAL**	Cost of taking the forbidden path.

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	INTEGER	Sequential value starting from **1**.
path_id	INTEGER	Path identifier. • Has value **1** for the first of a path fromstart_vid to end_vid.
path_seq	INTEGER	Relative position in the path. Has value**1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex.
end_vid	BIGINT	Identifier of the ending vertex.
node	BIGINT	Identifier of the node in the path fromstart_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go fromnode 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_vid to node.

Additional Examples

Example:
From vertex \(3\) to \(8\) with strict flag on.

No results because the only path available follows a restriction.

```
SELECT * FROM pgr_turnRestrictedPath(
  $$SELECT id, source, target, cost, reverse_cost FROM edges$$,
  $$SELECT path, cost FROM new_restrictions$$,
  3, 8, 3,
  strict => true);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
(0 rows)
```

Example:
From vertex \(3\) to vertex \(8\) on an undirected graph

```
SELECT * FROM pgr_turnRestrictedPath(
  $$SELECT id, source, target, cost, reverse_cost FROM edges$$,
  $$SELECT path, cost FROM new_restrictions$$,
  3, 8, 3,
  directed => false);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    3 |    7 |    1 |        0
   2 |       1 |        2 |    7 |    4 |    1 |        1
   3 |       1 |        3 |    6 |    2 |    1 |        2
   4 |       1 |        4 |   10 |    5 |    1 |        3
   5 |       1 |        5 |   11 |   11 |    1 |        4
   6 |       1 |        6 |   12 |   12 |    1 |        5
   7 |       1 |        7 |    8 |   -1 |    0 |        6
(7 rows)
```

Example:
From vertex \(3\) to vertex \(8\) with more alternatives

```
SELECT * FROM pgr_turnRestrictedPath(
  $$SELECT id, source, target, cost, reverse_cost FROM edges$$,
  $$SELECT path, cost FROM new_restrictions$$,
  3, 8, 3,
  directed => false,
  heap_paths => true,
  stop_on_first => false);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |    3 |    7 |    1 |        0
   2 |       1 |        2 |    7 |    4 |    1 |        1
   3 |       1 |        3 |    6 |    2 |    1 |        2
   4 |       1 |        4 |   10 |    5 |    1 |        3
   5 |       1 |        5 |   11 |   11 |    1 |        4
   6 |       1 |        6 |   12 |   12 |    1 |        5
   7 |       1 |        7 |    8 |   -1 |    0 |        6
   8 |       2 |        1 |    3 |    7 |    1 |        0
   9 |       2 |        2 |    7 |    8 |    1 |        1
  10 |       2 |        3 |   11 |    9 |    1 |        2
  11 |       2 |        4 |   16 |   15 |    1 |        3
  12 |       2 |        5 |   17 |   13 |    1 |        4
  13 |       2 |        6 |   12 |   12 |    1 |        5
  14 |       2 |        7 |    8 |   -1 |    0 |        6
(14 rows)
```

See Also

- **K shortest paths - Category**
- **Sample Data**

Indices and tables

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

- **Supported versions: Latest (3.3) 3.2**

pgr_lengauerTarjanDominatorTree -Experimental

`pgr_lengauerTarjanDominatorTree` — Returns the immediate dominator of all vertices.

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 3.2.0
  - New **experimental** function

Description

The algorithm calculates the *immidiate dominator* of each vertex called **idom**, once **idom** of each vertex is calculated then by making every **idom** of each vertex as its parent, the dominator tree can be built.

The main Characteristics are:

- The algorithm works in directed graph only.
- The returned values are not ordered.
- The algorithm returns *idom* of each vertex.
- If the *root vertex* not present in the graph then it returns empty set.
- Running time: $O((V+E)log(V+E))$

Signatures

Summary

```
pgr_lengauerTarjanDominatorTree(Edges SQL, root vertex)

RETURNS SET OF (seq, vertex_id, idom)
OR EMPTY SET
```

Example:
The dominator tree with root vertex 5

```
SELECT * FROM pgr_lengauertarjandominatortree(
  $$SELECT id,source,target,cost,reverse_cost FROM edges$$,
  5) ORDER BY vertex_id;
 seq | vertex_id | idom
-----+-----------+------
   1 |         1 |    2
   9 |         2 |    0
   2 |         3 |    3
  10 |         4 |    0
  17 |         5 |    0
   4 |         6 |   17
   3 |         7 |    4
   7 |         8 |    3
  11 |         9 |    7
   5 |        10 |   16
   6 |        11 |    3
   8 |        12 |    3
  12 |        13 |    0
  13 |        14 |    0
  16 |        15 |   15
  15 |        16 |    3
  14 |        17 |    3
(17 rows)
```

Parameters

Column	Type	Description
Edges SQL	TEXT	SQL query as described above.
root vertex	BIGINT	Identifier of the starting vertex.

Inner Queries

Edges SQL

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)
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, vertex_id, idom)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
vertex_id	BIGINT	Identifier of vertex .
idom	BIGINT	Immediate dominator of vertex.

Additional Examples

Example:
Dominator tree of another component.

```
SELECT * FROM pgr_lengauertarjandominatortree(
  $$SELECT id,source,target,cost,reverse_cost FROM edges$$,
  13) ORDER BY vertex_id;
 seq | vertex_id | idom
-----+-----------+------
   1 |         1 |    0
   9 |         2 |    0
   2 |         3 |    0
  10 |         4 |    0
  17 |         5 |    0
   4 |         6 |    0
   3 |         7 |    0
   7 |         8 |    0
  11 |         9 |    0
   5 |        10 |    0
   6 |        11 |    0
   8 |        12 |    0
  12 |        13 |    0
  13 |        14 |   12
  16 |        15 |    0
  15 |        16 |    0
  14 |        17 |    0
(17 rows)
```

See Also

- **Sample Data**
- **Boost: Lengauer-Tarjan dominator tree algorithm**
- **Wikipedia: dominator tree**

Indices and tables

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

See Also

Indices and tables

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

Release Notes

- **Supported versions: Latest** (**3.3**) **3.2 3.1 3.0**
- **Unsupported versions: 2.6 2.5 2.4 2.3 2.2 2.1 2.0**

Release Notes

To see the full list of changes check the list of **Git commits** on Github.

Contents

pgRouting 3.3.4 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.3.4**

Issue fixes

- **#2400**: pgRouting 3.3.3 does not build in focal

pgRouting 3.3.3 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.3.3**

Issue fixes

- **#1891**: pgr_ksp doesn’t give all correct shortest path

Official functions changes

- Flow functions
  - `pgr_maxCardinalityMatch(text,boolean)`
    - Ignoring optional boolean parameter, as the algorithm works only for undirected graphs.

pgRouting 3.3.2 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.3.2**

- Revised documentation
  - Simplifying table names and table columns, for example:
    - `edges` instead of `edge_table`
      - Removing unused columns `category_id` and `reverse_category_id`.
    - `combinations` instead of `combinations_table`
      - Using PostGIS standard for geometry column.
        - `geom` instead of `the_geom`
  - Avoiding usage of functions that modify indexes, columns etc on tables.
    - Using `pgr_extractVertices` to create a routing topology
  - Restructure of the pgRouting concepts page.

Issue fixes

- **#2276**: edgeDisjointPaths issues with start_vid and combinations
- **#2312**: pgr_extractVertices error when target is not BIGINT
- **#2357**: Apply clang-tidy performance-*

pgRouting 3.3.1 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.3.1** on Github.

Issue fixes

- **#2216**: Warnings when using clang

- **#2266**: Error processing restrictions

pgRouting 3.3.0 Release Notes

To see all issues & pull requests closed by this release see the **Git closed milestone for 3.3.0** on Github.

Issue fixes

- **#2057**: trspViaEdges columns in different order
- **#2087**: pgr_extractVertices to proposed
- **#2201**: pgr_depthFirstSearch to proposed
- **#2202**: pgr_sequentialVertexColoring to proposed
- **#2203**: pgr_dijkstraNear and pgr_dijkstraNearCost to proposed

New experimental functions

- Coloring
  - pgr_edgeColoring

Experimental promoted to Proposed

- Dijkstra
  - pgr_dijkstraNear
    - pgr_dijkstraNear(Combinations)
    - pgr_dijkstraNear(Many to Many)
    - pgr_dijkstraNear(Many to One)
    - pgr_dijkstraNear(One to Many)
  - pgr_dijkstraNearCost
    - pgr_dijkstraNearCost(Combinations)
    - pgr_dijkstraNearCost(Many to Many)
    - pgr_dijkstraNearCost(Many to One)
    - pgr_dijkstraNearCost(One to Many)
- Coloring
  - pgr_sequentialVertexColoring
- Topology
  - pgr_extractVertices
- Traversal
  - pgr_depthFirstSearch(Multiple vertices)
  - pgr_depthFirstSearch(Single vertex)

pgRouting 3.2.2 Release Notes

To see all issues & pull requests closed by this release see the **Git closed milestone for 3.2.2** on Github.

Issues

- **#2093**: Compilation on Visual Studio
- **#2189**: Build error on RHEL 7

pgRouting 3.2.1 Release Notes

To see all issues & pull requests closed by this release see the **Git closed milestone for 3.2.1** on Github.

Issue fixes

- **#1883**: pgr_TSPEuclidean crashes connection on Windows
  - The solution is to use Boost::graph::metric_tsp_approx
  - To not break user’s code the optional parameters related to the TSP Annaeling are ignored
  - The function with the annaeling optional parameters is deprecated

pgRouting 3.2.0 Release Notes

To see all issues & pull requests closed by this release see the **Git closed milestone for 3.2.0** on Github.

Build

- **#1850**: Change Boost min version to 1.56

    - Removing support for Boost v1.53, v1.54 & v1.55

New experimental functions

- pgr_bellmanFord(Combinations)
- pgr_binaryBreadthFirstSearch(Combinations)
- pgr_bipartite
- pgr_dagShortestPath(Combinations)
- pgr_depthFirstSearch
- Dijkstra Near
    - pgr_dijkstraNear
        - pgr_dijkstraNear(One to Many)
        - pgr_dijkstraNear(Many to One)
        - pgr_dijkstraNear(Many to Many)
        - pgr_dijkstraNear(Combinations)
    - pgr_dijkstraNearCost
        - pgr_dijkstraNearCost(One to Many)
        - pgr_dijkstraNearCost(Many to One)
        - pgr_dijkstraNearCost(Many to Many)
        - pgr_dijkstraNearCost(Combinations)
- pgr_edwardMoore(Combinations)
- pgr_isPlanar
- pgr_lengauerTarjanDominatorTree
- pgr_makeConnected
- Flow
    - pgr_maxFlowMinCost(Combinations)
    - pgr_maxFlowMinCost_Cost(Combinations)
- pgr_sequentialVertexColoring

New proposed functions

- Astar
    - pgr_aStar(Combinations)
    - pgr_aStarCost(Combinations)
- Bidirectional Astar
    - pgr_bdAstar(Combinations)
    - pgr_bdAstarCost(Combinations)
- Bidirectional Dijkstra
    - pgr_bdDijkstra(Combinations)
    - pgr_bdDijkstraCost(Combinations)
- Flow
    - pgr_boykovKolmogorov(Combinations)
    - pgr_edgeDisjointPaths(Combinations)
    - pgr_edmondsKarp(Combinations)
    - pgr_maxFlow(Combinations)
    - pgr_pushRelabel(Combinations)
- pgr_withPoints(Combinations)
- pgr_withPointsCost(Combinations)

pgRouting 3.1.4 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.1.4** on Github.

Issues fixes

- **#2189**: Build error on RHEL 7

pgRouting 3.1.3 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.1.3** on Github.

Issues fixes

- **#1825**: Boost versions are not honored
- **#1849**: Boost 1.75.0 geometry “point_xy.hpp” build error on macOS environment
- **#1861**: vrp functions crash server

pgRouting 3.1.2 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.1.2** on Github.

Issues fixes

- **#1304**: FreeBSD 12 64-bit crashes on pgr_vrOneDepot tests Experimental Function
- **#1356**: tools/testers/pg_prove_tests.sh fails when PostgreSQL port is not passed
- **#1725**: Server crash on pgr_pickDeliver and pgr_vrpOneDepot on openbsd
- **#1760**: TSP server crash on ubuntu 20.04 #1760
- **#1770**: Remove warnings when using clang compiler

pgRouting 3.1.1 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.1.1** on Github.

Issues fixes

- **#1733**: pgr_bdAstar fails when source or target vertex does not exist in the graph
- **#1647**: Linear Contraction contracts self loops
- **#1640**: pgr_withPoints fails when points_sql is empty
- **#1616**: Path evaluation on C++ not updated before the results go back to C
- **#1300**: pgr_chinesePostman crash on test data

pgRouting 3.1.0 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.1.0** on Github.

New proposed functions

- pgr_dijkstra(combinations)
- pgr_dijkstraCost(combinations)

Build changes

- Minimal requirement for Sphinx: version 1.8

pgRouting 3.0.6 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.0.6** on Github.

Issues fixes

- **#2189**: Build error on RHEL 7

pgRouting 3.0.5 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.0.5** on Github.

Backport issues fixes

- **#1825**: Boost versions are not honored
- **#1849**: Boost 1.75.0 geometry “point_xy.hpp” build error on macOS environment
- **#1861**: vrp functions crash server

pgRouting 3.0.4 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.0.4** on Github.

Backport issues fixes

- **#1304**: FreeBSD 12 64-bit crashes on pgr_vrOneDepot tests Experimental Function
- **#1356**: tools/testers/pg_prove_tests.sh fails when PostgreSQL port is not passed
- **#1725**: Server crash on pgr_pickDeliver and pgr_vrpOneDepot on openbsd
- **#1760**: TSP server crash on ubuntu 20.04 #1760
- **#1770**: Remove warnings when using clang compiler

pgRouting 3.0.3 Release Notes

Backport issues fixes

- **#1733**: pgr_bdAstar fails when source or target vertex does not exist in the graph
- **#1647**: Linear Contraction contracts self loops
- **#1640**: pgr_withPoints fails when points_sql is empty
- **#1616**: Path evaluation on C++ not updated before the results go back to C
- **#1300**: pgr_chinesePostman crash on test data

pgRouting 3.0.2 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.0.2** on Github.

Issues fixes

- **#1378**: Visual Studio build failing

pgRouting 3.0.1 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.0.1** on Github.

Issues fixes

- **#232**: Honor client cancel requests in C /C++ code

pgRouting 3.0.0 Release Notes

To see all issues & pull requests closed by this release see the**Git closed milestone for 3.0.0** on Github.

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 experimental 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 Release Notes** 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
- Compilation problems solved

Changes for release 1.01

- Shooting* bug fixes

Changes for release 1.0

- Core and extra functions are separated
- Cmake build process
- Bug fixes

Changes for release 1.0.0b

- Additional SQL file with more simple names for wrapper functions
- Bug fixes

Changes for release 1.0.0a

- Shooting* shortest path algorithm for real road networks
- Several SQL bugs were fixed

Changes for release 0.9.9

- PostgreSQL 8.2 support
- Shortest path functions return empty result if they could not find any path

Changes for release 0.9.8

- Renumbering scheme was added to shortest path functions
- Directed shortest path functions were added
- routing_postgis.sql was modified to use dijkstra in TSP search

Indices and tables

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