



## Table of Contents

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

This is the manual for pgRouting v3.0.5.



The pgRouting Manual is licensed under a **Creative Commons Attribution-Share Alike 3.0 License**. Feel free to use this material any way you like, but we ask that you attribute credit to the pgRouting Project and wherever possible, a link back to <https://pgrouting.org>. For other licenses used in pgRouting see the **Licensing** page.

## General

### Introduction

pgRouting is an extension of **PostGIS** and **PostgreSQL** geospatial database and adds routing and other network analysis functionality. A predecessor of pgRouting - pgDijkstra, written by Sylvain Pasche from **Camptocamp**, was later extended by **Orkney** and renamed to pgRouting. The project is now supported and maintained by **Georepublic**, **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 <b>GNU General Public License v2.0 or later</b> .
Boost Software License - Version 1.0	Some Boost extensions are available under <b>Boost Software License - Version 1.0</b> .
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 <b>Creative Commons Attribution-Share Alike 3.0 License</b> .

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

### Contributors

#### This Release Contributors

##### Individuals (in alphabetical order)

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

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

##### Corporate Sponsors (in alphabetical order)

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

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

#### Contributors Past & Present:

##### Individuals (in alphabetical order)

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

#### Corporate Sponsors (in alphabetical order)

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

- Camptocamp
- CSIS (University of Tokyo)
- Georepublic
- Google Summer of Code
- iMaptools
- 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 can be found at <https://github.com/pgRouting/pgrouting/wiki/Migration-Guide>.

## Installation

### Table of Contents

- **Short Version**
- **Get the sources**
- **Enabling and upgrading in the database**
- **Dependencies**
- **Configuring**
- **Building**
- **Testing**

Instructions for downloading and installing binaries for different Operative systems instructions and additional notes and corrections not included in this documentation can be found in [Installation wiki](#)

To use pgRouting postGIS needs to be installed, please read the information about installation in this [Install Guide](#)

### Short Version

Extracting the tar ball

```
tar xvfz pgrouting-3.0.5.tar.gz
cd pgrouting-3.0.5
```

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

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

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

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

### Get the sources

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

## wget

To download this release:

```
wget -O pgrouting-3.0.5.tar.gz https://github.com/pgRouting/pgrouting/archive/v3.0.5.tar.gz
```

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

## git

To download the repository

```
git clone git://github.com/pgRouting/pgrouting.git
cd pgrouting
git checkout v3.0.5
```

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

Enabling and upgrading in the database

### Enabling the database

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

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

### Upgrading the database

To upgrade pgRouting in the database to version 3.0.5 use the following command:

```
ALTER EXTENSION pgrouting UPDATE TO "3.0.5";
```

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

### optional dependencies

For user's documentation

- Sphinx >= 1.1
- Latex

For developer's documentation

- Doxygen >= 1.7

For testing

- pgtap
- pg\_prove

For using:

- PostGIS version >= 2.2

### Example: Installing dependencies on linux

Installing the compilation dependencies

## Database dependencies

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

## Build dependencies

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

## Optional dependencies

For documentation and testing

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

## Configuring

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

The build directory is different from the source directory

Create the build directory

```
$ mkdir build
```

## Configurable variables

### To see the variables that can be configured

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

## Configuring The Documentation

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

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

Configuring with documentation

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



### Note

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

## Building

Using `make` to build the code and the documentation

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

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

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



### Warning

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

## MinGW on Windows

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

## Linux

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

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

When the configuration changes:

```
rm -rf build
```

and start the build process as mentioned above.

## Testing

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

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

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

## See Also

### Indices and tables

- [Index](#)
- [Search Page](#)

## Support

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

## Reporting Problems

Bugs are reported and managed in an [issue tracker](#). Please follow these steps:

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

- appears.
- For the versions where you can replicate the problem, note the operating system and version of pgRouting, PostGIS and PostgreSQL.
  - 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	<a href="https://georepublic.info">https://georepublic.info</a>
Paragon Corporation	United States	<a href="https://www.paragoncorporation.com">https://www.paragoncorporation.com</a>
Camptocamp	Switzerland, France	<a href="https://www.camptocamp.com">https://www.camptocamp.com</a>
Netlab	Capranica, Italy	<a href="https://www.osgeo.org/service-providers/netlab/">https://www.osgeo.org/service-providers/netlab/</a>

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

### Sample Data

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

#### Create table

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

#### Insert data

```

INSERT INTO edge_table (
  category_id, reverse_category_id,
  cost, reverse_cost,
  capacity, reverse_capacity,
  x1, y1,
  x2, y2) VALUES
(3, 1, 1, 1, 80, 130, 2, 0, 2, 1),
(3, 2, -1, 1, -1, 100, 2, 1, 3, 1),
(2, 1, -1, 1, -1, 130, 3, 1, 4, 1),
(2, 4, 1, 1, 100, 50, 2, 1, 2, 2),
(1, 4, 1, -1, 130, -1, 3, 1, 3, 2),
(4, 2, 1, 1, 50, 100, 0, 2, 1, 2),
(4, 1, 1, 1, 50, 130, 1, 2, 2, 2),
(2, 1, 1, 1, 100, 130, 2, 2, 3, 2),
(1, 3, 1, 1, 130, 80, 3, 2, 4, 2),
(1, 4, 1, 1, 130, 50, 2, 2, 2, 3),
(1, 2, 1, -1, 130, -1, 3, 2, 3, 3),
(2, 3, 1, -1, 100, -1, 2, 3, 3, 3),
(2, 4, 1, -1, 100, -1, 3, 3, 4, 3),
(3, 1, 1, 1, 80, 130, 2, 3, 2, 4),
(3, 4, 1, 1, 80, 50, 4, 2, 4, 3),
(3, 3, 1, 1, 80, 80, 4, 1, 4, 2),
(1, 2, 1, 1, 130, 100, 0.5, 3.5, 1.9999999999999999, 3.5),
(4, 1, 1, 1, 50, 130, 3.5, 2.3, 3.5, 4);

```

## Updating geometry

```

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

```

## Topology

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

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

## Points of interest

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

```

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

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

UPDATE pointsOfInterest SET the_geom = st_makePoint(x,y);

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

```

## Restrictions

- Used with the **pgr\_trsp - Turn Restriction Shortest Path (TRSP)** functions.

```

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

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

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

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

```

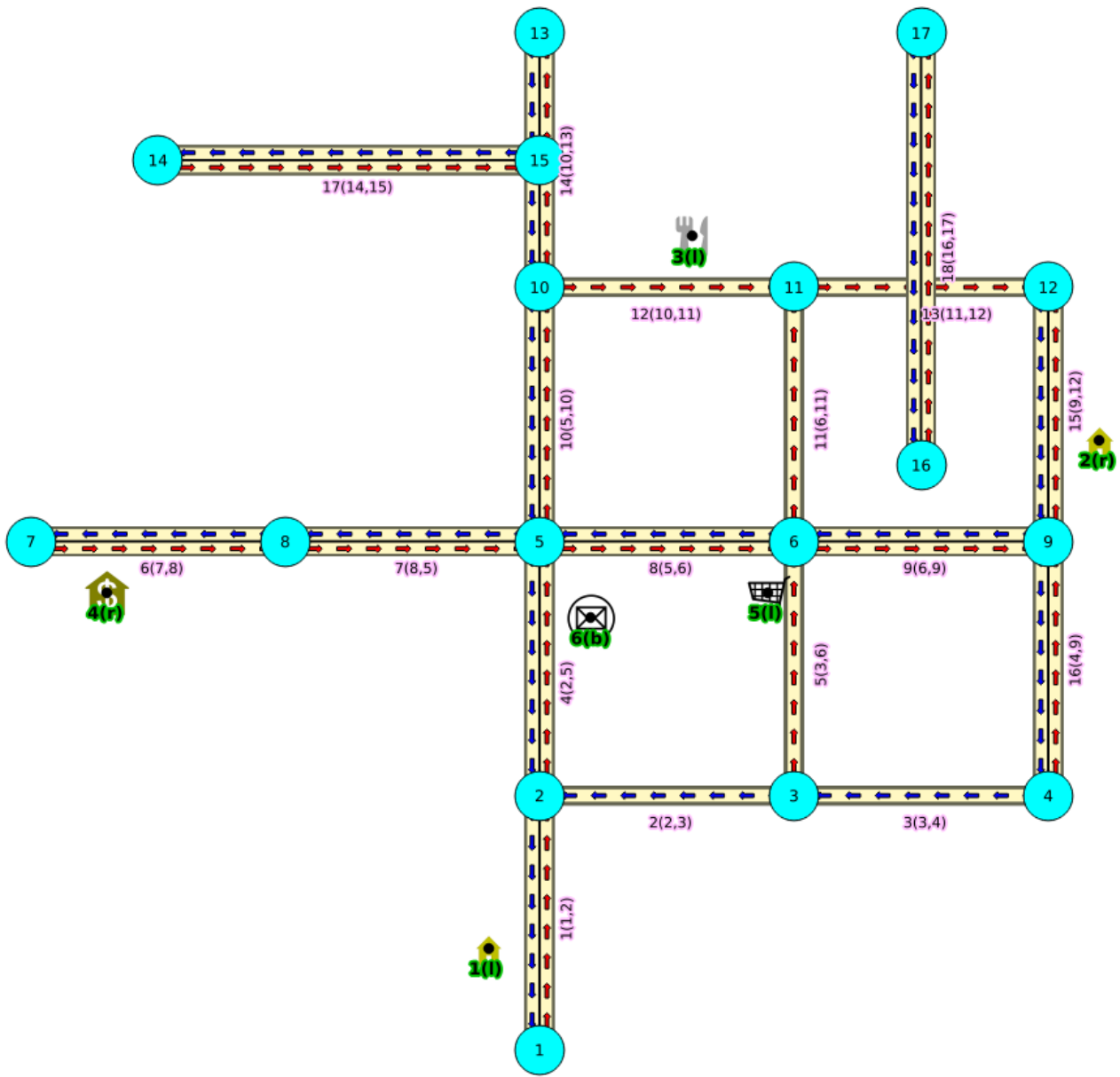
## Images

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

**Network for queries marked as directed and cost and reverse\_cost columns are used**

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

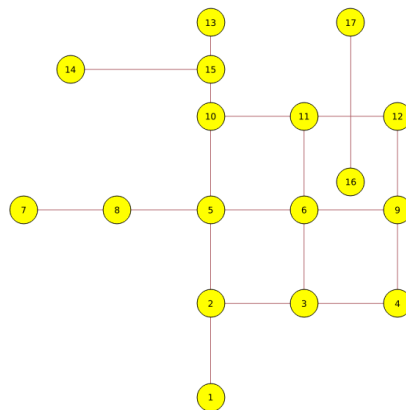




Graph 1: Directed, with cost and reverse cost

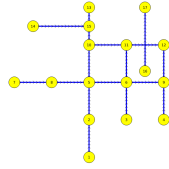
Network for queries marked as undirected and cost and reverse\_cost columns are used

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



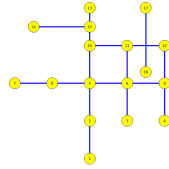
Graph 2: Undirected, with cost and reverse cost

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



Graph 3: Directed, with cost

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



Graph 4: Undirected, with cost

Pick & Deliver Data

```

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

INSERT INTO customer(
  id, x, y, demand, opentime, closetime, servicetime, pindex, dindex) VALUES
( 0, 40, 50, 0, 0, 1236, 0, 0, 0),
( 1, 45, 68, -10, 912, 967, 90, 11, 0),
( 2, 45, 70, -20, 825, 870, 90, 6, 0),
( 3, 42, 66, 10, 65, 146, 90, 0, 75),
( 4, 42, 68, -10, 727, 782, 90, 9, 0),
( 5, 42, 65, 10, 15, 67, 90, 0, 7),
( 6, 40, 69, 20, 621, 702, 90, 0, 2),
( 7, 40, 66, -10, 170, 225, 90, 5, 0),
( 8, 38, 68, 20, 255, 324, 90, 0, 10),
( 9, 38, 70, 10, 534, 605, 90, 0, 4),
(10, 35, 66, -20, 357, 410, 90, 8, 0),
(11, 35, 69, 10, 448, 505, 90, 0, 1),
(12, 25, 85, -20, 652, 721, 90, 18, 0),
(13, 22, 75, 30, 30, 92, 90, 0, 17),
(14, 22, 85, -40, 567, 620, 90, 16, 0),
(15, 20, 80, -10, 384, 429, 90, 19, 0),
(16, 20, 85, 40, 475, 528, 90, 0, 14),
(17, 18, 75, -30, 99, 148, 90, 13, 0),
(18, 15, 75, 20, 179, 254, 90, 0, 12),
(19, 15, 80, 10, 278, 345, 90, 0, 15),
(20, 30, 50, 10, 10, 73, 90, 0, 24),
(21, 30, 52, -10, 914, 965, 90, 30, 0),
(22, 28, 52, -20, 812, 883, 90, 28, 0),
(23, 28, 55, 10, 732, 777, 0, 0, 103),
(24, 25, 50, -10, 65, 144, 90, 20, 0),
(25, 25, 52, 40, 169, 224, 90, 0, 27),
(26, 25, 55, -10, 622, 701, 90, 29, 0),
(27, 23, 52, -40, 261, 316, 90, 25, 0),
(28, 23, 55, 20, 546, 593, 90, 0, 22),
(29, 20, 50, 10, 358, 405, 90, 0, 26),
(30, 20, 55, 10, 449, 504, 90, 0, 21),
(31, 10, 35, -30, 200, 237, 90, 32, 0),
(32, 10, 40, 30, 31, 100, 90, 0, 31),
(33, 8, 40, 40, 87, 158, 90, 0, 37),
(34, 8, 45, -30, 751, 816, 90, 38, 0),
(35, 5, 35, 10, 283, 344, 90, 0, 39),
(36, 5, 45, 10, 665, 716, 0, 0, 105),
(37, 2, 40, -40, 383, 434, 90, 33, 0),
(38, 0, 40, 30, 479, 522, 90, 0, 34),
(39, 0, 45, -10, 567, 624, 90, 35, 0),
(40, 0, 50, 20, 659, 710, 90, 36, 0)

```

```

(40, 33, 30, -20, 204, 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);

```

## Pgrouing Concepts

### pgRouting Concepts

#### Contents

- **pgRouting Concepts**
  - **Getting Started**
    - **Create a routing Database**
    - **Load Data**
    - **Build a Routing Topology**
    - **Check the Routing Topology**
    - **Compute a Path**
  - **Group of Functions**
    - **One to One**
    - **One to Many**
    - **Many to One**

- **Many to Many**
- **Inner Queries**
  - **Description of the edges\_sql query for dijkstra like functions**
- **Parameters**
  - **edges\_sql query for aStar - Family of functions** and **aStar - Family of functions** functions
- **Return columns & values**
  - **Return values for a path**
  - **Return values for multiple paths from the same source and destination**
  - **Description of the return values for a Cost Matrix - Category** function
  - **Description of the Return Values**
- **Advanced Topics**
  - **Routing Topology**
  - **Graph Analytics**
  - **Analyze a Graph**
  - **Analyze One Way Streets**
    - **Example**
- **Performance Tips**
  - **For the Routing functions**
  - **For the topology functions:**
- **How to contribute**

## Getting Started

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

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

### Create a routing Database

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

For Postgresql 9.2 and later versions

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

### Load Data

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

**osm2pgrouting:**

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

**shp2pgsql:**

- this is the postgresql shapefile loader

**ogr2ogr:**

- this is a vector data conversion utility


**osm2pgsql:**

- this is a tool for loading OSM data into postgresql

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

### Build a Routing Topology

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


Note

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

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

- **pgr\_createTopology**

#### Check the Routing Topology

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

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

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

- **pgr\_analyzeGraph**
- **pgr\_analyzeOneWay**
- **pgr\_nodeNetwork**

#### Compute a Path

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

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

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

- **pgr\_dijkstra**

#### Group of Functions

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

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

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

##### One to One

When routing from:

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

##### One to Many

When routing from:

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

##### Many to One

When routing from:

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

##### Many to Many

When routing from:

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

## Inner Queries

- **Description of the edges\_sql query for dijkstra like functions**

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

Where:

### **ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

### **ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Description of the edges\_sql query for dijkstra like functions

Column	Type	Default	Description
<b>id</b>	<code>ANY-INTEGERS</code>		Identifier of the edge.
<b>source</b>	<code>ANY-INTEGERS</code>		Identifier of the first end point vertex of the edge.
<b>target</b>	<code>ANY-INTEGERS</code>		Identifier of the second end point vertex of the edge.
<b>cost</b>	<code>ANY-NUMERICAL</code>		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>• When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	<code>ANY-NUMERICAL</code>	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>• When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

### **ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

### **ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Description of the edges\_sql query (id is not necessary)

### **edges\_sql:**

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

Column	Type	Default	Description
<b>source</b>	<code>ANY-INTEGERS</code>		Identifier of the first end point vertex of the edge.
<b>target</b>	<code>ANY-INTEGERS</code>		Identifier of the second end point vertex of the edge.
<b>cost</b>	<code>ANY-NUMERICAL</code>		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>• When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	<code>ANY-NUMERICAL</code>	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>• When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

### **ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

### **ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	<code>TEXT</code>		SQL query as described above.

Parameter	Type	Default	Description
<b>via_vertices</b>	ARRAY[ANY-INTEGER]		Array of ordered vertices identifiers that are going to be visited.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>
<b>strict</b>	BOOLEAN	false	<ul style="list-style-type: none"> <li>When <b>false</b> ignores missing paths returning all paths found</li> <li>When <b>true</b> if a path is missing stops and returns <i>EMPTY SET</i></li> </ul>
<b>U_turn_on_edge</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> 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 <i>id</i> is allowed.</li> <li>When <b>false</b> 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 <i>id</i> is used when no other path is found.</li> </ul>

### edges\_sql query for aStar - Family of functions and aStar - Family of functions functions

#### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### For pgr\_pushRelabel, pgr\_edmondsKarp, pgr\_boykovKolmogorov :

#### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

### For pgr\_maxFlowMinCost - Experimental and pgr\_maxFlowMinCost\_Cost - Experimental:

#### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Capacity of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Capacity of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) if it exists.
<b>reverse_cost</b>	ANY-NUMERICAL	0	Weight of the edge ( <i>target</i> , <i>source</i> ) if it exists.

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

smallint, int, bigint, real, float

**Description of the Points SQL query**

**points\_sql:**

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

Column	Type	Description
<b>pid</b>	ANY-INTEGERS	(optional) Identifier of the point. <ul style="list-style-type: none"> <li>If column present, it can not be NULL.</li> <li>If column not present, a sequential identifier will be given automatically.</li> </ul>
<b>edge_id</b>	ANY-INTEGERS	Identifier of the "closest" edge to the point.
<b>fraction</b>	ANY-NUMERICAL	Value in <0,1> that indicates the relative position from the first end point of the edge.
<b>side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: <ul style="list-style-type: none"> <li>In the right, left of the edge or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>

Where:

**ANY-INTEGERS:**

smallint, int, bigint

**ANY-NUMERICAL:**

smallint, int, bigint, real, float

**Return columns & values**

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

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

**Return values for a path**

Returns set of (seq, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

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



Column	Type	Description
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li>• <b>Many to One</b></li> <li>• <b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li>• <b>One to Many</b></li> <li>• <b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

#### Return values for multiple paths from the same source and destination

Returns set of (seq, path\_id, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_id</b>	INT	Path identifier. Has value <b>1</b> for the first of a path. Used when there are multiple paths for the same <code>start_vid</code> to <code>end_vid</code> combination.
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li>• <b>Many to One</b></li> <li>• <b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li>• <b>One to Many</b></li> <li>• <b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

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

Returns SET OF (start\_vid, end\_vid, agg\_cost)

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

#### Description of the Return Values

For **pgr\_pushRelabel**, **pgr\_edmondsKarp**, **pgr\_boykovKolmogorov** :

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

For **pgr\_maxFlowMinCost - Experimental**

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .

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

## Advanced Topics

- **Routing Topology**
- **Graph Analytics**
- **Analyze a Graph**
- **Analyze One Way Streets**
  - **Example**

## Routing Topology

### Overview

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

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

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

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

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

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

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

```

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

-- UPDATE the cost information based on oneway streets

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

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

VACUUM ANALYZE VERBOSE edge_table;

```

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

## Graph Analytics

### Overview

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

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

### Analyze a Graph

With `pgr_analyzeGraph` the graph can be checked for errors. For example for table “mytab” that has “mytab\_vertices\_pgr” as the vertices table:

```

SELECT pgr_analyzeGraph('mytab', 0.000002);
NOTICE: Performing checks, 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: 158
NOTICE: Dead ends: 20028
NOTICE: Potential gaps found near dead ends: 527
NOTICE: Intersections detected: 2560
NOTICE: Ring geometries: 0
pgr_analyzeGraph
-----
OK
(1 row)

```

In the vertices table “mytab\_vertices\_pgr”:

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

```

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

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

```

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

```

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

```

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

### Analyze One Way Streets

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

This query will add two columns to the vertices\_tmp table `in int` and `out int` and populate it with the appropriate counts. After running this on a graph you can identify nodes with potential problems with the following query.

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

#### Example

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

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

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

```

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

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

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

```

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

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

## Performance Tips

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

### For the Routing functions

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

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

```

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

```

Integrating the inner query to the pgRouting function:

```

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

```

### For the topology functions:

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

Analyze the new topology based on the actual topology:

```

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

```

Or create a new topology if the change is permanent:

```

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

```

## How to contribute

### Wiki

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

### Adding Functionaity to pgRouting

Consult the **developer's documentation**

## Indices and tables

- [Index](#)
- [Search Page](#)

## Reference

- [pgr\\_version](#) - Get pgRouting's version information.
- [pgr\\_full\\_version](#) - Get pgRouting's details of version.

## pgr\_version

`pgr_version` — Query for pgRouting version information.

## Availability

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

## Support

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

## Description

Returns pgRouting version information.

## Signature

```
TEXT pgr_version();
```

## Example:

pgRouting Version for this documentatoin

```
SELECT pgr_version();
pgr_version
-----
3.0.5
(1 row)
```

## Result Columns

Type	Description
<code>TEXT</code>	pgRouting version

## See Also

- [pgr\\_full\\_version](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

## pgr\_full\_version

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

## Availability

- Version 3.0.0
  - New **official** function

## Support

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

## Description

Get the details of pgRouting version information

## Signatures

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

### Example:

Information when this documentation was build

```
SELECT * FROM pgr_full_version();  
version | build_type | compile_date | library | system | postgresql | compiler | boost | hash  
-----+-----+-----+-----+-----+-----+-----+-----+-----  
3.0.5 | Debug | 2021/01/22 | pgrouting-3.0.5 | Linux-5.4.0-62-generic | PostgreSQL 12.5 (Ubuntu 12.5-0ubuntu0.20.04.1) | GNU-8.4.0 | 1.71.0 | e7dd70b5b2  
(1 row)
```

## Result Columns

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

## See Also

- [pgr\\_version](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

## Function Families

### Function Families

#### All Pairs - Family of Functions

- [pgr\\_floydWarshall](#) - Floyd-Warshall's algorithm.
- [pgr\\_johnson](#) - Johnson's algorithm

#### aStar - Family of functions

- [pgr\\_aStar](#) - A\* algorithm for the shortest path.
- [pgr\\_aStarCost](#) - Get the aggregate cost of the shortest paths.
- [pgr\\_aStarCostMatrix](#) - Get the cost matrix of the shortest paths.

#### Bidirectional A\* - Family of functions

- [pgr\\_bdAStar](#) - Bidirectional A\* algorithm for obtaining paths.
- [pgr\\_bdAStarCost](#) - Bidirectional A\* algorithm to calculate the cost of the paths.
- [pgr\\_bdAStarCostMatrix](#) - Bidirectional A\* algorithm to calculate a cost matrix of paths.

#### Bidirectional Dijkstra - Family of functions

- **pgr\_bdDijkstra** - Bidirectional Dijkstra algorithm for the shortest paths.
- **pgr\_bdDijkstraCost** - Bidirectional Dijkstra to calculate the cost of the shortest paths
- **pgr\_bdDijkstraCostMatrix** - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

### Components - Family of functions

- **pgr\_connectedComponents** - Connected components of an undirected graph.
- **pgr\_strongComponents** - Strongly connected components of a directed graph.
- **pgr\_biconnectedComponents** - Biconnected components of an undirected graph.
- **pgr\_articulationPoints** - Articulation points of an undirected graph.
- **pgr\_bridges** - Bridges of an undirected graph.

### Contraction - Family of functions

- **pgr\_contraction**

### Dijkstra - Family of functions

- **pgr\_dijkstra** - Dijkstra's algorithm for the shortest paths.
- **pgr\_dijkstraCost** - Get the aggregate cost of the shortest paths.
- **pgr\_dijkstraCostMatrix** - Use pgr\_dijkstra to create a costs matrix.
- **pgr\_drivingDistance** - Use pgr\_dijkstra to calculate catchment information.
- **pgr\_KSP** - Use Yen algorithm with pgr\_dijkstra to get the K shortest paths.

### Flow - Family of functions

- **pgr\_maxFlow** - Only the Max flow calculation using Push and Relabel algorithm.
- **pgr\_boykovKolmogorov** - Boykov and Kolmogorov with details of flow on edges.
- **pgr\_edmondsKarp** - Edmonds and Karp algorithm with details of flow on edges.
- **pgr\_pushRelabel** - Push and relabel algorithm with details of flow on edges.
- Applications
  - **pgr\_edgeDisjointPaths** - Calculates edge disjoint paths between two groups of vertices.
  - **pgr\_maxCardinalityMatch** - Calculates a maximum cardinality matching in a graph.

### Kruskal - Family of functions

- **pgr\_kruskal**
- **pgr\_kruskalBFS**
- **pgr\_kruskalDD**
- **pgr\_kruskalDFS**

### Prim - Family of functions

- **pgr\_prim**
- **pgr\_primBFS**
- **pgr\_primDD**
- **pgr\_primDFS**

### Topology - Family of Functions

- **pgr\_createTopology** - to create a topology based on the geometry.
- **pgr\_createVerticesTable** - to reconstruct the vertices table based on the source and target information.
- **pgr\_analyzeGraph** - to analyze the edges and vertices of the edge table.
- **pgr\_analyzeOneWay** - to analyze directionality of the edges.
- **pgr\_nodeNetwork** -to create nodes to a not noded edge table.

### Traveling Sales Person - Family of functions

- **pgr\_TSP** - When input is given as matrix cell information.
- **pgr\_TSPeuclidean** - When input are coordinates.

**pgr\_trsp - Turn Restriction Shortest Path (TRSP)** - Turn Restriction Shortest Path (TRSP)

Functions by categories

### Cost - Category

- **pgr\_aStarCost**
- **pgr\_dijkstraCost**

### Cost Matrix - Category

- **pgr\_aStarCostMatrix**
- **pgr\_dijkstraCostMatrix**

### Driving Distance - Category



- **pgr\_drivingDistance** - Driving Distance based on Dijkstra's algorithm
- **pgr\_primDD** - Driving Distance based on Prim's algorithm
- **pgr\_kruskalDD** - Driving Distance based on Kruskal's algorithm
- Post processing
  - **pgr\_alphaShape** - Alpha shape computation

### K shortest paths - Category

- **pgr\_KSP** - Yen's algorithm based on pgr\_dijkstra

### Spanning Tree - Category

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

### All Pairs - Family of Functions

The following functions work on all vertices pair combinations

- **pgr\_floydWarshall** - Floyd-Warshall's algorithm.
- **pgr\_johnson** - Johnson's algorithm

#### pgr\_floydWarshall

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



Boost Graph Inside

### Availability

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

### Support

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

### Description

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

The main characteristics are:

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

### Signatures

### Summary

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

## Using defaults

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

### Example 1:

For vertices  $\{1, 2, 3, 4\}$  on a **directed** graph

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

## Complete Signature

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

### Example 2:

For vertices  $\{1, 2, 3, 4\}$  on an **undirected** graph

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

## Parameters

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

## Inner query

### Description of the edges\_sql query (id is not necessary)

#### edges\_sql:

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

Column	Type	Default	Description
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (start\_vid, end\_vid, agg\_cost)

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

See Also

- **pgr\_johnson**
- **Boost floyd-Warshall** algorithm
- Queries uses the **Sample Data** network.

Indices and tables

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

**pgr\_johnson**

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



Boost Graph Inside

Availability

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

Support

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

Description

The Johnson algorithm, is a good choice to calculate the sum of the costs of the shortest path for each pair of nodes in the graph, for *sparse graphs*. It uses 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  $agg\_cost = 0$ .

Signatures

Summary

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

### Using default

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

#### Example 1:

For vertices  $\{1, 2, 3, 4\}$  on a **directed** graph

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

### Complete Signature

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

#### Example 2:

For vertices  $\{1, 2, 3, 4\}$  on an **undirected** graph

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

### Parameters

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

### Inner query

#### Description of the edges\_sql query (id is not necessary)

##### edges\_sql:

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

Column	Type	Default	Description
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (start\_vid, end\_vid, agg\_cost)

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

See Also

- [pgr\\_floydWarshall](#)
- [Boost Johnson](#) algorithm implementation.
- Queries uses the [Sample Data](#) network.

**Indices and tables**

- [Index](#)
- [Search Page](#)

**Previous versions of this page**

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

**Performance**

The following tests:

- non server computer
- with AMD 64 CPU
- 4G memory
- trusty
- postgresSQL 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?*[@meta]"
```

Data processing was done with osm2pgrouting-alpha

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

Results

**Test:**

One

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

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

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

The results of this tests are presented as:

**SIZE:**

is the number of edges given as input.

**EDGES:**

is the total number of records in the query.

**DENSITY:**

is the density of the data  $\frac{E}{V \times (V-1)}$ .

**OUT ROWS:**

is the number of records returned by the queries.

**Floyd-Warshall:**

is the average execution time in seconds of pgr\_floydWarshall.

**Johnson:**

is the average execution time in seconds of pgr\_johnson.

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

**Test:**

Two

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

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

The tested queries

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

The results of this tests are presented as:

**SIZE:**

is the size of the bounding box.

**EDGES:**

is the total number of records in the query.

**DENSITY:**

is the density of the data  $\frac{E}{V \times (V-1)}$ .

**OUT ROWS:**

is the number of records returned by the queries.

**Floyd-Warshall:**

is the average execution time in seconds of pgr\_floydWarshall.

**Johnson:**

is the average execution time in seconds of pgr\_johnson.

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
0.001	44	0.0608	1197	0.10	0.10
0.002	99	0.0251	4330	0.10	0.10

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

See Also

- [pgr\\_johnson](#)
- [pgr\\_floydWarshall](#)
- [Boost floyd-Warshall](#) algorithm

## Indices and tables

- [Index](#)
- [Search Page](#)

## aStar - Family of functions

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

- [pgr\\_aStar](#) - A\* algorithm for the shortest path.
- [pgr\\_aStarCost](#) - Get the aggregate cost of the shortest paths.
- [pgr\\_aStarCostMatrix](#) - Get the cost matrix of the shortest paths.

`pgr_aStar`

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



Boost Graph Inside

## Availability

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

## Support

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

## Description

### The main characteristics are:

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

## Signatures

### Summary

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

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

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

### Using defaults

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

### Example:

From vertex  $2$  to vertex  $12$  on a **directed** graph

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

### One to One

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

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

### Example:

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



```
SELECT * FROM pgr_astar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
  2, 12,
  directed := false, heuristic := 2);
seq | path_seq | node | edge | cost | agg_cost
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	2	1	0
2	2	3	3	1	1
3	3	4	16	1	2
4	4	9	15	1	3
5	5	12	-1	0	4

(5 rows)

### One to many

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

#### Example:

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

```
SELECT * FROM pgr_astar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
  2, ARRAY[3, 12], heuristic := 2);
seq | path_seq | end_vid | node | edge | cost | agg_cost
```

seq	path_seq	end_vid	node	edge	cost	agg_cost
1	1	3	2	4	1	0
2	2	3	5	8	1	1
3	3	3	6	9	1	2
4	4	3	9	16	1	3
5	5	3	4	3	1	4
6	6	3	3	-1	0	5
7	1	12	2	4	1	0
8	2	12	5	10	1	1
9	3	12	10	12	1	2
10	4	12	11	13	1	3
11	5	12	12	-1	0	4

(11 rows)

### Many to One

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

#### Example:

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

```
SELECT * FROM pgr_astar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
  ARRAY[7, 2], 12, heuristic := 0);
seq | path_seq | start_vid | node | edge | cost | agg_cost
```

seq	path_seq	start_vid	node	edge	cost	agg_cost
1	1	2	2	4	1	0
2	2	2	5	10	1	1
3	3	2	10	12	1	2
4	4	2	11	13	1	3
5	5	2	12	-1	0	4
6	1	7	7	6	1	0
7	2	7	8	7	1	1
8	3	7	5	10	1	2
9	4	7	10	12	1	3
10	5	7	11	13	1	4
11	6	7	12	-1	0	5

(11 rows)

### Many to Many

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

#### Example:

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

```
SELECT * FROM pgr_astar(
  'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
  ARRAY[7, 2], ARRAY[3, 12], heuristic := 2);
seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 2 | 3 | 2 | 4 | 1 | 0
 2 | 2 | 2 | 3 | 5 | 8 | 1 | 1
 3 | 3 | 2 | 3 | 6 | 9 | 1 | 2
 4 | 4 | 2 | 3 | 9 | 16 | 1 | 3
 5 | 5 | 2 | 3 | 4 | 3 | 1 | 4
 6 | 6 | 2 | 3 | 3 | -1 | 0 | 5
 7 | 1 | 2 | 12 | 2 | 4 | 1 | 0
 8 | 2 | 2 | 12 | 5 | 10 | 1 | 1
 9 | 3 | 2 | 12 | 10 | 12 | 1 | 2
10 | 4 | 2 | 12 | 11 | 13 | 1 | 3
11 | 5 | 2 | 12 | 12 | -1 | 0 | 4
12 | 1 | 7 | 3 | 7 | 6 | 1 | 0
13 | 2 | 7 | 3 | 8 | 7 | 1 | 1
14 | 3 | 7 | 3 | 5 | 8 | 1 | 2
15 | 4 | 7 | 3 | 6 | 9 | 1 | 3
16 | 5 | 7 | 3 | 9 | 16 | 1 | 4
17 | 6 | 7 | 3 | 4 | 3 | 1 | 5
18 | 7 | 7 | 3 | 3 | -1 | 0 | 6
19 | 1 | 7 | 12 | 7 | 6 | 1 | 0
20 | 2 | 7 | 12 | 8 | 7 | 1 | 1
21 | 3 | 7 | 12 | 5 | 10 | 1 | 2
22 | 4 | 7 | 12 | 10 | 12 | 1 | 3
23 | 5 | 7 | 12 | 11 | 13 | 1 | 4
24 | 6 | 7 | 12 | 12 | -1 | 0 | 5
(24 rows)
```

#### Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	<b>edges_sql</b> inner query.
<b>from_vid</b>	ANY-INTEGERS	Starting vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>One to Many</li> </ul>
<b>from_vids</b>	ARRAY[ANY-INTEGERS]	Array of starting vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>Many to One</li> <li>Many to Many</li> </ul>
<b>to_vid</b>	ANY-INTEGERS	Ending vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>Many to One</li> </ul>
<b>to_vids</b>	ARRAY[ANY-INTEGERS]	Array of ending vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>One to Many</li> <li>Many to Many</li> </ul>

#### Optional Parameters

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <code>true</code> the graph is considered as Directed.</li> <li>When <code>false</code> the graph is considered as Undirected.</li> </ul>
<b>heuristic</b>	INTEGER	5	Heuristic number. Current valid values 0~5. Defaults <ul style="list-style-type: none"> <li>0: <math>h(v) = 0</math> (Use this value to compare with <code>pgr_dijkstra</code>)</li> <li>1: <math>h(v) = \text{abs}(\max(dx, dy))</math></li> <li>2: <math>h(v) = \text{abs}(\min(dx, dy))</math></li> <li>3: <math>h(v) = dx * dx + dy * dy</math></li> <li>4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	1	For units manipulation. $(\text{factor} > 0)$ . See <b>Factor</b>
<b>epsilon</b>	FLOAT	1	For less restricted results. $(\text{epsilon} \geq 1)$ .

#### Inner query

edges\_sql

#### edges\_sql:

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

Column	Type	Default	Description
--------	------	---------	-------------

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li><b>One to Many</b></li> <li><b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

See Also

- [aStar - Family of functions](#)
- [Sample Data](#)
- [https://www.boost.org/libs/graph/doc/astar\\_search.html](https://www.boost.org/libs/graph/doc/astar_search.html)
- [https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

**Indices and tables**

- [Index](#)
- [Search Page](#)

`pgr_aStarCost`

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



## Availability

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

## Support

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

## Description

### The main characteristics are:

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

## Signatures

## Summary

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

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

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

## Using defaults

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

## Example:

From vertex  $2$  to vertex  $12$  on a **directed** graph

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

## One to One

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

**Example:**

From vertex  $\{2\}$  to vertex  $\{12\}$  on an **undirected** graph using heuristic  $\{2\}$

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

One to many

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

**Example:**

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

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

Many to One

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

**Example:**

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

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

Many to Many

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

**Example:**

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

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

Parameters

Parameter	Type	Description
-----------	------	-------------

Parameter	Type	Description
<b>edges_sql</b>	TEXT	<b>edges_sql</b> inner query.
<b>from_vid</b>	ANY-INTEGER	Starting vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>One to Many</li> </ul>
<b>from_vids</b>	ARRAY[ANY-INTEGER]	Array of starting vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>Many to One</li> <li>Many to Many</li> </ul>
<b>to_vid</b>	ANY-INTEGER	Ending vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>Many to One</li> </ul>
<b>to_vids</b>	ARRAY[ANY-INTEGER]	Array of ending vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>One to Many</li> <li>Many to Many</li> </ul>

#### Optional Parameters

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> the graph is considered as Directed.</li> <li>When <b>false</b> the graph is considered as Undirected.</li> </ul>
<b>heuristic</b>	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 <ul style="list-style-type: none"> <li>0: <math>h(v) = 0</math> (Use this value to compare with <code>pgr_dijkstra</code>)</li> <li>1: <math>h(v) = \text{abs}(\max(dx, dy))</math></li> <li>2: <math>h(v) = \text{abs}(\min(dx, dy))</math></li> <li>3: <math>h(v) = dx * dx + dy * dy</math></li> <li>4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	1	For units manipulation. $(\text{factor} > 0)$ . See <b>Factor</b>
<b>epsilon</b>	FLOAT	1	For less restricted results. $(\text{epsilon} \geq 1)$ .

#### Inner query

edges\_sql

#### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

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

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

See Also

- [aStar - Family of functions](#)
- [Cost - Category](#)
- [Cost Matrix - Category](#)
- Examples use [Sample Data](#) network.

## Indices and tables

- [Index](#)
- [Search Page](#)

`pgr_aStarCostMatrix`

`pgr_aStarCostMatrix` - Calculates the a cost matrix using [pgr\\_aStar](#).



Boost Graph Inside

## Availability

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

## Support

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

Description

### The main characteristics are:

- Using internally 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  $\infty$
  - when  $(v = u)$  then
    - no corresponding row is returned
    - cost from  $v$  to  $u$  is  $0$
- When the graph is **undirected** the cost matrix is symmetric

Signatures

## Summary

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

## Using defaults

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

**Example:**

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

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

## Complete Signature

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

**Example:**

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

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

## Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	<b>edges_sql</b> inner query.
<b>vids</b>	ARRAY[ANY-INTEGERS]	Array of vertices identifiers.

## Optional Parameters

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> the graph is considered as Directed.</li> <li>When <b>false</b> the graph is considered as Undirected.</li> </ul>
<b>heuristic</b>	INTEGER	5	Heuristic number. Current valid values 0~5. Defaults <ul style="list-style-type: none"> <li>0: <math>h(v) = 0</math> (Use this value to compare with <code>pgr_dijkstra</code>)</li> <li>1: <math>h(v) = \text{abs}(\max(dx, dy))</math></li> <li>2: <math>h(v) = \text{abs}(\min(dx, dy))</math></li> <li>3: <math>h(v) = dx * dx + dy * dy</math></li> <li>4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	1	For units manipulation. $(\text{factor} > 0)$ . See <b>Factor</b>
<b>epsilon</b>	FLOAT	1	For less restricted results. $(\text{epsilon} \geq 1)$ .

Inner query  
edges\_sql



### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGGER		Identifier of the edge.
<b>source</b>	ANY-INTEGGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

#### ANY-INTEGGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start\_vid, end\_vid, agg\_cost)

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

Additional Examples

#### Example:

Use with **pgr\_TSP**

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

See Also

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

Indices and tables

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

## Previous versions of this page

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

### General Information

The main Characteristics are:

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

### Advanced documentation

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

Heuristic

Currently the heuristic functions available are:

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

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

- [pgr\\_aStar](#)
- [pgr\\_aStarCost](#)
- [pgr\\_aStarCostMatrix](#)
- [https://www.boost.org/libs/graph/doc/astar\\_search.html](https://www.boost.org/libs/graph/doc/astar_search.html)
- [https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

## Indices and tables

- [Index](#)
- [Search Page](#)

## Bidirectional A\* - Family of functions

- [pgr\\_bdAstar](#) - Bidirectional A\* algorithm for obtaining paths.
- [pgr\\_bdAstarCost](#) - Bidirectional A\* algorithm to calculate the cost of the paths.
- [pgr\\_bdAstarCostMatrix](#) - Bidirectional A\* algorithm to calculate a cost matrix of paths.

`pgr_bdAstar`

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



Boost Graph Inside

## Availability:

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

## Support

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

Description

## The main characteristics are:

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

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

Signature

## Summary

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

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

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

## Using defaults

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

### Example:

From vertex  $\{(2)\}$  to vertex  $\{(3)\}$  on a **directed** graph

```
SELECT * FROM pgr_bdAstar(
  'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
   FROM edge_table',
  2, 3
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	4	1	0
2	2	5	8	1	1
3	3	6	9	1	2
4	4	9	16	1	3
5	5	4	3	1	4
6	6	3	-1	0	5

(6 rows)

## One to One

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

### Example:

From vertex  $\{(2)\}$  to vertex  $\{(3)\}$  on a **directed** graph using heuristic  $\{(2)\}$

```
SELECT * FROM pgr_bdAstar(
  'SELECT id, source, target, cost, reverse_cost, x1,y1,x2,y2
   FROM edge_table',
  2, 3,
  true, heuristic := 2
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	4	1	0
2	2	5	8	1	1
3	3	6	9	1	2
4	4	9	16	1	3
5	5	4	3	1	4
6	6	3	-1	0	5

(6 rows)

## One to many

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

### Example:

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

```

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

```

### Many to One

```

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

```

#### Example:

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

```

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

```

### Many to Many

```

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

```

#### Example:

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

```

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

```

## Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	<b>edges_sql</b> inner query.
<b>from_vid</b>	ANY-INTEGGER	Starting vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>One to Many</li> </ul>
<b>from_vids</b>	ARRAY[ANY-INTEGGER]	Array of starting vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>Many to One</li> <li>Many to Many</li> </ul>
<b>to_vid</b>	ANY-INTEGGER	Ending vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>Many to One</li> </ul>
<b>to_vids</b>	ARRAY[ANY-INTEGGER]	Array of ending vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>One to Many</li> <li>Many to Many</li> </ul>

## Optional Parameters

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> the graph is considered as Directed.</li> <li>When <b>false</b> the graph is considered as Undirected.</li> </ul>
<b>heuristic</b>	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 <ul style="list-style-type: none"> <li>0: <math>h(v) = 0</math> (Use this value to compare with <code>pgr_dijkstra</code>)</li> <li>1: <math>h(v) = \text{abs}(\max(dx, dy))</math></li> <li>2: <math>h(v) = \text{abs}(\min(dx, dy))</math></li> <li>3: <math>h(v) = dx * dx + dy * dy</math></li> <li>4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	1	For units manipulation. $\backslash(\text{factor} > 0\backslash)$ . See <b>Factor</b>
<b>epsilon</b>	FLOAT	1	For less restricted results. $\backslash(\text{epsilon} >= 1\backslash)$ .

## Inner query

edges\_sql

### edges\_sql:

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

Column	Type	Default	Description
--------	------	---------	-------------

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, path\_id, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_id</b>	INT	Path identifier. Has value <b>1</b> for the first of a path. Used when there are multiple paths for the same <code>start_vid</code> to <code>end_vid</code> combination.
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li><b>One to Many</b></li> <li><b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

See Also

- [aStar - Family of functions](#)
- [Bidirectional A\\* - Family of functions](#)
- [Sample Data](#) network.
- [https://www.boost.org/libs/graph/doc/astar\\_search.html](https://www.boost.org/libs/graph/doc/astar_search.html)
- [https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

Indices and tables

- [Index](#)
- [Search Page](#)

`pgr_bdAstarCost`

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



### Availability

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

### Description

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

### Signatures

### Summary

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

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

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

### Using defaults

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

### Example:

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

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

### One to One



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

#### Example:

From vertex  $\{2\}$  to vertex  $\{3\}$  on an **directed** graph using heuristic  $\{2\}$

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

#### One to many

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

#### Example:

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

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

#### Many to One

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

#### Example:

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

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

#### Many to Many

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

#### Example:

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

```

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

```

## Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	<b>edges_sql</b> inner query.
<b>from_vid</b>	ANY-INTEGER	Starting vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>One to Many</li> </ul>
<b>from_vids</b>	ARRAY[ANY-INTEGER]	Array of starting vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>Many to One</li> <li>Many to Many</li> </ul>
<b>to_vid</b>	ANY-INTEGER	Ending vertex identifier. Parameter in: <ul style="list-style-type: none"> <li>One to One</li> <li>Many to One</li> </ul>
<b>to_vids</b>	ARRAY[ANY-INTEGER]	Array of ending vertices identifiers. Parameter in: <ul style="list-style-type: none"> <li>One to Many</li> <li>Many to Many</li> </ul>

## Optional Parameters

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> the graph is considered as Directed.</li> <li>When <b>false</b> the graph is considered as Undirected.</li> </ul>
<b>heuristic</b>	INTEGER	5	Heuristic number. Current valid values 0~5. Default 5 <ul style="list-style-type: none"> <li>0: <math>h(v) = 0</math> (Use this value to compare with pgr_dijkstra)</li> <li>1: <math>h(v) = \text{abs}(\max(dx, dy))</math></li> <li>2: <math>h(v) = \text{abs}(\min(dx, dy))</math></li> <li>3: <math>h(v) = dx * dx + dy * dy</math></li> <li>4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	1	For units manipulation. $(\text{factor} > 0)$ . See <b>Factor</b>
<b>epsilon</b>	FLOAT	1	For less restricted results. $(\text{epsilon} \geq 1)$ .

## Inner query

edges\_sql

### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.

Column	Type	Default	Description
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (start\_vid, end\_vid, agg\_cost)

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

See Also

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

**Indices and tables**

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

`pgr_bdAstarCostMatrix`

`pgr_bdAstarCostMatrix` - Calculates the a cost matrix using **pgr\_aStar**.



Boost Graph Inside

**Availability**

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

**Support**

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

Description

**The main characteristics are:**

- Using internally 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  $\infty$
  - when  $(v = u)$  then
    - no corresponding row is returned
    - cost from  $v$  to  $u$  is  $0$
- When the graph is **undirected** the cost matrix is symmetric

## Signatures

### Summary

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

### Using defaults

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

### Example:

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

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

### Complete Signature

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

### Example:

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

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

### Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	<b>edges_sql</b> inner query.
<b>vids</b>	ARRAY[ANY-INTEGER]	Array of vertices identifiers.

### Optional Parameters

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"><li>When <b>true</b> the graph is considered as Directed.</li><li>When <b>false</b> the graph is considered as Undirected.</li></ul>

Parameter	Type	Default	Description
<b>heuristic</b>	INTEGER	5	Heuristic number. Current valid values 0~5. Defaults <ul style="list-style-type: none"> <li>0: <math>h(v) = 0</math> (Use this value to compare with <code>pgr_dijkstra</code>)</li> <li>1: <math>h(v) = \text{abs}(\max(dx, dy))</math></li> <li>2: <math>h(v) = \text{abs}(\min(dx, dy))</math></li> <li>3: <math>h(v) = dx * dx + dy * dy</math></li> <li>4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	1	For units manipulation. $\backslash(\text{factor} > 0\backslash)$ . See <b>Factor</b>
<b>epsilon</b>	FLOAT	1	For less restricted results. $\backslash(\text{epsilon} \geq 1\backslash)$ .

Inner query  
edges\_sql

**edges\_sql:**

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

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

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

Additional Examples

**Example:**

Use with **pgr\_TSP**

```

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

```

See Also

- [aStar - Family of functions](#)
- [Bidirectional A\\* - Family of functions](#)
- [Cost - Category](#)
- [Cost Matrix - Category](#)
- [Traveling Sales Person - Family of functions](#)
- The queries use the [Sample Data](#) network.

## Indices and tables

- [Index](#)
- [Search Page](#)

## Previous versions of this page

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

## Description

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

The main Characteristics are:

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

## Signatures

### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGGER		Identifier of the edge.
<b>source</b>	ANY-INTEGGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>• When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>• When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>x1</b>	ANY-NUMERICAL		X coordinate of <i>source</i> vertex.

Column	Type	Default	Description
<b>y1</b>	ANY-NUMERICAL		Y coordinate of <i>source</i> vertex.
<b>x2</b>	ANY-NUMERICAL		X coordinate of <i>target</i> vertex.
<b>y2</b>	ANY-NUMERICAL		Y coordinate of <i>target</i> vertex.

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	Edges SQL query as described above.
<b>start_vid</b>	ANY-INTEGERS	Starting vertex identifier.
<b>start_vids</b>	ARRAY[ANY-INTEGERS]	Starting vertices identifiers.
<b>end_vid</b>	ANY-INTEGERS	Ending vertex identifier.
<b>end_vids</b>	ARRAY[ANY-INTEGERS]	Ending vertices identifiers.
<b>directed</b>	BOOLEAN	<ul style="list-style-type: none"> <li>• Optional.</li> <li>• When <code>false</code> the graph is considered as Undirected.</li> <li>• Default is <code>true</code> which considers the graph as Directed.</li> </ul>
<b>heuristic</b>	INTEGER	(optional). Heuristic number. Current valid values 0~5. Default <code>5</code> <ul style="list-style-type: none"> <li>• 0: <math>h(v) = 0</math> (Use this value to compare with <code>pgr_dijkstra</code>)</li> <li>• 1: <math>h(v) = \text{abs}(\text{max}(dx, dy))</math></li> <li>• 2: <math>h(v) = \text{abs}(\text{min}(dx, dy))</math></li> <li>• 3: <math>h(v) = dx * dx + dy * dy</math></li> <li>• 4: <math>h(v) = \text{sqrt}(dx * dx + dy * dy)</math></li> <li>• 5: <math>h(v) = \text{abs}(dx) + \text{abs}(dy)</math></li> </ul>
<b>factor</b>	FLOAT	(optional). For units manipulation. $(\text{factor} > 0)$ . Default <code>1</code> . see <b>Factor</b>
<b>epsilon</b>	FLOAT	(optional). For less restricted results. $(\text{epsilon} \geq 1)$ . Default <code>1</code> .

See Also

Indices and tables

- [Index](#)
- [Search Page](#)

**Bidirectional Dijkstra - Family of functions**

- **[pgr\\_bdDijkstra](#)** - Bidirectional Dijkstra algorithm for the shortest paths.
- **[pgr\\_bdDijkstraCost](#)** - Bidirectional Dijkstra to calculate the cost of the shortest paths
- **[pgr\\_bdDijkstraCostMatrix](#)** - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

**pgr\_bdDijkstra**

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



Boost Graph Inside

**Availability:**

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

- o pgr\_bdDijkstra(Many to Many)
- o Version 2.4.0
  - o Signature change on pgr\_bdDijkstra(One to One)
    - o Old signature no longer supported
- o Version 2.0.0
  - o **Official** pgr\_bdDijkstra(One to One)

## Support

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

## Description

### The main characteristics are:

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

## Signatures

### Summary

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

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

### Using defaults

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

### Example:

From vertex  $\backslash(2\backslash)$  to vertex  $\backslash(3\backslash)$

```
SELECT * FROM pgr_bdDijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3
);
```

```
seq | path_seq | node | edge | cost | agg_cost
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	4	1	0
2	2	5	8	1	1
3	3	6	9	1	2
4	4	9	16	1	3
5	5	4	3	1	4
6	6	3	-1	0	5

(6 rows)

### One to One

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

### Example:

From vertex  $\backslash(2\backslash)$  to vertex  $\backslash(3\backslash)$  on an **undirected** graph



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

### One to many

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

#### Example:

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

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

### Many to One

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

#### Example:

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

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

### Many to Many

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

#### Example:

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

```
SELECT * FROM pgr_bdijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[2, 7], ARRAY[3, 11]);
seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 2 | 3 | 2 | 4 | 1 | 0
 2 | 2 | 2 | 3 | 5 | 8 | 1 | 1
 3 | 3 | 2 | 3 | 6 | 9 | 1 | 2
 4 | 4 | 2 | 3 | 9 | 16 | 1 | 3
 5 | 5 | 2 | 3 | 4 | 3 | 1 | 4
 6 | 6 | 2 | 3 | 3 | -1 | 0 | 5
 7 | 1 | 2 | 11 | 2 | 4 | 1 | 0
 8 | 2 | 2 | 11 | 5 | 8 | 1 | 1
 9 | 3 | 2 | 11 | 6 | 11 | 1 | 2
10 | 4 | 2 | 11 | 11 | -1 | 0 | 3
11 | 1 | 7 | 3 | 7 | 6 | 1 | 0
12 | 2 | 7 | 3 | 8 | 7 | 1 | 1
13 | 3 | 7 | 3 | 5 | 8 | 1 | 2
14 | 4 | 7 | 3 | 6 | 9 | 1 | 3
15 | 5 | 7 | 3 | 9 | 16 | 1 | 4
16 | 6 | 7 | 3 | 4 | 3 | 1 | 5
17 | 7 | 7 | 3 | 3 | -1 | 0 | 6
18 | 1 | 7 | 11 | 7 | 6 | 1 | 0
19 | 2 | 7 | 11 | 8 | 7 | 1 | 1
20 | 3 | 7 | 11 | 5 | 10 | 1 | 2
21 | 4 | 7 | 11 | 10 | 12 | 1 | 3
22 | 5 | 7 | 11 | 11 | -1 | 0 | 4
(22 rows)
```

#### Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**  
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
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_id</b>	INT	Path identifier. Has value <b>1</b> for the first of a path. Used when there are multiple paths for the same <b>start_vid</b> to <b>end_vid</b> combination.
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>

Column	Type	Description
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li>One to Many</li> <li>Many to Many</li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. -1 for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

See Also

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

## Indices and tables

- Index**
- Search Page**

`pgr_bdDijkstraCost`

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



Boost Graph Inside

## Availability:

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

## Support

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

Description

## The main characteristics are:

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

Signatures

## Summary

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

RETURNS SET OF (start\_vid, end\_vid, agg\_cost)  
OR EMPTY SET

### Using default

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

#### Example:

From vertex  $\backslash(2\backslash)$  to vertex  $\backslash(3\backslash)$  on a **directed** graph

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

### One to One

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

#### Example:

From vertex  $\backslash(2\backslash)$  to vertex  $\backslash(3\backslash)$  on an **undirected** graph

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

### One to Many

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

#### Example:

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

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

### Many to One

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

#### Example:

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

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

## Many to Many

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

### Example:

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

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

## Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Result Columns

Returns SET OF (start\_vid, end\_vid, agg\_cost)

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

See Also

- The queries use the **Sample Data** network.
- **pgr\_bdDijkstra**
- <https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf>
- [https://en.wikipedia.org/wiki/Bidirectional\\_search](https://en.wikipedia.org/wiki/Bidirectional_search)

## Indices and tables

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

`pgr_bdDijkstraCostMatrix`

`pgr_bdDijkstraCostMatrix` - Calculates the a cost matrix using **pgr\_bdDijkstra**.



Boost Graph Inside

## Availability:

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

Description

## The main characteristics are:

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

Signatures

## Summary

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

## Using default

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

## Example:

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

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

### Complete Signature

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

### Example:

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

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

### Parameters

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

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Result Columns

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

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

## Additional Examples

### Example:

Use with `tsp`

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

## See Also

- [pgr\\_bdDijkstra](#)
- [Cost Matrix - Category](#)
- [pgr\\_TSP](#)
- The queries use the [Sample Data](#) network.

## Indices and tables

- [Index](#)
- [Search Page](#)

## Previous versions of this page

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

## Synopsis

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

## Characteristics

The main Characteristics are:

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

## See Also



## Indices and tables

- [Index](#)
- [Search Page](#)

## Components - Family of functions

- **pgr\_connectedComponents** - Connected components of an undirected graph.
- **pgr\_strongComponents** - Strongly connected components of a directed graph.
- **pgr\_biconnectedComponents** - Biconnected components of an undirected graph.
- **pgr\_articulationPoints** - Articulation points of an undirected graph.
- **pgr\_bridges** - Bridges of an undirected graph.

### pgr\_connectedComponents

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



Boost Graph Inside

## Availability

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

## Support

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

## Description

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

## The main characteristics are:

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

## Signatures

```
pgr_connectedComponents(edges_sql)
```

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

## Example:

The connected components of the graph

```

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

```

#### Parameters

Parameter	Type	Default	Description
<b>Edges SQL</b>	TEXT		Inner query as described below.

#### Inner query

##### edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

##### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

##### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

Returns set of (seq, component, node)

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

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**



## Boost Graph Inside

### Availability

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

### Support

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

### Description

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

### The main characteristics are:

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

### Signatures

```
pgr_strongComponents(Edges SQL)

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

### Example:

The strong components of the graph

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

### Parameters

Parameter	Type	Default	Description
<b>Edges SQL</b>	<code>TEXT</code>		Inner query as described below.

### Inner query

### edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, component, node)

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

See Also

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

### Indices and tables

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

#### pgr\_biconnectedComponents

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



Boost Graph Inside

### Availability

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

### Support

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

## Description

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

### The main characteristics are:

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

## Signatures

```
pgr_biconnectedComponents(Edges SQL)

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

### Example:

The biconnected components of the graph

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

## Parameters

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

## Inner query

### edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>• When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>• When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

## ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, component, edge)

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

See Also

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

## Indices and tables

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

pgr\_articulationPoints

pgr\_articulationPoints - Return the articulation points of an undirected graph.



Boost Graph Inside

## Availability

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

## Support

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

Description

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

## The main characteristics are:

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

Signatures

```
pgr_articulationPoints(Edges SQL)
```

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

## Example:

The articulation points of the graph

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

#### Parameters

Parameter	Type	Default	Description
<b>Edges SQL</b>	TEXT		Inner query as described below.

#### Inner query

##### edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

##### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

##### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

Returns set of (node)

Column	Type	Description
<b>node</b>	BIGINT	Identifier of the vertex.

See Also

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

#### Indices and tables

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

#### pgr\_bridges

pgr\_bridges - Return the bridges of an undirected graph.



Boost Graph Inside

#### Availability

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

## Support

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

## Description

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

## The main characteristics are:

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

## Signatures

```
pgr_bridges(Edges SQL)

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

## Example:

The bridges of the graph

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

## Parameters

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

## Inner query

### edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT



## Result Columns

Returns set of (edge)

Column	Type	Description
<b>edge</b>	BIGINT	Identifier of the edge that is a bridge.

See Also

- [https://en.wikipedia.org/wiki/Bridge\\_%28graph\\_theory%29](https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29)
- The queries use the **Sample Data** network.

## Indices and tables

- [Index](#)
- [Search Page](#)

## Previous versions of this page

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

## Parameters

Parameter	Type	Default	Description
<b>Edges SQL</b>	TEXT		Inner query as described below.

## Inner query

### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>• When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>• When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Result Columns

pgr\_connectedComponents & pgr\_strongComponents

Returns set of (seq, component, node)

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

pgr\_biconnectedComponents

Returns set of (seq, component, edge)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from <b>1</b> .
<b>component</b>	BIGINT	Component identifier. It is equal to the minimum edge identifier in the component.

Column	Type	Description
<b>edge</b>	BIGINT	Identifier of the edge.

pgr\_articulationPoints

Returns set of (node)

Column	Type	Description
<b>node</b>	BIGINT	Identifier of the vertex.

pgr\_bridges

Returns set of (edge)

Column	Type	Description
<b>edge</b>	BIGINT	Identifier of the edge that is a bridge.

See Also

## Indices and tables

- [Index](#)
- [Search Page](#)

## Contraction - Family of functions

- [pgr\\_contraction](#)

**pgr\_contraction**

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



Boost Graph Inside

## Availability

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

## Support

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

Description

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

The main Characteristics are:

- Process is done only on edges with positive costs.
- Does not return the full contracted graph
  - Only changes on the graph are returned
- Currently there are two types of contraction methods
  - Dead End Contraction
  - Linear Contraction
- The returned values include
  - the added edges by linear contraction.
  - the modified vertices by dead end contraction.

- The returned values are ordered as follows:
  - column *id* ascending when type = v
  - column *id* descending when type = e

## Signatures

## Summary

The `pgr_contraction` function has the following signature:

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

### Example:

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

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

## Parameters

Column	Type	Description
<b>Edges SQL</b>	TEXT	SQL query as described in <b>Inner query</b>
<b>Contraction Order</b>	ARRAY[ANY-INTEGER]	Ordered contraction operations. <ul style="list-style-type: none"> <li>• 1 = Dead end contraction</li> <li>• 2 = Linear contraction</li> </ul>

## Optional Parameters

Column	Type	Default	Description
<b>forbidden_vertices</b>	ARRAY[ANY-INTEGER]	Empty	Identifiers of vertices forbidden from contraction.
<b>max_cycles</b>	INTEGER	\(1\)	Number of times the contraction operations on <i>contraction_order</i> will be performed.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>• When <b>true</b> the graph is considered as <i>Directed</i>.</li> <li>• When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>• When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>• When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**  
SMALLINT, INTEGER, BIGINT

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

## Result Columns

RETURNS SETOF (type, id, contracted\_vertices, source, target, cost)

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

Column	Type	Description
<b>type</b>	TEXT	Type of the <i>id</i> . <ul style="list-style-type: none"> <li>'v' when the row is a vertex.</li> <li>'e' when the row is an edge.</li> </ul>
<b>id</b>	BIGINT	All numbers on this column are <b>DISTINCT</b> . <ul style="list-style-type: none"> <li>When <b>type</b> = 'v'. <ul style="list-style-type: none"> <li>Identifier of the modified vertex.</li> </ul> </li> <li>When <b>type</b> = 'e'. <ul style="list-style-type: none"> <li>Decreasing sequence starting from <b>-1</b>.</li> <li>Representing a pseudo <i>id</i> as is not incorporated in the set of original edges.</li> </ul> </li> </ul>
<b>contracted_vertices</b>	ARRAY[BIGINT]	Array of contracted vertex identifiers.
<b>source</b>	BIGINT	<ul style="list-style-type: none"> <li>When <b>type</b> = 'v': \(-1\)</li> <li>When <b>type</b> = 'e': Identifier of the source vertex of the current edge (<i>source</i>, <i>target</i>).</li> </ul>
<b>target</b>	BIGINT	<ul style="list-style-type: none"> <li>When <b>type</b> = 'v': \(-1\)</li> <li>When <b>type</b> = 'e': Identifier of the target vertex of the current edge (<i>source</i>, <i>target</i>).</li> </ul>
<b>cost</b>	FLOAT	<ul style="list-style-type: none"> <li>When <b>type</b> = 'v': \(-1\)</li> <li>When <b>type</b> = 'e': Weight of the current edge (<i>source</i>, <i>target</i>).</li> </ul>

#### Additional Examples

##### Example:

Only dead end contraction

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

##### Example:

Only linear contraction

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

See Also

- [Contraction - Family of functions](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### Previous versions of this page

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

#### Introduction

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

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

1. Dead end contraction

## 2. Linear contraction

Allowing the user to:

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

### Dead end contraction

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

Dead end

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

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

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

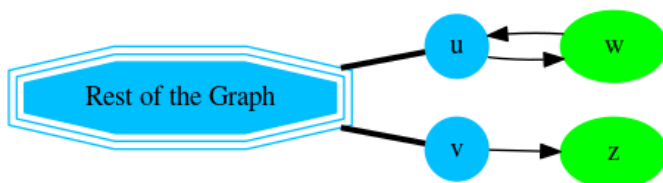
- **The number of adjacent vertices is 1.**
  - **There are no outgoing edges and has at least one incoming edge.**
  - **There are no incoming edges and has at least one outgoing edge.**

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

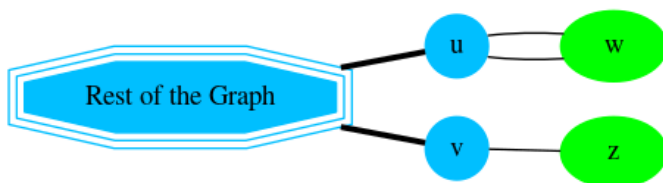
The number of adjacent vertices is 1.

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

### Directed graph



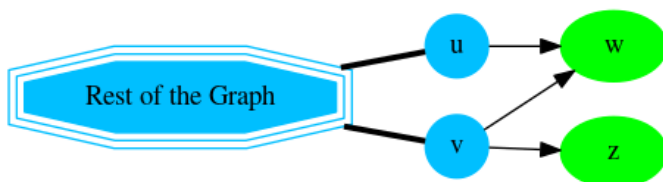
### Undirected graph



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

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

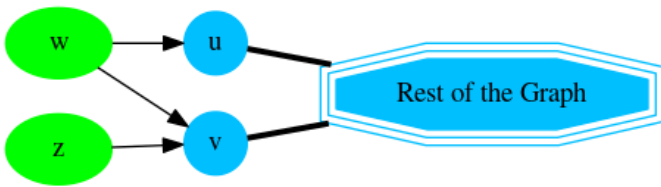
### Directed graph



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

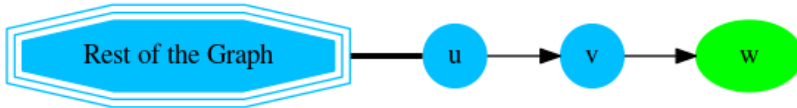
- The green nodes are **dead end** nodes
- The blue nodes have an unlimited number of incoming and outgoing edges.
- Considering that the nodes are *dead starts* nodes

### Directed graph

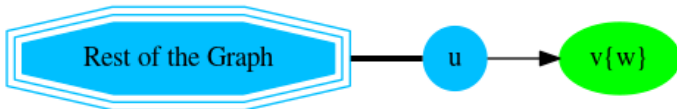


Operation: Dead End Contraction

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



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



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



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 *linear* node when

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

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

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

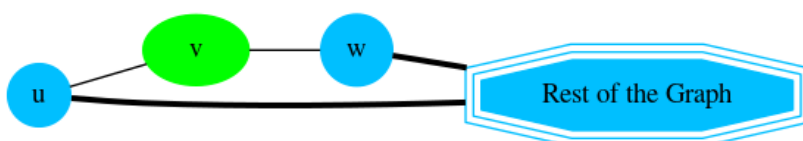
The number of adjacent vertices is 2.

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

**Directed**

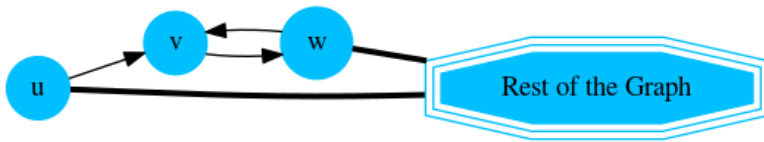


**Undirected**



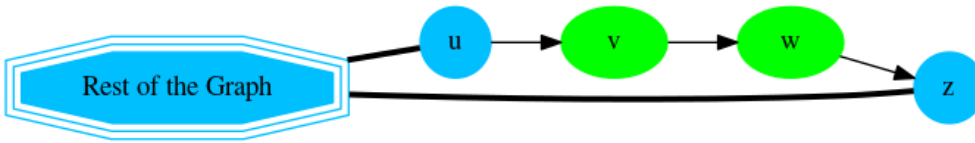
Linearity is symmetrical

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



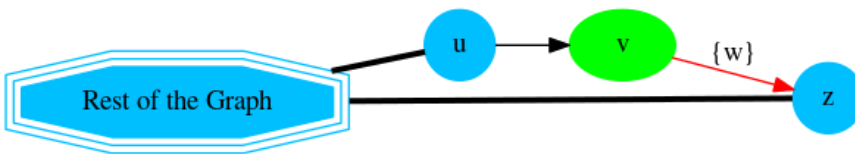
Operation: Linear Contraction

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



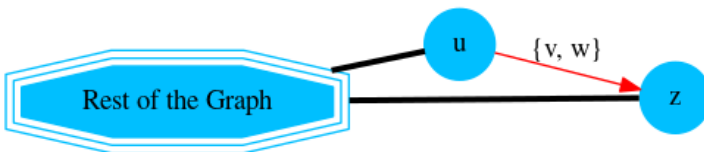
After contracting  $w$ ,

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



Contracting  $v$ :

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



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

**The cycle**

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

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

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

**Contracting Sample Data**

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

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

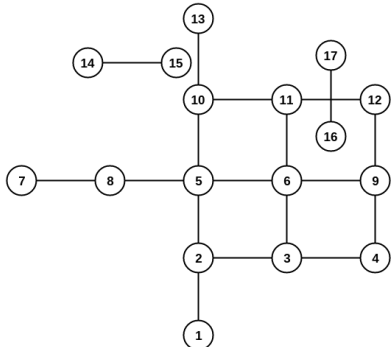
Construction of the graph in the database

**Original Data**

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

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

The original graph:



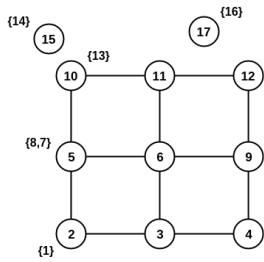
Contraction Results

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

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

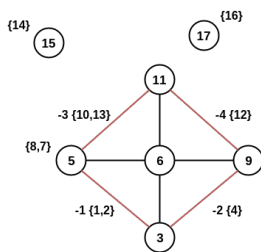
```
SELECT * FROM pgr_contraction(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  array[1,2], directed:=false);
type | id | contracted_vertices | source | target | cost
-----+-----+-----+-----+-----+-----
v | 5 | {7,8} | -1 | -1 | -1
v | 15 | {14} | -1 | -1 | -1
v | 17 | {16} |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

After doing the dead end contraction operation:



After doing the linear contraction operation to the graph above:





The process to create the contraction graph on the database:

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

Add additional columns

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

Column	Description
<b>contracted_vertices</b>	The vertices set belonging to the vertex/edge
<b>is_contracted</b>	On the <i>vertex</i> table <ul style="list-style-type: none"> <li>• when <b>true</b> the vertex is contracted, its not part of the contracted graph.</li> <li>• when <b>false</b> the vertex is not contracted, its part of the contracted graph.</li> </ul>
<b>is_new</b>	On the <i>edge</i> table: <ul style="list-style-type: none"> <li>• when <b>true</b> the edge was generated by the contraction algorithm. its part of the contracted graph.</li> <li>• when <b>false</b> the edge is an original edge, might be or not part of the contracted graph.</li> </ul>

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

Store contraction information

Store the **contraction results** in a table

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

Update the vertices and edge tables

### Update the *vertex* table using the contraction information

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

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

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

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

The modified `edge_table_vertices_pgr`.

```

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

```

### Update the *edge* table using the contraction information

Insert the new edges generated by `pgr_contraction`.

```

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

```

The modified `edge_table`.

```

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

```

The contracted graph

Vertices that belong to the contracted graph.

```

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

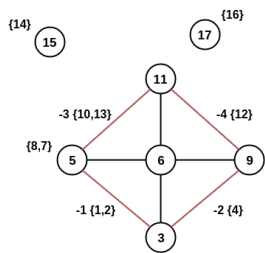
```

Edges that belong to the contracted graph.

```

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

```



Using the contracted graph

Using the contracted graph with `pgr_dijkstra`

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

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

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

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

```

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

```

### Case 1

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

```

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

```

### Case 2

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

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

```

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

```

### Case 3

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

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

```

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

```

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

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

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

```

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

```

### Case 1

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

```

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

```

### Case 2

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

The routing graph now has an edge connecting with node(4).

```

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

```

### Case 3

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

In this case, the contracted graph do not have an edge connecting with node(7).

```

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

```

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

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

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

```

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

```

### Case 1

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

```

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

```

### Case 2

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

```

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

```

### Case 3

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

Now, the routing graph has an edge connecting with node(7).

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

See Also

- <https://www.cs.cmu.edu/afs/cs/academic/class/15210-f12/www/lectures/lecture16.pdf>
- [https://algo2.iti.kit.edu/documents/routeplanning/geisberger\\_dipl.pdf](https://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf)
- The queries use **pgr\_contraction** function and the **Sample Data** network.


### Indices and tables

- [Index](#)
- [Search Page](#)

### Dijkstra - Family of functions

- **pgr\_dijkstra** - Dijkstra’s algorithm for the shortest paths.
- **pgr\_dijkstraCost** - Get the aggregate cost of the shortest paths.
- **pgr\_dijkstraCostMatrix** - Use pgr\_dijkstra to create a costs matrix.
- **pgr\_drivingDistance** - Use pgr\_dijkstra to calculate catchment 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-INTEGGER 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 seunce of vertices.

### pgr\_dijkstra

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



Boost Graph Inside

### Availability

- Version 3.0.0
  - **Official** functions
- Version 2.2.0
  - New **proposed** functions:
    - `pgr_dijkstra(One to Many)`
    - `pgr_dijkstra(Many to One)`

- o pgr\_dijkstra(Many to Many)
- o Version 2.1.0
  - o Signature change on pgr\_dijkstra(One to One)
- o Version 2.0.0
  - o **Official** pgr\_dijkstra(One to One)

## Support

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

## Description

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

The main characteristics are:

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

## Signatures

## Summary

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

## Using defaults

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

## Example:

From vertex  $\{2\}$  to vertex  $\{3\}$  on a **directed** graph

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

## One to One

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



**Example:**

From vertex  $\{2\}$  to vertex  $\{3\}$  on an **undirected** graph

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3,
  FALSE
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	2	1	0
2	2	3	-1	0	1

(2 rows)

## One to many

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

**Example:**

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

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edge_table',
  2, ARRAY[3,5],
  FALSE
);
```

seq	path_seq	end_vid	node	edge	cost	agg_cost
1	1	3	2	4	1	0
2	2	3	5	8	1	1
3	3	3	6	5	1	2
4	4	3	3	-1	0	3
5	1	5	2	4	1	0
6	2	5	5	-1	0	1

(6 rows)

## Many to One

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

**Example:**

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

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[2,11], 5
);
```

seq	path_seq	start_vid	node	edge	cost	agg_cost
1	1	2	2	4	1	0
2	2	2	5	-1	0	1
3	1	11	11	13	1	0
4	2	11	12	15	1	1
5	3	11	9	9	1	2
6	4	11	6	8	1	3
7	5	11	5	-1	0	4

(7 rows)

## Many to Many

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

**Example:**

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

```

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

```

#### Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When true Graph is considered <i>Directed</i></li> <li>When false the graph is considered as <i>Undirected</i>.</li> </ul>

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Return Columns

Returns set of (seq, path\_id, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from 1.
<b>path_id</b>	INT	Path identifier. Has value 1 for the first of a path. Used when there are multiple paths for the same start_vid to end_vid combination.
<b>path_seq</b>	INT	Relative position in the path. Has value 1 for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li>Many to One</li> <li>Many to Many</li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li>One to Many</li> <li>Many to Many</li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from start_vid to end_vid.
<b>edge</b>	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.

**Column Type Description****cost** FLOAT Cost to traverse from `node` using `edge` to the next node in the path sequence.**agg\_cost** FLOAT Aggregate cost from `start_v` to `node`.

## Additional Examples

The examples of this section are based on the **Sample Data** network.The examples include combinations from starting vertices 2 and 11 to ending vertices 3 and 5 in a directed and undirected graph with and with out `reverse_cost`.**Examples:**For queries marked as `directed` with `cost` and `reverse_cost` columnsThe examples in this section use the following **Network for queries marked as directed and cost and reverse\_cost columns are used**

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	4	1	0
2	2	5	8	1	1
3	3	6	9	1	2
4	4	9	16	1	3
5	5	4	3	1	4
6	6	3	-1	0	5

(6 rows)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 5
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	4	1	0
2	2	5	-1	0	1

(2 rows)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, ARRAY[3,5]
);
```

seq	path_seq	end_vid	node	edge	cost	agg_cost
1	1	3	2	4	1	0
2	2	3	5	8	1	1
3	3	3	6	9	1	2
4	4	3	9	16	1	3
5	5	3	4	3	1	4
6	6	3	3	-1	0	5
7	1	5	2	4	1	0
8	2	5	5	-1	0	1

(8 rows)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  11, 3
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	11	13	1	0
2	2	12	15	1	1
3	3	9	16	1	2
4	4	4	3	1	3
5	5	3	-1	0	4

(5 rows)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  11, 5
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	11	13	1	0
2	2	12	15	1	1
3	3	9	9	1	2
4	4	6	8	1	3
5	5	5	-1	0	4

(5 rows)

```
SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[2,11], 5
);
```

seq	path_seq	start_vid	node	edge	cost	agg_cost
-----	----------	-----------	------	------	------	----------

```

1 | 1 | 2 | 2 | 4 | 1 | 0
2 | 2 | 2 | 5 | -1 | 0 | 1
3 | 1 | 11 | 11 | 13 | 1 | 0
4 | 2 | 11 | 12 | 15 | 1 | 1
5 | 3 | 11 | 9 | 9 | 1 | 2
6 | 4 | 11 | 6 | 8 | 1 | 3
7 | 5 | 11 | 5 | -1 | 0 | 4

```

(7 rows)

```

SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[2, 11], ARRAY[3,5]
);

```

```

seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----+-----

```

```

1 | 1 | 2 | 3 | 2 | 4 | 1 | 0
2 | 2 | 2 | 3 | 5 | 8 | 1 | 1
3 | 3 | 2 | 3 | 6 | 9 | 1 | 2
4 | 4 | 2 | 3 | 9 | 16 | 1 | 3
5 | 5 | 2 | 3 | 4 | 3 | 1 | 4
6 | 6 | 2 | 3 | 3 | -1 | 0 | 5
7 | 1 | 2 | 5 | 2 | 4 | 1 | 0
8 | 2 | 2 | 5 | 5 | -1 | 0 | 1
9 | 1 | 11 | 3 | 11 | 13 | 1 | 0
10 | 2 | 11 | 3 | 12 | 15 | 1 | 1
11 | 3 | 11 | 3 | 9 | 16 | 1 | 2
12 | 4 | 11 | 3 | 4 | 3 | 1 | 3
13 | 5 | 11 | 3 | 3 | -1 | 0 | 4
14 | 1 | 11 | 5 | 11 | 13 | 1 | 0
15 | 2 | 11 | 5 | 12 | 15 | 1 | 1
16 | 3 | 11 | 5 | 9 | 9 | 1 | 2
17 | 4 | 11 | 5 | 6 | 8 | 1 | 3
18 | 5 | 11 | 5 | 5 | -1 | 0 | 4

```

(18 rows)

### Examples:

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

**Examples:**

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

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

```

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

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

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

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

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

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

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

```

**Examples:**

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

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

```

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

SELECT * FROM pgr_dijkstra(
  'SELECT id, source, target, cost FROM edge_table',
  2, 5,

```

```

FALSE
);
seq | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 | 1 | 2 | 4 | 1 | 0
 2 | 2 | 5 | -1 | 0 | 1
(2 rows)

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

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

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

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

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

```

Equivalences between signatures

#### Examples:

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

The examples in this section use the following:

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

### Examples:

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

The examples in this section use the following:



- **Network for queries marked as undirected and cost and reverse\_cost columns are used**

```

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

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

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

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

```

See Also

- [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
- The queries use the **Sample Data** network.

**Indices and tables**

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

**pgr\_dijkstraCost**

pgr\_dijkstraCost

Using Dijkstra algorithm implemented by Boost.Graph, and extract only the aggregate cost of the shortest path(s) found, for the combination of vertices given.



Boost Graph Inside

**Availability**

- Version 2.2.0
  - New **Official** function
- **Supported versions:** current(3.0) 2.6
- **Unsupported versions:** 2.5 2.4 2.3 2.3

**Description**

The pgr\_dijkstraCost algorithm, is a good choice to calculate the sum of the costs of the shortest path for a subset of pairs of nodes of the graph. We make use of the Boost’s implementation of dijkstra which runs in  $\mathcal{O}(V \log V + E)$  time.

The main characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of  $(start\_vid, end\_vid, agg\_cost)$ .
  - When the starting vertex and ending vertex are the same, there is no path.
    - The  $agg\_cost$  in the non included values  $(v, v)$  is 0
  - When the starting vertex and ending vertex are the different and there is no path.
    - The  $agg\_cost$  in the non included values  $(u, v)$  is  $(\infty)$
- Let be the case the values returned are stored in a table, so the unique index would be the pair  $(start\_vid, end\_vid)$ .
- For undirected graphs, the results are symmetric.
  - The  $agg\_cost$  of  $(u, v)$  is the same as for  $(v, u)$ .
- Any duplicated value in the  $start\_vids$  or  $end\_vids$  is ignored.
- The returned values are ordered:
  - $start\_vid$  ascending
  - $end\_vid$  ascending
- Running time:  $(O(|start\_vids| * (V \log V + E)))$

Signatures

## Summary

```
pgr_dijkstraCost(edges_sql, from_vid, to_vid [, directed])
pgr_dijkstraCost(edges_sql, from_vid, to_vids [, directed])
pgr_dijkstraCost(edges_sql, from_vids, to_vid [, directed])
pgr_dijkstraCost(edges_sql, from_vids, to_vids [, directed])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

## Using defaults

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

### Example:

From vertex  $(2)$  to vertex  $(3)$  on a **directed** graph

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

## One to One

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

### Example:

From vertex  $(2)$  to vertex  $(3)$  on an **undirected** graph

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

## One to Many

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

**Example:**

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

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

## Many to One

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

**Example:**

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

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

## Many to Many

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

**Example:**

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

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

## Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Column	Type	Default	Description
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

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

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

Additional Examples

**Example 1:**

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

```
SELECT * FROM pgr_dijkstraCost(
  'select id, source, target, cost, reverse_cost from edge_table',
  ARRAY[5, 3, 4, 3, 3, 4], ARRAY[3, 5, 3, 4]);
start_vid | end_vid | agg_cost
-----+-----+-----
3 | 4 | 3
3 | 5 | 2
4 | 3 | 1
4 | 5 | 3
5 | 3 | 4
5 | 4 | 3
(6 rows)
```

**Example 2:**

Making `start_vids` the same as `end_vids`

```
SELECT * FROM pgr_dijkstraCost(
  'select id, source, target, cost, reverse_cost from edge_table',
  ARRAY[5, 3, 4], ARRAY[5, 3, 4]);
start_vid | end_vid | agg_cost
-----+-----+-----
3 | 4 | 3
3 | 5 | 2
4 | 3 | 1
4 | 5 | 3
5 | 3 | 4
5 | 4 | 3
(6 rows)
```

See Also

- [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
- **Sample Data** network.

**Indices and tables**

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

**pgr\_dijkstraCostMatrix**

`pgr_dijkstraCostMatrix` - Calculates the a cost matrix using `pgr_dijktras`.



### Availability

- Version 3.0.0
  - **Official** function
- Version 2.3.0
  - New **proposed** function
- **Supported versions:** current(**3.0**)
- **Unsupported versions:** **2.6 2.5 2.4 2.3**

### Description

Using Dijkstra algorithm, calculate and return a cost matrix.

### Signatures

### Summary

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

### Using defaults

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

### Example:

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

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

### Complete Signature

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

### Example:

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

```

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

```

#### Parameters

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

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

#### Where:

##### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

##### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Return Columns

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

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

#### Additional Examples

##### Example:

Use with `tsp`

```

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

```

See Also

- [Dijkstra - Family of functions](#)
- [Cost Matrix - Category](#)
- [Traveling Sales Person - Family of functions](#)
- The queries use the [Sample Data](#) network.

## Indices and tables

- [Index](#)
- [Search Page](#)

### pgr\_drivingDistance

`pgr_drivingDistance` - Returns the driving distance from a start node.



Boost Graph Inside

## Availability

- Version 2.1.0:
  - Signature change `pgr_drivingDistance`(single vertex)
  - New **Official** `pgr_drivingDistance`(multiple vertices)
- Version 2.0.0:
  - **Official** `pgr_drivingDistance`(single vertex)

## Support

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

## Description

Using the Dijkstra algorithm, extracts all the nodes that have costs less than or equal to the value `distance`. The edges extracted will conform to the corresponding spanning tree.

## Signatures

## Summary

```

pgr_drivingDistance(edges_sql, start_vid, distance [, directed])
pgr_drivingDistance(edges_sql, start_vids, distance [, directed] [, equicost])
RETURNS SET OF (seq, [start_vid,] node, edge, cost, agg_cost)

```

## Using defaults

```

pgr_drivingDistance(edges_sql, start_vid, distance)
RETURNS SET OF (seq, node, edge, cost, agg_cost)

```

**Example:**

TBD

Single Vertex

```
pgr_drivingDistance(edges_sql, start_vid, distance [, directed])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

**Example:**

TBD

Multiple Vertices

```
pgr_drivingDistance(edges_sql, start_vids, distance, [, directed] [, equicost])
RETURNS SET OF (seq, start_vid, node, edge, cost, agg_cost)
```

**Example:**

TBD

Parameters

Column	Type	Description
<b>edges_sql</b>	TEXT	SQL query as described above.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex.
<b>start_vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the starting vertices.
<b>distance</b>	FLOAT	Upper limit for the inclusion of the node in the result.
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>equicost</b>	BOOLEAN	(optional). When <code>true</code> the node will only appear in the closest <code>start_vid</code> list. Default is <code>false</code> which resembles several calls using the single starting point signatures. Tie brakes are arbitrary.

Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICALS:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq [, start\_v], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INTEGER	Sequential value starting from <b>1</b> .
<b>start_vid</b>	INTEGER	Identifier of the starting vertex.
<b>node</b>	BIGINT	Identifier of the node in the path within the limits from <code>start_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to arrive to <code>node</code> . 0 when the <code>node</code> is the <code>start_vid</code> .
<b>cost</b>	FLOAT	Cost to traverse <code>edge</code> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

Additional Examples

**Example:**For queries marked as `directed` with `cost` and `reverse_cost` columns



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

```

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | 2 | -1 | 0 | 0
 2 | 1 | 1 | 1 | 1
 3 | 5 | 4 | 1 | 1
 4 | 6 | 8 | 1 | 2
 5 | 8 | 7 | 1 | 2
 6 | 10 | 10 | 1 | 2
 7 | 7 | 6 | 1 | 3
 8 | 9 | 9 | 1 | 3
 9 | 11 | 12 | 1 | 3
10 | 13 | 14 | 1 | 3
(10 rows)

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  13, 3
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | 13 | -1 | 0 | 0
 2 | 10 | 14 | 1 | 1
 3 | 5 | 10 | 1 | 2
 4 | 11 | 12 | 1 | 2
 5 | 2 | 4 | 1 | 3
 6 | 6 | 8 | 1 | 3
 7 | 8 | 7 | 1 | 3
 8 | 12 | 13 | 1 | 3
(8 rows)

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

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  array[2,13], 3, equicost:=true
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 | 2 | 2 | -1 | 0 | 0
 2 | 2 | 1 | 1 | 1 | 1
 3 | 2 | 5 | 4 | 1 | 1
 4 | 2 | 6 | 8 | 1 | 2
 5 | 2 | 8 | 7 | 1 | 2
 6 | 2 | 7 | 6 | 1 | 3
 7 | 2 | 9 | 9 | 1 | 3
 8 | 13 | 13 | -1 | 0 | 0
 9 | 13 | 10 | 14 | 1 | 1
10 | 13 | 11 | 12 | 1 | 2
11 | 13 | 12 | 13 | 1 | 3
(11 rows)

```

**Example:**  
 For queries marked as `undirected` with `cost` and `reverse_cost` columns

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

```

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3, false
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 |  2 |  -1 |  0 |      0
 2 |  1 |   1 |  1 |      1
 3 |  3 |   2 |  1 |      1
 4 |  5 |   4 |  1 |      1
 5 |  4 |   3 |  1 |      2
 6 |  6 |   8 |  1 |      2
 7 |  8 |   7 |  1 |      2
 8 | 10 |  10 |  1 |      2
 9 |  7 |   6 |  1 |      3
10 |  9 |  16 |  1 |      3
11 | 11 |  12 |  1 |      3
12 | 13 |  14 |  1 |      3
(12 rows)

```

```

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  13, 3, false
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | 13 |  -1 |  0 |      0
 2 | 10 |  14 |  1 |      1
 3 |  5 |  10 |  1 |      2
 4 | 11 |  12 |  1 |      2
 5 |  2 |   4 |  1 |      3
 6 |  6 |   8 |  1 |      3
 7 |  8 |   7 |  1 |      3
 8 | 12 |  13 |  1 |      3
(8 rows)

```

```

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  array[2,13], 3, false
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 |  2 |  2 |  -1 |  0 |      0
 2 |  2 |  1 |   1 |  1 |      1
 3 |  2 |  3 |   2 |  1 |      1
 4 |  2 |  5 |   4 |  1 |      1
 5 |  2 |  4 |   3 |  1 |      2
 6 |  2 |  6 |   8 |  1 |      2
 7 |  2 |  8 |   7 |  1 |      2
 8 |  2 | 10 |  10 |  1 |      2
 9 |  2 |  7 |   6 |  1 |      3
10 |  2 |  9 |  16 |  1 |      3
11 |  2 | 11 |  12 |  1 |      3
12 |  2 | 13 |  14 |  1 |      3
13 | 13 | 13 |  -1 |  0 |      0
14 | 13 | 10 |  14 |  1 |      1
15 | 13 |  5 |  10 |  1 |      2
16 | 13 | 11 |  12 |  1 |      2
17 | 13 |  2 |   4 |  1 |      3
18 | 13 |  6 |   8 |  1 |      3
19 | 13 |  8 |   7 |  1 |      3
20 | 13 | 12 |  13 |  1 |      3
(20 rows)

```

```

SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  array[2,13], 3, false, equicost:=true
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 |  2 |  2 |  -1 |  0 |      0
 2 |  2 |  1 |   1 |  1 |      1
 3 |  2 |  3 |   2 |  1 |      1
 4 |  2 |  5 |   4 |  1 |      1
 5 |  2 |  4 |   3 |  1 |      2
 6 |  2 |  6 |   8 |  1 |      2
 7 |  2 |  8 |   7 |  1 |      2
 8 |  2 |  7 |   6 |  1 |      3
 9 |  2 |  9 |  16 |  1 |      3
10 | 13 | 13 |  -1 |  0 |      0
11 | 13 | 10 |  14 |  1 |      1
12 | 13 | 11 |  12 |  1 |      2
13 | 13 | 12 |  13 |  1 |      3
(13 rows)

```

**Example:**

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

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

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  2,3
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
1 | 2 | -1 | 0 | 0
2 | 5 | 4 | 1 | 1
3 | 6 | 8 | 1 | 2
4 | 10 | 10 | 1 | 2
5 | 9 | 9 | 1 | 3
6 | 11 | 11 | 1 | 3
7 | 13 | 14 | 1 | 3
(7 rows)
```

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  13,3
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
1 | 13 | -1 | 0 | 0
(1 row)
```

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  array[2,13], 3
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
1 | 2 | 2 | -1 | 0 | 0
2 | 2 | 5 | 4 | 1 | 1
3 | 2 | 6 | 8 | 1 | 2
4 | 2 | 10 | 10 | 1 | 2
5 | 2 | 9 | 9 | 1 | 3
6 | 2 | 11 | 11 | 1 | 3
7 | 2 | 13 | 14 | 1 | 3
8 | 13 | 13 | -1 | 0 | 0
(8 rows)
```

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  array[2,13], 3, equicost:=true
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
1 | 2 | 2 | -1 | 0 | 0
2 | 2 | 5 | 4 | 1 | 1
3 | 2 | 6 | 8 | 1 | 2
4 | 2 | 10 | 10 | 1 | 2
5 | 2 | 9 | 9 | 1 | 3
6 | 2 | 11 | 11 | 1 | 3
7 | 13 | 13 | -1 | 0 | 0
(7 rows)
```

### Example:

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

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

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  2, 3, false
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 |  2 |  -1 |   0 |         0
 2 |  1 |   1 |   1 |         1
 3 |  5 |   4 |   1 |         1
 4 |  6 |   8 |   1 |         2
 5 |  8 |   7 |   1 |         2
 6 | 10 |  10 |   1 |         2
 7 |  3 |   5 |   1 |         3
 8 |  7 |   6 |   1 |         3
 9 |  9 |   9 |   1 |         3
10 | 11 |  12 |   1 |         3
11 | 13 |  14 |   1 |         3
(11 rows)
```

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  13, 3, false
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | 13 |  -1 |   0 |         0
 2 | 10 |  14 |   1 |         1
 3 |  5 |  10 |   1 |         2
 4 | 11 |  12 |   1 |         2
 5 |  2 |   4 |   1 |         3
 6 |  6 |   8 |   1 |         3
 7 |  8 |   7 |   1 |         3
 8 | 12 |  13 |   1 |         3
(8 rows)
```

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  array[2,13], 3, false
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 |  2 |  2 |  -1 |   0 |         0
 2 |  2 |  1 |   1 |   1 |         1
 3 |  2 |  5 |   4 |   1 |         1
 4 |  2 |  6 |   8 |   1 |         2
 5 |  2 |  8 |   7 |   1 |         2
 6 |  2 | 10 |  10 |   1 |         2
 7 |  2 |  3 |   5 |   1 |         3
 8 |  2 |  7 |   6 |   1 |         3
 9 |  2 |  9 |   9 |   1 |         3
10 |  2 | 11 |  12 |   1 |         3
11 |  2 | 13 |  14 |   1 |         3
12 | 13 | 13 |  -1 |   0 |         0
13 | 13 | 10 |  14 |   1 |         1
14 | 13 |  5 |  10 |   1 |         2
15 | 13 | 11 |  12 |   1 |         2
16 | 13 |  2 |   4 |   1 |         3
17 | 13 |  6 |   8 |   1 |         3
18 | 13 |  8 |   7 |   1 |         3
19 | 13 | 12 |  13 |   1 |         3
(19 rows)
```

```
SELECT * FROM pgr_drivingDistance(
  'SELECT id, source, target, cost FROM edge_table',
  array[2,13], 3, false, equicost:=true
);
seq | from_v | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 |  2 |  2 |  -1 |   0 |         0
 2 |  2 |  1 |   1 |   1 |         1
 3 |  2 |  5 |   4 |   1 |         1
 4 |  2 |  6 |   8 |   1 |         2
 5 |  2 |  8 |   7 |   1 |         2
 6 |  2 |  3 |   5 |   1 |         3
 7 |  2 |  7 |   6 |   1 |         3
 8 |  2 |  9 |   9 |   1 |         3
 9 | 13 | 13 |  -1 |   0 |         0
10 | 13 | 10 |  14 |   1 |         1
11 | 13 | 11 |  12 |   1 |         2
12 | 13 | 12 |  13 |   1 |         3
(12 rows)
```

See Also

- [pgr\\_alphaShape](#) - Alpha shape computation
- [Sample Data](#) network.

## Indices and tables

- [Index](#)

## Search Page

pgr\_KSP

pgr\_KSP — Returns the “K” shortest paths.



Boost Graph Inside

### Availability

- Version 2.1.0
  - Signature change
    - Old signature no longer supported
- Version 2.0.0
  - Official** function

### Support

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

### Description

The K shortest path routing algorithm based on Yen’s algorithm. “K” is the number of shortest paths desired.

### Signatures

### Summary

```
pgr_KSP(edges_sql, start_vid, end_vid, K [, directed] [, heap_paths])  
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

### Using defaults

```
pgr_ksp(edges_sql, start_vid, end_vid, K);  
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

#### Example:

TBD

### Complete Signature

```
pgr_KSP(edges_sql, start_vid, end_vid, K [, directed] [, heap_paths])  
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

#### Example:

TBD

### Parameters

Column	Type	Description
<b>edges_sql</b>	TEXT	SQL query as described above.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex.
<b>end_vid</b>	BIGINT	Identifier of the ending vertex.
<b>k</b>	INTEGER	The desired number of paths.
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>heap_paths</b>	BOOLEAN	(optional). When <code>true</code> returns all the paths stored in the process heap. Default is <code>false</code> which only returns <code>k</code> paths.

Roughly, if the shortest path has `N` edges, the heap will contain about than `N * k` paths for small value of `k` and `k > 1`.

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of (seq, path\_seq, path\_id, node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INTEGER	Sequential value starting from <b>1</b> .
<b>path_seq</b>	INTEGER	Relative position in the path of <code>node</code> and <code>edge</code> . Has value <b>1</b> for the beginning of a path.
<b>path_id</b>	BIGINT	Path identifier. The ordering of the paths For two paths <i>i</i> , <i>j</i> if <i>i</i> < <i>j</i> then <code>agg_cost(i) &lt;= agg_cost(j)</code> .
<b>node</b>	BIGINT	Identifier of the node in the path.
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <b>-1</b> for the last node of the route.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

Additional Examples

**Example:**

To handle the one flag to choose signatures

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

```

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 12, 2,
  directed:=true
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
(10 rows)

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 12, 2
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
(10 rows)

```

**Example:**

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

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

```

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 12, 2
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 2 | 4 | 1 | 0
 2 | 1 | 2 | 5 | 8 | 1 | 1
 3 | 1 | 3 | 6 | 9 | 1 | 2
 4 | 1 | 4 | 9 | 15 | 1 | 3
 5 | 1 | 5 | 12 | -1 | 0 | 4
 6 | 2 | 1 | 2 | 4 | 1 | 0
 7 | 2 | 2 | 5 | 8 | 1 | 1
 8 | 2 | 3 | 6 | 11 | 1 | 2
 9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
(10 rows)

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 12, 2, heap_paths:=true
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 2 | 4 | 1 | 0
 2 | 1 | 2 | 5 | 8 | 1 | 1
 3 | 1 | 3 | 6 | 9 | 1 | 2
 4 | 1 | 4 | 9 | 15 | 1 | 3
 5 | 1 | 5 | 12 | -1 | 0 | 4
 6 | 2 | 1 | 2 | 4 | 1 | 0
 7 | 2 | 2 | 5 | 8 | 1 | 1
 8 | 2 | 3 | 6 | 11 | 1 | 2
 9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
11 | 3 | 1 | 2 | 4 | 1 | 0
12 | 3 | 2 | 5 | 10 | 1 | 1
13 | 3 | 3 | 10 | 12 | 1 | 2
14 | 3 | 4 | 11 | 13 | 1 | 3
15 | 3 | 5 | 12 | -1 | 0 | 4
(15 rows)

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 12, 2, true, true
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 2 | 4 | 1 | 0
 2 | 1 | 2 | 5 | 8 | 1 | 1
 3 | 1 | 3 | 6 | 9 | 1 | 2
 4 | 1 | 4 | 9 | 15 | 1 | 3
 5 | 1 | 5 | 12 | -1 | 0 | 4
 6 | 2 | 1 | 2 | 4 | 1 | 0
 7 | 2 | 2 | 5 | 8 | 1 | 1
 8 | 2 | 3 | 6 | 11 | 1 | 2
 9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
11 | 3 | 1 | 2 | 4 | 1 | 0
12 | 3 | 2 | 5 | 10 | 1 | 1
13 | 3 | 3 | 10 | 12 | 1 | 2
14 | 3 | 4 | 11 | 13 | 1 | 3
15 | 3 | 5 | 12 | -1 | 0 | 4
(15 rows)

```

**Examples:**

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

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

```

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 12, 2, directed:=false
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 2 | 2 | 1 | 0
 2 | 1 | 2 | 3 | 3 | 1 | 1
 3 | 1 | 3 | 4 | 16 | 1 | 2
 4 | 1 | 4 | 9 | 15 | 1 | 3
 5 | 1 | 5 | 12 | -1 | 0 | 4
 6 | 2 | 1 | 2 | 4 | 1 | 0
 7 | 2 | 2 | 5 | 10 | 1 | 1
 8 | 2 | 3 | 10 | 12 | 1 | 2
 9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
(10 rows)

```

```

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

```

**Example:**

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

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



```

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

```

```

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost FROM edge_table',
  2, 12, 2
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
(10 rows)

```

```

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost FROM edge_table',
  2, 12, 2, heap_paths:=true
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
11 | 3 | 1 | 2 | 4 | 1 | 0
12 | 3 | 2 | 5 | 10 | 1 | 1
13 | 3 | 3 | 10 | 12 | 1 | 2
14 | 3 | 4 | 11 | 13 | 1 | 3
15 | 3 | 5 | 12 | -1 | 0 | 4
(15 rows)

```

```

SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost FROM edge_table',
  2, 12, 2, true, true
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
11 | 3 | 1 | 2 | 4 | 1 | 0
12 | 3 | 2 | 5 | 10 | 1 | 1
13 | 3 | 3 | 10 | 12 | 1 | 2
14 | 3 | 4 | 11 | 13 | 1 | 3
15 | 3 | 5 | 12 | -1 | 0 | 4
(15 rows)

```

**Example:**

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

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

```
SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost FROM edge_table',
  2, 12, 2, directed:=false
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
(10 rows)
```

```
SELECT * FROM pgr_KSP(
  'SELECT id, source, target, cost FROM edge_table',
  2, 12, 2, directed:=false, heap_paths:=true
);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 4 | 1 | 0
2 | 1 | 2 | 5 | 8 | 1 | 1
3 | 1 | 3 | 6 | 9 | 1 | 2
4 | 1 | 4 | 9 | 15 | 1 | 3
5 | 1 | 5 | 12 | -1 | 0 | 4
6 | 2 | 1 | 2 | 4 | 1 | 0
7 | 2 | 2 | 5 | 8 | 1 | 1
8 | 2 | 3 | 6 | 11 | 1 | 2
9 | 2 | 4 | 11 | 13 | 1 | 3
10 | 2 | 5 | 12 | -1 | 0 | 4
11 | 3 | 1 | 2 | 4 | 1 | 0
12 | 3 | 2 | 5 | 10 | 1 | 1
13 | 3 | 3 | 10 | 12 | 1 | 2
14 | 3 | 4 | 11 | 13 | 1 | 3
15 | 3 | 5 | 12 | -1 | 0 | 4
(15 rows)
```

See Also

- [https://en.wikipedia.org/wiki/K\\_shortest\\_path\\_routing](https://en.wikipedia.org/wiki/K_shortest_path_routing)
- **Sample Data** network.

## Indices and tables

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

**pgr\_dijkstraVia** - Proposed

**pgr\_dijkstraVia** — Using dijkstra algorithm, it finds the route that goes through a list of vertices.



### Warning

Proposed functions for next mayor release.

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



Boost Graph Inside

## Availability

- Version 2.2.0
  - New **proposed** function

## Support

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

## Description

Given a list of vertices and a graph, this function is equivalent to finding the shortest path between(vertex\_i) and (vertex\_{i+1}) for all (i < size\_of(vertex\_via)).

The paths represents the sections of the route.

## Signatures

## Summary

```
pgr_dijkstraVia(edges_sql, via_vertices [, directed] [, strict] [, U_turn_on_edge])
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
node, edge, cost, agg_cost, route_agg_cost)
OR EMPTY SET
```

## Using default

```
pgr_dijkstraVia(edges_sql, via_vertices)
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
node, edge, cost, agg_cost, route_agg_cost)
OR EMPTY SET
```

## Example:

Find the route that visits the vertices (1, 3, 9) in that order

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

## Complete Signature

```
pgr_dijkstraVia(edges_sql, via_vertices [, directed] [, strict] [, U_turn_on_edge])
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
node, edge, cost, agg_cost, route_agg_cost)
OR EMPTY SET
```

## Example:

Find the route that visits the vertices (1, 3, 9) in that order on an **undirected** graph, avoiding U-turns when possible

```
SELECT * FROM pgr_dijkstraVia(
'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
ARRAY[1, 3, 9], false, strict:=true, U_turn_on_edge:=false
);
seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost | route_agg_cost
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 0 | 0
 2 | 1 | 2 | 1 | 3 | 2 | 2 | 1 | 1 | 1
 3 | 1 | 3 | 1 | 3 | 3 | -1 | 0 | 2 | 2
 4 | 2 | 1 | 3 | 9 | 3 | 3 | 1 | 0 | 2
 5 | 2 | 2 | 3 | 9 | 4 | 16 | 1 | 1 | 3
 6 | 2 | 3 | 3 | 9 | 9 | -2 | 0 | 2 | 4
(6 rows)
```

## Parameters

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

Parameter	Type	Default	Description
<b>via_vertices</b>	ARRAY[ANY-INTEGER]		Array of ordered vertices identifiers that are going to be visited.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>
<b>strict</b>	BOOLEAN	false	<ul style="list-style-type: none"> <li>When <b>false</b> ignores missing paths returning all paths found</li> <li>When <b>true</b> if a path is missing stops and returns <i>EMPTY SET</i></li> </ul>
<b>U_turn_on_edge</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> 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 <i>id</i> is allowed.</li> <li>When <b>false</b> 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 <i>id</i> is used when no other path is found.</li> </ul>

Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (start\_vid, end\_vid, agg\_cost)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from 1.
<b>path_pid</b>	BIGINT	Identifier of the path.
<b>path_seq</b>	BIGINT	Sequential value starting from 1 for the path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex of the path.
<b>end_vid</b>	BIGINT	Identifier of the ending vertex of the path.
<b>node</b>	BIGINT	Identifier of the node in the path from start_vid to end_vid.
<b>edge</b>	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.
<b>cost</b>	FLOAT	Cost to traverse from node using edge to the next node in the route sequence.
<b>agg_cost</b>	FLOAT	Total cost from start_vid to end_vid of the path.
<b>route_agg_cost</b>	FLOAT	Total cost from start_vid of path_pid = 1 to end_vid of the current path_pid .

Additional Examples

**Example 1:**

Find the route that visits the vertices\({1, 5, 3, 9, 4}\) in that order

```

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

```

### Example 2:

What's the aggregate cost of the third path?

```

SELECT agg_cost FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
  ARRAY[1, 5, 3, 9, 4]
)
WHERE path_id = 3 AND edge < 0;
agg_cost
-----
      2
(1 row)

```

### Example 3:

What's the route's aggregate cost of the route at the end of the third path?

```

SELECT route_agg_cost FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
  ARRAY[1, 5, 3, 9, 4]
)
WHERE path_id = 3 AND edge < 0;
route_agg_cost
-----
      8
(1 row)

```

### Example 4:

How are the nodes visited in the route?

```

SELECT row_number() over () as node_seq, node
FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
  ARRAY[1, 5, 3, 9, 4]
)
WHERE edge <> -1 ORDER BY seq;
node_seq | node
-----+-----
      1 |    1
      2 |    2
      3 |    5
      4 |    6
      5 |    9
      6 |    4
      7 |    3
      8 |    6
      9 |    9
     10 |    4
(10 rows)

```

### Example 5:

What are the aggregate costs of the route when the visited vertices are reached?

```
SELECT path_id, route_agg_cost FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
  ARRAY[1, 5, 3, 9, 4]
)
WHERE edge < 0;
path_id | route_agg_cost
-----+-----
1 | 2
2 | 6
3 | 8
4 | 9
(4 rows)
```

### Example 6:

Show the route's seq and aggregate cost and a status of "passes in front" or "visits" node(9)

```
SELECT seq, route_agg_cost, node, agg_cost ,
CASE WHEN edge = -1 THEN 'visits'
ELSE 'passes in front'
END as status
FROM pgr_dijkstraVia(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
  ARRAY[1, 5, 3, 9, 4])
WHERE node = 9 and (agg_cost <= 0 or seq = 1);
seq | route_agg_cost | node | agg_cost | status
-----+-----+-----+-----+-----
6 | 4 | 9 | 2 | passes in front
11 | 8 | 9 | 2 | visits
(2 rows)

ROLLBACK;
ROLLBACK
```

See Also

- [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
- **Sample Data** network.

### Indices and tables

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

### Previous versions of this page

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

### The problem definition (Advanced documentation)

Given the following query:

```
pgr_dijkstra(\(sql, start_{vid}, end_{vid}, directed\))
```

where  $\text{\(sql\}} = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $\text{\(source = \bigcup source\_i\)}$ ,
- $\text{\(target = \bigcup target\_i\)}$ ,

The graphs are defined as follows:

### Directed graph

The weighted directed graph,  $\text{\(G\_d(V,E)\)}$ , is defined by:

- the set of vertices  $\text{\(V\)}$ 
  - $\text{\(V = source \cup target \cup \{start_{vid}\} \cup \{end_{vid}\}\)}$
- the set of edges  $\text{\(E\)}$ 
  - $\text{\(E = \begin{cases} \text{\( \{(source_i, target_i, cost_i) \} \text{ when } cost \geq 0 \} \} \quad \text{\( \{(source_i, target_i, cost_i) \} \text{ when } cost \geq 0 \} \} \quad \text{\( \{(source_i, target_i, cost_i) \} \text{ when } cost \geq 0 \} \} \quad \text{\( \{(target_i, source_i, reverse\_cost_i) \} \text{ when } reverse\_cost_i \geq 0 \} \} \quad \text{\( \{(target_i, source_i, reverse\_cost_i) \} \text{ when } reverse\_cost_i \geq 0 \} \} \end{cases}\)}$

### Undirected graph

The weighted undirected graph,  $\text{\(G\_u(V,E)\)}$ , is defined by:

- the set of vertices  $V$ 
  - $V = \text{source} \cup \text{target} \cup \{\text{start}_v\} \cup \{\text{end}_v\}$
- the set of edges  $E$ 
  - $E = \begin{cases} \text{ } \\ \{(source_i, target_i, cost_i) \mid \text{cost} \geq 0\} \cup \{(target_i, source_i, cost_i) \mid \text{cost} \geq 0\} \cup \{(target_i, source_i, reverse\_cost_i) \mid \text{reverse\_cost}_i \geq 0\} \cup \{(source_i, target_i, reverse\_cost_i) \mid \text{reverse\_cost}_i \geq 0\} \end{cases}$

## The problem

Given:

- $\text{start}_v \in V$  a starting vertex
- $\text{end}_v \in V$  an ending vertex
- $G(V,E) = \begin{cases} G_d(V,E) & \text{if } \text{directed} = \text{true} \\ G_u(V,E) & \text{if } \text{directed} = \text{false} \end{cases}$

Then:

- $\pi = (\text{path\_seq}_i, \text{node}_i, \text{edge}_i, \text{cost}_i, \text{agg\_cost}_i)$

where:

- $\text{path\_seq}_i = i$
- $\text{path\_seq}_{|\pi|} = |\pi|$
- $\text{node}_i \in V$
- $\text{node}_1 = \text{start}_v$
- $\text{node}_{|\pi|} = \text{end}_v$
- $\forall i \neq |\pi|, (node_i, node_{i+1}, cost_i) \in E$
- $\text{edge}_i = \begin{cases} id_{(node_i, node_{i+1}, cost_i)} & \text{if } i \neq |\pi| - 1 \\ \text{ } & \text{if } i = |\pi| - 1 \end{cases}$
- $\text{cost}_i = \text{cost}_{(node_i, node_{i+1})}$
- $\text{agg\_cost}_i = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{k=1}^i \text{cost}_{(node_{k-1}, node_k)} & \text{if } i \neq 1 \end{cases}$

In other words: The algorithm returns a the shortest path between  $\text{start}_v$  and  $\text{end}_v$ , if it exists, in terms of a sequence of nodes and of edges,

- $\text{path\_seq}$  indicates the relative position in the path of the  $(node)$  or  $(edge)$ .
- $\text{cost}$  is the cost of the edge to be used to go to the next node.
- $\text{agg\_cost}$  is the cost from the  $\text{start}_v$  up to the node.

If there is no path, the resulting set is empty.

See Also

## Indices and tables

- [Index](#)
- [Search Page](#)

## Flow - Family of functions

- pgr\_maxFlow** - Only the Max flow calculation using Push and Relabel algorithm.
- pgr\_boykovKolmogorov** - Boykov and Kolmogorov with details of flow on edges.
- pgr\_edmondsKarp** - Edmonds and Karp algorithm with details of flow on edges.
- pgr\_pushRelabel** - Push and relabel algorithm with details of flow on edges.
- Applications
  - pgr\_edgeDisjointPaths** - Calculates edge disjoint paths between two groups of vertices.
  - pgr\_maxCardinalityMatch** - Calculates a maximum cardinality matching in a graph.

## Experimental



### Warning

Possible server crash

- These functions might create a server crash



### Warning

## Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

- **pgr\_maxFlowMinCost - Experimental** - Details of flow and cost on edges.
- **pgr\_maxFlowMinCost\_Cost - Experimental** - Only the Min Cost calculation.

### pgr\_maxFlow

`pgr_maxFlow` — Calculates the maximum flow in a directed graph from the source(s) to the targets(s) using the Push Relabel algorithm.



Boost Graph Inside

### Availability

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

### Support

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

### Description

#### The main characteristics are:

- The graph is **directed**.
- Calculates the maximum flow from the *source(s)* to the *target(s)*.
  - When the maximum flow is **0** then there is no flow and **0** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Uses the **pgr\_pushRelabel** algorithm.
- Running time:  $O(V^3)$

### Signatures

### Summary

```
pgr_maxFlow(Edges SQL, source, target)
pgr_maxFlow(Edges SQL, sources, target)
pgr_maxFlow(Edges SQL, source, targets)
pgr_maxFlow(Edges SQL, sources, targets)
RETURNS BIGINT
```

### One to One

```
pgr_maxFlow(Edges SQL, source, target)
RETURNS BIGINT
```



**Example:**

From vertex  $\{(6)\}$  to vertex  $\{(11)\}$

```
SELECT * FROM pgr_maxFlow(
  'SELECT id,
   source,
   target,
   capacity,
   reverse_capacity
  FROM edge_table'
  , 6, 11
);
pgr_maxflow
-----
      230
(1 row)
```

**One to Many**

```
pgr_maxFlow(Edges SQL, source, targets)
RETURNS BIGINT
```

**Example:**

From vertex  $\{(6)\}$  to vertices  $\{(11, 1, 13)\}$

```
SELECT * FROM pgr_maxFlow(
  'SELECT id,
   source,
   target,
   capacity,
   reverse_capacity
  FROM edge_table'
  , 6, ARRAY[11, 1, 13]
);
pgr_maxflow
-----
      340
(1 row)
```

**Many to One**

```
pgr_maxFlow(Edges SQL, sources, target)
RETURNS BIGINT
```

**Example:**

From vertices  $\{(6, 8, 12)\}$  to vertex  $\{(11)\}$

```
SELECT * FROM pgr_maxFlow(
  'SELECT id,
   source,
   target,
   capacity,
   reverse_capacity
  FROM edge_table'
  , ARRAY[6, 8, 12], 11
);
pgr_maxflow
-----
      230
(1 row)
```

**Many to Many**

```
pgr_maxFlow(Edges SQL, sources, targets)
RETURNS BIGINT
```

**Example:**

From vertices  $\{(6, 8, 12)\}$  to vertices  $\{(1, 3, 11)\}$

```

SELECT * FROM pgr_maxFlow(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
pgr_maxflow
-----
      360
(1 row)

```

## Parameters

Column	Type	Default	Description
<b>Edges SQL</b>	TEXT		The edges SQL query as described in <b>Inner Query</b> .
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

## Inner query

### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

## Where:

### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

## Return Columns

Type	Description
BIGINT	Maximum flow possible from the source(s) to the target(s)

## See Also

- **Flow - Family of functions**
- [https://www.boost.org/libs/graph/doc/push\\_relabel\\_max\\_flow.html](https://www.boost.org/libs/graph/doc/push_relabel_max_flow.html)
- [https://en.wikipedia.org/wiki/Push%20%93relabel\\_maximum\\_flow\\_algorithm](https://en.wikipedia.org/wiki/Push%20%93relabel_maximum_flow_algorithm)

## Indices and tables

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

## pgr\_boykovKolmogorov

`pgr_boykovKolmogorov` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Boykov Kolmogorov algorithm.



## Availability:

- Version 3.0.0
  - Official** function
- Version 2.5.0
  - Renamed from `pgr_maxFlowBoykovKolmogorov`
  - Proposed** function
- Version 2.3.0
  - New **Experimental** function

## Support

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

## Description

### The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by `pgr_maxFlow` when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time: Polynomial

## Signatures

## Summary

```
pgr_boykovKolmogorov(Edges SQL, source, target)
pgr_boykovKolmogorov(Edges SQL, sources, target)
pgr_boykovKolmogorov(Edges SQL, source, targets)
pgr_boykovKolmogorov(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

## One to One

```
pgr_boykovKolmogorov(Edges SQL, source, target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

## Example:

From vertex `\(6\)` to vertex `\(11\)`

```
SELECT * FROM pgr_boykovKolmogorov(
'SELECT id,
 source,
 target,
 capacity,
 reverse_capacity
FROM edge_table'
,6,11
);
seq | edge | start_vid | end_vid | flow | residual_capacity
-----+-----+-----+-----+-----+-----
 1 | 10 | 5 | 10 | 100 | 30
 2 | 8 | 6 | 5 | 100 | 30
 3 | 11 | 6 | 11 | 130 | 0
 4 | 12 | 10 | 11 | 100 | 0
(4 rows)
```

## One to Many

```
pgr_boykovKolmogorov(Edges SQL, source, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

**Example:**

From vertex  $\{6\}$  to vertices  $\{1, 3, 11\}$

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , 6, ARRAY[1, 3, 11]
);
```

seq	edge	start_vid	end_vid	flow	residual_capacity
1	1	2	1	50	80
2	3	4	3	80	50
3	4	5	2	50	0
4	10	5	10	80	50
5	8	6	5	130	0
6	9	6	9	80	50
7	11	6	11	130	0
8	16	9	4	80	0
9	12	10	11	80	20

(9 rows)

Many to One

```
pgr_boykovKolmogorov(Edges SQL, sources, target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

**Example:**

From vertices  $\{6, 8, 12\}$  to vertex  $\{11\}$

```
SELECT * FROM pgr_boykovKolmogorov(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , ARRAY[6, 8, 12], 11
);
```

seq	edge	start_vid	end_vid	flow	residual_capacity
1	10	5	10	100	30
2	8	6	5	100	30
3	11	6	11	130	0
4	12	10	11	100	0

(4 rows)

Many to Many

```
pgr_boykovKolmogorov(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

**Example:**

From vertices  $\{6, 8, 12\}$  to vertices  $\{1, 3, 11\}$

```

SELECT * FROM pgr_boykovKolmogorov(
'SELECT id,
  source,
  target,
  capacity,
  reverse_capacity
FROM edge_table'
, ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
seq | edge | start_vid | end_vid | flow | residual_capacity
-----+-----+-----+-----+-----+-----
 1 | 1 | 2 | 1 | 50 | 80
 2 | 3 | 4 | 3 | 80 | 50
 3 | 4 | 5 | 2 | 50 | 0
 4 | 10 | 5 | 10 | 100 | 30
 5 | 8 | 6 | 5 | 130 | 0
 6 | 9 | 6 | 9 | 80 | 50
 7 | 11 | 6 | 11 | 130 | 0
 8 | 7 | 8 | 5 | 20 | 30
 9 | 16 | 9 | 4 | 80 | 0
10 | 12 | 10 | 11 | 100 | 0
(10 rows)

```

#### Parameters

Column	Type	Default	Description
<b>Edges SQL</b>	TEXT		The edges SQL query as described in <b>Inner Query</b> .
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

#### Inner query

##### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

##### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### Result Columns

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

See Also

- **Flow - Family of functions**, [pgr\\_pushRelabel](#), [pgr\\_edmondsKarp](#)
- [https://www.boost.org/libs/graph/doc/boykov\\_kolmogorov\\_max\\_flow.html](https://www.boost.org/libs/graph/doc/boykov_kolmogorov_max_flow.html)

#### Indices and tables

- [Index](#)
- [Search Page](#)

pgr\_edmondsKarp — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Push Relabel Algorithm.



Boost Graph Inside

### Availability

- Version 3.0.0
  - Official** function
- Version 2.5.0
  - Renamed from pgr\_maxFlowEdmondsKarp
  - Proposed** function
- Version 2.3.0
  - New **Experimental** function

### Support

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

### Description

#### The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the target(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr\_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time:  $O(V * E^2)$

### Signatures

### Summary

```
pgr_edmondsKarp(Edges SQL, source, target)
pgr_edmondsKarp(Edges SQL, sources, target)
pgr_edmondsKarp(Edges SQL, source, targets)
pgr_edmondsKarp(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

### One to One

```
pgr_edmondsKarp(Edges SQL, source, target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

### Example:

From vertex  $\{(6)\}$  to vertex  $\{(11)\}$

```
SELECT * FROM pgr_edmondsKarp(
'SELECT id,
  source,
  target,
  capacity,
  reverse_capacity
FROM edge_table'
, 6, 11
);
```

seq	edge	start_vid	end_vid	flow	residual_capacity
1	10	5	10	100	30
2	8	6	5	100	30
3	11	6	11	130	0
4	12	10	11	100	0

(4 rows)

### One to Many

```
pgr_edmondsKarp(Edges SQL, source, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

#### Example:

From vertex  $\{6\}$  to vertices  $\{1, 3, 11\}$

```
SELECT * FROM pgr_edmondsKarp(
'SELECT id,
  source,
  target,
  capacity,
  reverse_capacity
FROM edge_table'
, 6, ARRAY[1, 3, 11]
);
```

seq	edge	start_vid	end_vid	flow	residual_capacity
1	1	2	1	50	80
2	3	4	3	80	50
3	4	5	2	50	0
4	10	5	10	80	50
5	8	6	5	130	0
6	9	6	9	80	50
7	11	6	11	130	0
8	16	9	4	80	0
9	12	10	11	80	20

(9 rows)

### Many to One

```
pgr_edmondsKarp(Edges SQL, sources, target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

#### Example:

From vertices  $\{6, 8, 12\}$  to vertex  $\{11\}$

```
SELECT * FROM pgr_edmondsKarp(
'SELECT id,
  source,
  target,
  capacity,
  reverse_capacity
FROM edge_table'
, ARRAY[6, 8, 12], 11
);
```

seq	edge	start_vid	end_vid	flow	residual_capacity
1	10	5	10	100	30
2	8	6	5	100	30
3	11	6	11	130	0
4	12	10	11	100	0

(4 rows)

### Many to Many

```
pgr_edmondsKarp(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

#### Example:

From vertices  $\{6, 8, 12\}$  to vertices  $\{1, 3, 11\}$

```
SELECT * FROM pgr_edmondsKarp(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
seq | edge | start_vid | end_vid | flow | residual_capacity
-----+-----+-----+-----+-----+-----
 1 |  1 |      2 |      1 |   50 |             80
 2 |  3 |      4 |      3 |   80 |             50
 3 |  4 |      5 |      2 |   50 |              0
 4 | 10 |      5 |     10 |  100 |             30
 5 |  8 |      6 |      5 |  130 |              0
 6 |  9 |      6 |      9 |   80 |             50
 7 | 11 |      6 |     11 |  130 |              0
 8 |  7 |      8 |      5 |   20 |             30
 9 | 16 |      9 |      4 |   80 |              0
10 | 12 |     10 |     11 |  100 |              0
(10 rows)
```

Parameters

Column	Type	Default	Description
<b>Edges SQL</b>	TEXT		The edges SQL query as described in <b>Inner Query</b> .
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

**Edges SQL:**

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

Result Columns

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

See Also

- **Flow - Family of functions**, [pgr\\_boykovKolmogorov](#), [pgr\\_pushRelabel](#)
- [https://www.boost.org/libs/graph/doc/edmonds\\_karp\\_max\\_flow.html](https://www.boost.org/libs/graph/doc/edmonds_karp_max_flow.html)
- [https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp\\_algorithm](https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm)

Indices and tables



- [Index](#)
- [Search Page](#)

## pgr\_pushRelabel

`pgr_pushRelabel` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Push Relabel Algorithm.



Boost Graph Inside

### Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - Renamed from `pgr_maxFlowPushRelabel`
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

### Support

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

### Description

#### The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by `pgr_maxFlow` when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time:  $O(V^3)$

### Signatures

### Summary

```
pgr_pushRelabel(Edges SQL, source, target)
pgr_pushRelabel(Edges SQL, sources, target)
pgr_pushRelabel(Edges SQL, source, targets)
pgr_pushRelabel(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

### One to One

```
pgr_pushRelabel(Edges SQL, source, target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

### Example:

From vertex `\(6\)` to vertex `\(11\)`

```

SELECT * FROM pgr_pushRelabel(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , 6, 11
);
seq | edge | start_vid | end_vid | flow | residual_capacity
-----+-----+-----+-----+-----+-----
 1 | 10 | 5 | 10 | 100 | 30
 2 | 8 | 6 | 5 | 100 | 30
 3 | 11 | 6 | 11 | 130 | 0
 4 | 12 | 10 | 11 | 100 | 0
(4 rows)

```

### One to Many

Calculates the flow on the graph edges that maximizes the flow from the *source* to all of the *targets*.

```

pgr_pushRelabel(Edges SQL, source, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

```

#### Example:

From vertex  $\{6\}$  to vertices  $\{11, 1, 13\}$

```

SELECT * FROM pgr_pushRelabel(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , 6, ARRAY[11, 1, 13]
);
seq | edge | start_vid | end_vid | flow | residual_capacity
-----+-----+-----+-----+-----+-----
 1 | 1 | 2 | 1 | 130 | 0
 2 | 2 | 3 | 2 | 80 | 20
 3 | 3 | 4 | 3 | 80 | 50
 4 | 4 | 5 | 2 | 50 | 0
 5 | 7 | 5 | 8 | 50 | 80
 6 | 10 | 5 | 10 | 80 | 50
 7 | 8 | 6 | 5 | 130 | 0
 8 | 9 | 6 | 9 | 80 | 50
 9 | 11 | 6 | 11 | 130 | 0
10 | 6 | 7 | 8 | 50 | 0
11 | 6 | 8 | 7 | 50 | 50
12 | 7 | 8 | 5 | 50 | 0
13 | 16 | 9 | 4 | 80 | 0
14 | 12 | 10 | 11 | 80 | 20
(14 rows)

```

### Many to One

```

pgr_pushRelabel(Edges SQL, sources, target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET

```

#### Example:

From vertices  $\{6, 8, 12\}$  to vertex  $\{11\}$

```

SELECT * FROM pgr_pushRelabel(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , ARRAY[6, 8, 12], 11
);
seq | edge | start_vid | end_vid | flow | residual_capacity
-----+-----+-----+-----+-----+-----
 1 | 10 | 5 | 10 | 100 | 30
 2 | 8 | 6 | 5 | 100 | 30
 3 | 11 | 6 | 11 | 130 | 0
 4 | 12 | 10 | 11 | 100 | 0
(4 rows)

```

### Many to Many

```
pgr_pushRelabel(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

### Example:

From vertices  $\{6, 8, 12\}$  to vertices  $\{1, 3, 11\}$

```
SELECT * FROM pgr_pushRelabel(
  'SELECT id,
    source,
    target,
    capacity,
    reverse_capacity
  FROM edge_table'
  , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
```

seq	edge	start_vid	end_vid	flow	residual_capacity
1	1	2	1	50	80
2	3	4	3	80	50
3	4	5	2	50	0
4	10	5	10	100	30
5	8	6	5	130	0
6	9	6	9	30	100
7	11	6	11	130	0
8	7	8	5	20	30
9	16	9	4	80	0
10	12	10	11	100	0
11	15	12	9	50	0

(11 rows)

### Parameters

Column	Type	Default	Description
<b>Edges SQL</b>	TEXT		The edges SQL query as described in <b>Inner Query</b> .
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

### Inner query

#### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

### Result Columns

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

See Also

- **Flow - Family of functions**, [pgr\\_boykovKolmogorov](#), [pgr\\_edmondsKarp](#)
- [https://www.boost.org/libs/graph/doc/push\\_relabel\\_max\\_flow.html](https://www.boost.org/libs/graph/doc/push_relabel_max_flow.html)
- [https://en.wikipedia.org/wiki/Push%E2%80%93relabel\\_maximum\\_flow\\_algorithm](https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm)

## Indices and tables

- [Index](#)
- [Search Page](#)

### pgr\_edgeDisjointPaths

`pgr_edgeDisjointPaths` — Calculates edge disjoint paths between two groups of vertices.



Boost Graph Inside

## Availability

- Version 3.0.0
  - **Official** function
- Version 2.5.0
  - **Proposed** function
- Version 2.3.0
  - New **Experimental** function

## Support

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

## Description

Calculates the edge disjoint paths between two groups of vertices. Utilizes underlying maximum flow algorithms to calculate the paths.

The main 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.
- One to many, many to one, many to many versions are also supported.
- Uses [pgr\\_boykovKolmogorov](#) to calculate the paths.

## Signatures

## Summary

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid)
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid [, directed])
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vids [, directed])
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vid [, directed])
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vids [, directed])

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

## Using defaults

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

## Example:

From vertex `\(3\)` to vertex `\(5\)` on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  3, 5
);
```

seq	path_id	path_seq	node	edge	cost	agg_cost
1	1	1	3	2	1	0
2	1	2	2	4	1	1
3	1	3	5	-1	0	2
4	2	1	3	5	1	0
5	2	2	6	8	1	1
6	2	3	5	-1	0	2

(6 rows)

### One to One

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vid, directed)
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

#### Example:

From vertex  $\{3\}$  to vertex  $\{5\}$  on an **undirected** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  3, 5,
  directed := false
);
```

seq	path_id	path_seq	node	edge	cost	agg_cost
1	1	1	3	2	1	0
2	1	2	2	4	1	1
3	1	3	5	-1	0	2
4	2	1	3	3	-1	0
5	2	2	4	16	1	-1
6	2	3	9	9	1	0
7	2	4	6	8	1	1
8	2	5	5	-1	0	2
9	3	1	3	5	1	0
10	3	2	6	11	1	1
11	3	3	11	12	-1	2
12	3	4	10	10	1	1
13	3	5	5	-1	0	2

(13 rows)

### One to Many

```
pgr_edgeDisjointPaths(Edges SQL, start_vid, end_vids, directed)
RETURNS SET OF (seq, path_id, path_seq, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

#### Example:

From vertex  $\{3\}$  to vertices  $\{4, 5, 10\}$  on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  3, ARRAY[4, 5, 10]
);
```

seq	path_id	path_seq	end_vid	node	edge	cost	agg_cost
1	1	1	4	3	5	1	0
2	1	2	4	6	9	1	1
3	1	3	4	9	16	1	2
4	1	4	4	4	-1	0	3
5	2	1	5	3	2	1	0
6	2	2	5	2	4	1	1
7	2	3	5	5	-1	0	2
8	3	1	5	3	5	1	0
9	3	2	5	6	8	1	1
10	3	3	5	5	-1	0	2
11	4	1	10	3	2	1	0
12	4	2	10	2	4	1	1
13	4	3	10	5	10	1	2
14	4	4	10	10	-1	0	3

(14 rows)

### Many to One

```
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vid, directed)
RETURNS SET OF (seq, path_id, path_seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

**Example:**

From vertices  $\{3, 6\}$  to vertex  $\{5\}$  on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[3, 6], 5
);
```

seq	path_id	path_seq	start_vid	node	edge	cost	agg_cost
1	1	1	0	3	2	1	0
2	1	2	0	2	4	1	1
3	1	3	0	5	-1	0	2
4	2	1	1	3	5	1	0
5	2	2	1	6	8	1	1
6	2	3	1	5	-1	0	2
7	3	1	2	6	8	1	0
8	3	2	2	5	-1	0	1
9	4	1	3	6	9	1	0
10	4	2	3	9	16	1	1
11	4	3	3	4	3	1	2
12	4	4	3	3	2	1	3
13	4	5	3	2	4	1	4
14	4	6	3	5	-1	0	5

(14 rows)

**Many to Many**

```
pgr_edgeDisjointPaths(Edges SQL, start_vids, end_vids, directed)
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

**Example:**

From vertices  $\{3, 6\}$  to vertices  $\{4, 5, 10\}$  on a **directed** graph

```
SELECT * FROM pgr_edgeDisjointPaths(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[3, 6], ARRAY[4, 5, 10]
);
```

seq	path_id	path_seq	start_vid	end_vid	node	edge	cost	agg_cost
1	1	1	0	4	3	5	1	0
2	1	2	0	4	6	9	1	1
3	1	3	0	4	9	16	1	2
4	1	4	0	4	4	-1	0	3
5	2	1	1	5	3	2	1	0
6	2	2	1	5	2	4	1	1
7	2	3	1	5	5	-1	0	2
8	3	1	2	5	3	5	1	0
9	3	2	2	5	6	8	1	1
10	3	3	2	5	5	-1	0	2
11	4	1	3	10	3	2	1	0
12	4	2	3	10	2	4	1	1
13	4	3	3	10	5	10	1	2
14	4	4	3	10	10	-1	0	3
15	5	1	4	4	6	9	1	0
16	5	2	4	4	9	16	1	1
17	5	3	4	4	4	-1	0	2
18	6	1	5	5	6	8	1	0
19	6	2	5	5	5	-1	0	1
20	7	1	6	5	6	9	1	0
21	7	2	6	5	9	16	1	1
22	7	3	6	5	4	3	1	2
23	7	4	6	5	3	2	1	3
24	7	5	6	5	2	4	1	4
25	7	6	6	5	5	-1	0	5
26	8	1	7	10	6	8	1	0
27	8	2	7	10	5	10	1	1
28	8	3	7	10	10	-1	0	2

(28 rows)

**Parameters**

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

**Inner query**

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Return Columns

Returns set of (seq, path\_id, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_id</b>	INT	Path identifier. Has value <b>1</b> for the first of a path. Used when there are multiple paths for the same <code>start_vid</code> to <code>end_vid</code> combination.
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li><b>One to Many</b></li> <li><b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

See Also

- Flow - Family of functions**

**Indices and tables**

- Index**
- Search Page**

`pgr_maxCardinalityMatch`

`pgr_maxCardinalityMatch` — Calculates a maximum cardinality matching in a graph.



Boost Graph Inside

**Availability**

- Version 3.0.0
  - Official** function
- Version 2.5.0
  - Renamed from `pgr_maximumCardinalityMatching`
  - Proposed** function

- Version 2.3.0
  - New **Experimental** function

## Support

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

## Description

### The main characteristics are:

- A matching or independent edge set in a graph is a set of edges without common vertices.
- A maximum matching is a matching that contains the largest possible number of edges.
  - There may be many maximum matchings.
  - Calculates **one** possible maximum cardinality matching in a graph.
- The graph can be **directed** or **undirected**.
- Running time:  $\mathcal{O}(E \cdot V \cdot \alpha(E, V))$ 
  - $\alpha(E, V)$  is the inverse of the **Ackermann function**.

## Signatures

```
pgr_maxCardinalityMatch(Edges SQL [, directed])

RETURNS SET OF (seq, edge_id, source, target)
OR EMPTY SET
```

### Example:

For an **undirected** graph

```
SELECT * FROM pgr_maxCardinalityMatch(
  'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
  directed := false
);
seq | edge | source | target
-----+-----+-----+-----
 1 |  1 |    1 |    2
 2 |  3 |    3 |    4
 3 |  9 |    6 |    9
 4 |  6 |    7 |    8
 5 | 14 |   10 |   13
 6 | 13 |   11 |   12
 7 | 17 |   14 |   15
 8 | 18 |   16 |   17
(8 rows)
```

## Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		SQL query as described above.
<b>directed</b>	BOOLEAN	true	Determines the type of the graph. - When <b>true</b> Graph is considered <i>Directed</i> - When <b>false</b> the graph is considered as <i>Undirected</i> .

## Inner query

### Edges SQL:

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

Column	Type	Description
<b>id</b>	ANY-INTEGER	Identifier of the edge.
<b>source</b>	ANY-INTEGER	Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER	Identifier of the second end point vertex of the edge.
<b>going</b>	ANY-NUMERIC	A positive value represents the existence of the edge <code>source, target</code> ).
<b>coming</b>	ANY-NUMERIC	A positive value represents the existence of the edge <code>target, source</code> ).

Where:

### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERIC:

SMALLINT, INTEGER, BIGINT, REAL FLOAT



## Result Columns

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>edge</b>	BIGINT	Identifier of the edge in the original query.
<b>source</b>	BIGINT	Identifier of the first end point of the edge.
<b>target</b>	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://www.boost.org/libs/graph/doc/maximum_matching.html)
- [https://en.wikipedia.org/wiki/Matching\\_%28graph\\_theory%29](https://en.wikipedia.org/wiki/Matching_%28graph_theory%29)
- [https://en.wikipedia.org/wiki/Ackermann\\_function](https://en.wikipedia.org/wiki/Ackermann_function)

## Indices and tables

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

## pgr\_maxFlowMinCost - Experimental

`pgr_maxFlowMinCost` — Calculates the flow on the graph edges that maximizes the flow and minimizes the cost from the sources to the targets.



## Boost Graph Inside



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGERS 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Availability

- Version 3.0.0
- New **experimental** function

## Support

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

## Description

## The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr\_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- **TODO** check which statement is true:
  - The cost value of all input edges must be nonnegative.
  - Process is done when the cost value of all input edges is nonnegative.
  - Process is done on edges with nonnegative cost.
- Running time:  $O(U * (E + V * \log V))$ 
  - where  $U$  is the value of the max flow.
  - $U$  is upper bound on number of iterations. In many real world cases number of iterations is much smaller than  $U$ .

## Signatures

### Summary

```
pgr_maxFlowMinCost(Edges SQL, source, target)
pgr_maxFlowMinCost(Edges SQL, sources, target)
pgr_maxFlowMinCost(Edges SQL, source, targets)
pgr_maxFlowMinCost(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

### One to One

```
pgr_maxFlowMinCost(Edges SQL, source, target)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

### Example:

From vertex  $\{2\}$  to vertex  $\{3\}$

```
SELECT * FROM pgr_MaxFlowMinCost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
2, 3
);
```

seq	edge	source	target	flow	residual_capacity	cost	agg_cost
1	4	2	5	80	20	80	80
2	3	4	3	80	50	80	160
3	8	5	6	80	20	80	240
4	9	6	9	80	50	80	320
5	16	9	4	80	0	80	400

(5 rows)

### One to Many

```
pgr_maxFlowMinCost(Edges SQL, source, targets)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

### Example:

From vertex  $\{13\}$  to vertices  $\{7, 1, 4\}$

```
SELECT * FROM pgr_MaxFlowMinCost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
13, ARRAY[7, 1, 4]
);
seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 1 | 50 | 80 | 50 | 50
2 | 4 | 5 | 2 | 50 | 0 | 50 | 100
3 | 16 | 9 | 4 | 50 | 30 | 50 | 150
4 | 10 | 10 | 5 | 50 | 0 | 50 | 200
5 | 12 | 10 | 11 | 50 | 50 | 50 | 250
6 | 13 | 11 | 12 | 50 | 50 | 50 | 300
7 | 15 | 12 | 9 | 50 | 0 | 50 | 350
8 | 14 | 13 | 10 | 100 | 30 | 100 | 450
(8 rows)
```

### Many to One

```
pgr_maxFlowMinCost(Edges SQL, sources, target)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

#### Example:

From vertices  $\{\{1, 7, 14\}\}$  to vertex  $\{12\}$

```
SELECT * FROM pgr_MaxFlowMinCost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
ARRAY[1, 7, 14], 12
);
seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 2 | 80 | 0 | 80 | 80
2 | 4 | 2 | 5 | 80 | 20 | 80 | 160
3 | 8 | 5 | 6 | 100 | 0 | 100 | 260
4 | 10 | 5 | 10 | 30 | 100 | 30 | 290
5 | 9 | 6 | 9 | 50 | 80 | 50 | 340
6 | 11 | 6 | 11 | 50 | 80 | 50 | 390
7 | 6 | 7 | 8 | 50 | 0 | 50 | 440
8 | 7 | 8 | 5 | 50 | 0 | 50 | 490
9 | 15 | 9 | 12 | 50 | 30 | 50 | 540
10 | 12 | 10 | 11 | 30 | 70 | 30 | 570
11 | 13 | 11 | 12 | 80 | 20 | 80 | 650
(11 rows)
```

### Many to Many

```
pgr_maxFlowMinCost(Edges SQL, sources, targets)
RETURNS SET OF (seq, edge, source, target, flow, residual_capacity, cost, agg_cost)
OR EMPTY SET
```

#### Example:

From vertices  $\{\{7, 13\}\}$  to vertices  $\{\{3, 9\}\}$

```
SELECT * FROM pgr_MaxFlowMinCost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
ARRAY[7, 13], ARRAY[3, 9]
);
seq | edge | source | target | flow | residual_capacity | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----+-----
1 | 8 | 5 | 6 | 100 | 0 | 100 | 100
2 | 9 | 6 | 9 | 100 | 30 | 100 | 200
3 | 6 | 7 | 8 | 50 | 0 | 50 | 250
4 | 7 | 8 | 5 | 50 | 0 | 50 | 300
5 | 10 | 10 | 5 | 50 | 0 | 50 | 350
6 | 12 | 10 | 11 | 50 | 50 | 50 | 400
7 | 13 | 11 | 12 | 50 | 50 | 50 | 450
8 | 15 | 12 | 9 | 50 | 0 | 50 | 500
9 | 14 | 13 | 10 | 100 | 30 | 100 | 600
(9 rows)
```

### Parameters

Column	Type	Default	Description
--------	------	---------	-------------

Column	Type	Default	Description
<b>Edges SQL</b>	TEXT		The edges SQL query as described in <b>Inner Query</b> .
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

#### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Capacity of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Capacity of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>cost</b>	ANY-NUMERICALS		Weight of the edge ( <i>source</i> , <i>target</i> ) if it exists.
<b>reverse_cost</b>	ANY-NUMERICALS	0	Weight of the edge ( <i>target</i> , <i>source</i> ) if it exists.

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICALS:

smallint, int, bigint, real, float

Result Columns

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>edge</b>	BIGINT	Identifier of the edge in the original query(edges_sql).
<b>source</b>	BIGINT	Identifier of the first end point vertex of the edge.
<b>target</b>	BIGINT	Identifier of the second end point vertex of the edge.
<b>flow</b>	BIGINT	Flow through the edge in the direction (source, target).
<b>residual_capacity</b>	BIGINT	Residual capacity of the edge in the direction (source, target).
<b>cost</b>	FLOAT	The cost of sending this flow through the edge in the direction (source, target).
<b>agg_cost</b>	FLOAT	The aggregate cost.

See Also

- **Flow - Family of functions**
- [https://www.boost.org/libs/graph/doc/successive\\_shortest\\_path\\_nonnegative\\_weights.html](https://www.boost.org/libs/graph/doc/successive_shortest_path_nonnegative_weights.html)

#### Indices and tables

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

pgr\_maxFlowMinCost\_Cost - Experimental

pgr\_maxFlowMinCost\_Cost — Calculates the minimum cost maximum flow in a directed graph from the source(s) to the target(s).



Boost Graph Inside

**Warning**

## Possible server crash

- These functions might create a server crash



## Warning

### Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Availability

- Version 3.0.0
  - New **experimental** function

## Support

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

## Description

### The main characteristics are:

- The graph is **directed**.
- **The cost value of all input edges must be nonnegative.**
- When the maximum flow is 0 then there is no flow and 0 is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Uses the **pgr\_maxFlowMinCost** algorithm.
- Running time:  $\mathcal{O}(U * (E + V * \log V))$ , where  $U$  is the value of the max flow.  $U$  is upper bound on number of iteration. In many real world cases number of iterations is much smaller than  $U$ .

## Signatures

## Summary

```
pgr_maxFlowMinCost_Cost(Edges SQL, source, target)
pgr_maxFlowMinCost_Cost(Edges SQL, sources, target)
pgr_maxFlowMinCost_Cost(Edges SQL, source, targets)
pgr_maxFlowMinCost_Cost(Edges SQL, sources, targets)
RETURNS FLOAT
```

## One to One

```
pgr_maxFlowMinCost_Cost(Edges SQL, source, target)
RETURNS FLOAT
```

## Example:

From vertex  $\{2\}$  to vertex  $\{3\}$

```

SELECT * FROM pgr_MaxFlowMinCost_Cost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
2, 3
);
pgr_maxflowmincost_cost
-----
400
(1 row)

```

### One to Many

```

pgr_maxFlowMinCost_Cost(Edges SQL, source, targets)
RETURNS FLOAT

```

#### Example:

From vertex  $\{13\}$  to vertices  $\{7, 1, 4\}$

```

SELECT * FROM pgr_MaxFlowMinCost_Cost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
13, ARRAY[7, 1, 4]
);
pgr_maxflowmincost_cost
-----
450
(1 row)

```

### Many to One

```

pgr_maxFlowMinCost_Cost(Edges SQL, sources, target)
RETURNS FLOAT

```

#### Example:

From vertices  $\{1, 7, 14\}$  to vertex  $\{12\}$

```

SELECT * FROM pgr_MaxFlowMinCost_Cost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
ARRAY[1, 7, 14], 12
);
pgr_maxflowmincost_cost
-----
650
(1 row)

```

### Many to Many

```

pgr_maxFlowMinCost_Cost(Edges SQL, sources, targets)
RETURNS FLOAT

```

#### Example:

From vertices  $\{7, 13\}$  to vertices  $\{3, 9\}$

```

SELECT * FROM pgr_MaxFlowMinCost_Cost(
'SELECT id,
source, target,
capacity, reverse_capacity,
cost, reverse_cost FROM edge_table',
ARRAY[7, 13], ARRAY[3, 9]
);
pgr_maxflowmincost_cost
-----
600
(1 row)

```

### Parameters

Column	Type	Default	Description
Edges SQL	TEXT		The edges SQL query as described in <b>Inner Query</b> .

Column	Type	Default	Description
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

#### Edges SQL:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Capacity of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Capacity of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) if it exists.
<b>reverse_cost</b>	ANY-NUMERICAL	0	Weight of the edge ( <i>target</i> , <i>source</i> ) if it exists.

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

smallint, int, bigint, real, float

Result Columns

Type	Description
FLOAT	Minimum Cost Maximum Flow possible from the source(s) to the target(s)

See Also

- Flow - Family of functions
- [https://www.boost.org/libs/graph/doc/successive\\_shortest\\_path\\_nonnegative\\_weights.html](https://www.boost.org/libs/graph/doc/successive_shortest_path_nonnegative_weights.html)

#### Indices and tables

- Index
- Search Page

#### Previous versions of this page

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

#### Flow Functions General Information

#### The main characteristics are:

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by **pgr\_maxFlow** when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets

**pgr\_maxFlow** is the maximum Flow and that maximum is guaranteed to be the same on the functions **pgr\_pushRelabel**, **pgr\_edmondsKarp**, **pgr\_boykovKolmogorov**, but the actual flow through each edge may vary.

Parameters

Column	Type	Default	Description
<b>Edges SQL</b>	TEXT		The edges SQL query as described in <b>Inner Query</b> .
<b>source</b>	BIGINT		Identifier of the starting vertex of the flow.
<b>sources</b>	ARRAY[BIGINT]		Array of identifiers of the starting vertices of the flow.
<b>target</b>	BIGINT		Identifier of the ending vertex of the flow.
<b>targets</b>	ARRAY[BIGINT]		Array of identifiers of the ending vertices of the flow.

Inner query

For **pgr\_pushRelabel**, **pgr\_edmondsKarp**, **pgr\_boykovKolmogorov** :

**Edges SQL:**

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

For **pgr\_maxFlowMinCost - Experimental** and **pgr\_maxFlowMinCost\_Cost - Experimental**:

**Edges SQL:**

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>capacity</b>	ANY-INTEGERS		Capacity of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_capacity</b>	ANY-INTEGERS	-1	Capacity of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) if it exists.
<b>reverse_cost</b>	ANY-NUMERICAL	0	Weight of the edge ( <i>target</i> , <i>source</i> ) if it exists.

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

smallint, int, bigint, real, float

Result Columns

For **pgr\_pushRelabel**, **pgr\_edmondsKarp**, **pgr\_boykovKolmogorov** :

Column	Type	Description
--------	------	-------------



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

### For **pgr\_maxFlowMinCost** - Experimental

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>edge</b>	BIGINT	Identifier of the edge in the original query(edges_sql).
<b>source</b>	BIGINT	Identifier of the first end point vertex of the edge.
<b>target</b>	BIGINT	Identifier of the second end point vertex of the edge.
<b>flow</b>	BIGINT	Flow through the edge in the direction (source, target).
<b>residual_capacity</b>	BIGINT	Residual capacity of the edge in the direction (source, target).
<b>cost</b>	FLOAT	The cost of sending this flow through the edge in the direction (source, target).
<b>agg_cost</b>	FLOAT	The aggregate cost.

#### Advanced 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  $(V)$ 
  - $(source\_vertex \cup sink\_vertex \cup source\_i \cup target\_i)$
- the set of edges  $(E)$ 
  - $(E = \begin{cases} \text{ } \\ \{(source\_i, target\_i, capacity\_i) \text{ when } capacity > 0 \} \ \& \ \text{if } \\ reverse\_capacity = \varnothing \ \& \ \text{if } \\ \{(source\_i, target\_i, capacity\_i) \text{ when } capacity > 0 \} \ \cup \ \{(target\_i, source\_i, reverse\_capacity\_i) \text{ when } reverse\_capacity\_i > 0 \} \ \& \ \text{if } \\ reverse\_capacity \neq \varnothing \ \end{cases})$

#### Maximum flow problem

Given:

- $(G(V,E))$
- $(source\_vertex \in V)$  the source vertex
- $(sink\_vertex \in V)$  the sink vertex

Then:

- $(pgr\_maxFlow(edges\_sql, source, sink) = \Phi)$
- $(\Phi = \{(id\_i, edge\_id\_i, source\_i, target\_i, flow\_i, residual\_capacity\_i)\})$

Where:

$(\Phi)$  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](https://en.wikipedia.org/wiki/Maximum_flow_problem)

## Indices and tables

- [Index](#)
- [Search Page](#)

## Kruskal - Family of functions

- [pgr\\_kruskal](#)
- [pgr\\_kruskalBFS](#)
- [pgr\\_kruskalDD](#)
- [pgr\\_kruskalDFS](#)



Boost Graph Inside

### pgr\_kruskal

`pgr_kruskal` — Returns the minimum spanning tree of graph using Kruskal algorithm.



Boost Graph Inside

## Availability

- Version 3.0.0
  - New **Official** function

## Support

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

## Description

This algorithm finds the minimum spanning forest in a possibly disconnected graph using Kruskal's algorithm.

## The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time:  $O(E * \log E)$
- EMPTY SET is returned when there are no edges in the graph.

## Signatures

## Summary

```
pgr_kruskal(edges_sql)
RETURNS SET OF (seq, edge, cost)
OR EMPTY SET
```

## Example:

Minimum Spanning Forest

```

SELECT * FROM pgr_kruskal(
  'SELECT id, source, target, cost, reverse_cost
   FROM edge_table ORDER BY id'
) ORDER BY edge;
edge | cost
-----+-----
 1 | 1
 2 | 1
 3 | 1
 6 | 1
 7 | 1
10 | 1
11 | 1
12 | 1
13 | 1
14 | 1
15 | 1
16 | 1
17 | 1
18 | 1
(14 rows)

```

#### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

#### Where:

##### **ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

##### **ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

Returns SET OF (*edge*, *cost*)

Column	Type	Description
<b>edge</b>	BIGINT	Identifier of the edge.
<b>cost</b>	FLOAT	Cost to traverse the edge.

#### See Also

- [Spanning Tree - Category](#)
- [Kruskal - Family of functions](#)
- The queries use the **Sample Data** network.
- [Boost: Kruskal's algorithm documentation](#)
- [Wikipedia: Kruskal's algorithm](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### pgr\_kruskalBFS

pgr\_kruskalBFS — Prim algorithm for Minimum Spanning Tree with Depth First Search ordering.



## Boost Graph Inside

### Availability

- Version 3.0.0
  - New **Official** function

### Support

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

### Description

Visits and extracts the nodes information in Breath First Search ordering of the Minimum Spanning Tree created using Prim's algorithm.

### The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time:  $\mathcal{O}(E \cdot \log E)$
- Returned tree nodes from a root vertex are on Breath First Search order
- Breath First Search Running time:  $\mathcal{O}(E + V)$

### Signatures

```
pgr_kruskalBFS(Edges SQL, Root vid [, max_depth])
pgr_kruskalBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

### Single vertex

```
pgr_kruskalBFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

### Example:

The Minimum Spanning Tree having as root vertex  $\{2\}$

```
SELECT * FROM pgr_kruskalBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 3 | 2 | 9 | 16 | 1 | 3
 6 | 4 | 2 | 12 | 15 | 1 | 4
 7 | 5 | 2 | 11 | 13 | 1 | 5
 8 | 6 | 2 | 6 | 11 | 1 | 6
 9 | 6 | 2 | 10 | 12 | 1 | 6
10 | 7 | 2 | 5 | 10 | 1 | 7
11 | 7 | 2 | 13 | 14 | 1 | 7
12 | 8 | 2 | 8 | 7 | 1 | 8
13 | 9 | 2 | 7 | 6 | 1 | 9
(13 rows)
```

### Multiple vertices

```
pgr_kruskalBFS(Edges SQL, Root vids [, max_depth])
RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

### Example:

The Minimum Spanning Tree starting on vertices  $\{13, 2\}$  with  $(depth \leq 3)$

```
SELECT * FROM pgr_kruskalBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  ARRAY[13,2], max_depth := 3
);
```

seq	depth	start_vid	node	edge	cost	agg_cost
1	0	2	2	-1	0	0
2	1	2	1	1	1	1
3	1	2	3	2	1	1
4	2	2	4	3	1	2
5	3	2	9	16	1	3
6	0	13	13	-1	0	0
7	1	13	10	14	1	1
8	2	13	5	10	1	2
9	2	13	11	12	1	2
10	3	13	8	7	1	3
11	3	13	6	11	1	3
12	3	13	12	13	1	3

(12 rows)

### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"> <li>Used on <b>Single vertex</b></li> <li>When value is <math>\{0\}</math> then gets the spanning forest starting in aleatory nodes for each tree in the forest.</li> </ul>
<b>Root vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. <ul style="list-style-type: none"> <li>Used on <b>Multiple vertices</b></li> <li><math>\{0\}</math> values are ignored</li> <li>For optimization purposes, any duplicated value is ignored.</li> </ul>

### Optional Parameters

Parameter	Type	Default	Description
<b>max_depth</b>	BIGINT	$\{9223372036854775807\}$	Upper limit for depth of node in the tree <ul style="list-style-type: none"> <li>When value is <u>Negative</u> then <b>throws error</b></li> </ul>

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICALS:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
--------	------	-------------

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from $\backslash(1\backslash)$ .
<b>depth</b>	BIGINT	Depth of the <code>node</code> . <ul style="list-style-type: none"> <li><math>\backslash(0\backslash)</math> when <code>node = start_vid</code>.</li> </ul>
<b>start_vid</b>	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"> <li>In <b>Multiple Vertices</b> results are in ascending order.</li> </ul>
<b>node</b>	BIGINT	Identifier of <code>node</code> reached using <code>edge</code> .
<b>edge</b>	BIGINT	Identifier of the <code>edge</code> used to arrive to <code>node</code> . <ul style="list-style-type: none"> <li><math>\backslash(-1\backslash)</math> when <code>node = start_vid</code>.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse <code>edge</code> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

See Also

- [Spanning Tree - Category](#)
- [Kruskal - Family of functions](#)
- The queries use the [Sample Data](#) network.
- [Boost: Kruskal's algorithm documentation](#)
- [Wikipedia: Kruskal's algorithm](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

`pgr_kruskalDD`

`pgr_kruskalDD` — Catchment nodes using Kruskal's algorithm.



Boost Graph Inside

## Availability

- Version 3.0.0
  - New **Official** function

## Support

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

## Description

Using Kruskal's algorithm, extracts the nodes that have aggregate costs less than or equal to the value `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.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time:  $\backslash(O(E * \log E)\backslash)$
- Returned tree nodes from a root vertex are on Depth First Search order.
- Depth First Search running time:  $\backslash(O(E + V)\backslash)$

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  $\backslash(2\backslash)$  with  $\backslash(agg\_cost \leq 3.5\backslash)$

```
SELECT * FROM pgr_kruskalDD(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2, 3.5
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 3 | 2 | 9 | 16 | 1 | 3
(5 rows)
```

### Multiple vertices

```
pgr_kruskalDD(edges_sql, root_vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

#### Example:

The Minimum Spanning Tree starting on vertices  $\backslash(\{13, 2\}\backslash)$  with  $\backslash(agg\_cost \leq 3.5\backslash)$ ;

```
SELECT * FROM pgr_kruskalDD(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  ARRAY[13,2],
  3.5
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 3 | 2 | 9 | 16 | 1 | 3
 6 | 0 | 13 | 13 | -1 | 0 | 0
 7 | 1 | 13 | 10 | 14 | 1 | 1
 8 | 2 | 13 | 5 | 10 | 1 | 2
 9 | 3 | 13 | 8 | 7 | 1 | 3
10 | 2 | 13 | 11 | 12 | 1 | 2
11 | 3 | 13 | 6 | 11 | 1 | 3
12 | 3 | 13 | 12 | 13 | 1 | 3
(12 rows)
```

### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"> <li>Used on <b>Single vertex</b></li> <li>When <math>\backslash(0\backslash)</math> gets the spanning forest starting in aleatory nodes for each tree.</li> </ul>
<b>Root vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. <ul style="list-style-type: none"> <li>Used on <b>Multiple vertices</b></li> <li><math>\backslash(0\backslash)</math> values are ignored</li> <li>For optimization purposes, any duplicated value is ignored.</li> </ul>
<b>Distance</b>	ANY-NUMERIC	Upper limit for the inclusion of the node in the result. <ul style="list-style-type: none"> <li>When the value is Negative <b>throws error</b></li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERIC:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT, NUMERIC

Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from \{1\}.
<b>depth</b>	BIGINT	Depth of the <code>node</code> . <ul style="list-style-type: none"> <li>\{0\} when <code>node = start_vid</code>.</li> </ul>
<b>start_vid</b>	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"> <li>In <b>Multiple Vertices</b> results are in ascending order.</li> </ul>
<b>node</b>	BIGINT	Identifier of <code>node</code> reached using <code>edge</code> .
<b>edge</b>	BIGINT	Identifier of the <code>edge</code> used to arrive to <code>node</code> . <ul style="list-style-type: none"> <li>\{-1\} when <code>node = start_vid</code>.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse <code>edge</code> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

See Also

- [Spanning Tree - Category](#)
- [Kruskal - Family of functions](#)
- The queries use the [Sample Data](#) network.
- [Boost: Kruskal's algorithm documentation](#)
- [Wikipedia: Kruskal's algorithm](#)

**Indices and tables**

- [Index](#)
- [Search Page](#)

`pgr_kruskalDFS`

`pgr_kruskalDFS` — Kruskal algorithm for Minimum Spanning Tree with Depth First Search ordering.



Boost Graph Inside

**Availability**



- Version 3.0.0
  - New **Official** function

## Support

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

## Description

Visits and extracts the nodes information in Depth First Search ordering of the Minimum Spanning Tree created using Kruskal's algorithm.

## The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time:  $\mathcal{O}(E * \log E)$
- Returned tree nodes from a root vertex are on Depth First Search order
- Depth First Search Running time:  $\mathcal{O}(E + V)$

## Signatures

```
pgr_kruskalDFS(Edges SQL, Root vid [, max_depth])
pgr_kruskalDFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

## Single vertex

```
pgr_kruskalDFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

## Example:

The Minimum Spanning Tree starting on vertex  $\{2\}$

```
SELECT * FROM pgr_kruskalDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 3 | 2 | 9 | 16 | 1 | 3
 6 | 4 | 2 | 12 | 15 | 1 | 4
 7 | 5 | 2 | 11 | 13 | 1 | 5
 8 | 6 | 2 | 6 | 11 | 1 | 6
 9 | 6 | 2 | 10 | 12 | 1 | 6
10 | 7 | 2 | 5 | 10 | 1 | 7
11 | 8 | 2 | 8 | 7 | 1 | 8
12 | 9 | 2 | 7 | 6 | 1 | 9
13 | 7 | 2 | 13 | 14 | 1 | 7
(13 rows)
```

## Multiple vertices

```
pgr_kruskalDFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

## Example:

The Minimum Spanning Tree starting on vertices  $\{\{13, 2\}\}$  with  $\{depth \leq 3\}$

```

SELECT * FROM pgr_kruskalDFS(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
ARRAY[13,2], max_depth := 3
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 0 | 2 | 2 | -1 | 0 | 0
2 | 1 | 2 | 1 | 1 | 1 | 1
3 | 1 | 2 | 3 | 2 | 1 | 1
4 | 2 | 2 | 4 | 3 | 1 | 2
5 | 3 | 2 | 9 | 16 | 1 | 3
6 | 0 | 13 | 13 | -1 | 0 | 0
7 | 1 | 13 | 10 | 14 | 1 | 1
8 | 2 | 13 | 5 | 10 | 1 | 2
9 | 3 | 13 | 8 | 7 | 1 | 3
10 | 2 | 13 | 11 | 12 | 1 | 2
11 | 3 | 13 | 6 | 11 | 1 | 3
12 | 3 | 13 | 12 | 13 | 1 | 3
(12 rows)

```

#### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"> <li>Used on <b>Single vertex</b></li> <li>When value is \(\0\) then gets the spanning forest starting in aleatory nodes for each tree in the forest.</li> </ul>
<b>Root vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. <ul style="list-style-type: none"> <li>Used on <b>Multiple vertices</b></li> <li>\(\0\) values are ignored</li> <li>For optimization purposes, any duplicated value is ignored.</li> </ul>

#### Optional Parameters

Parameter	Type	Default	Description
<b>max_depth</b>	BIGINT	\(9223372036854775807\)	Upper limit for depth of node in the tree <ul style="list-style-type: none"> <li>When value is <b>Negative</b> then <b>throws error</b></li> </ul>

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from \(\1\).
<b>depth</b>	BIGINT	Depth of the <i>node</i> . <ul style="list-style-type: none"> <li>\(\0\) when <i>node</i> = <i>start_vid</i>.</li> </ul>

Column	Type	Description
<b>start_vid</b>	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"> <li>In <b>Multiple Vertices</b> results are in ascending order.</li> </ul>
<b>node</b>	BIGINT	Identifier of <code>node</code> reached using <code>edge</code> .
<b>edge</b>	BIGINT	Identifier of the <code>edge</code> used to arrive to <code>node</code> . <ul style="list-style-type: none"> <li><math>\setminus(-1)</math> when <code>node</code> = <code>start_vid</code>.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse <code>edge</code> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

See Also

- Spanning Tree - Category
- Kruskal - Family of functions
- The queries use the **Sample Data** network.
- Boost: Kruskal's algorithm documentation**
- Wikipedia: Kruskal's algorithm**

## Indices and tables

- Index**
- Search Page**

## Previous versions of this page

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

## Description

Kruskal's algorithm is a greedy minimum spanning tree algorithm that in each cycle finds and adds the edge of the least possible weight that connects any two trees in the forest.

## The main Characteristics are:

- It's implementation is only on **undirected** graph.
- Process is done only on edges with positive costs.
- The total weight of all the edges in the tree or forest is minimized.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Kruskal's running time:  $\setminus(O(E * \log E))$

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

See Also

- Spanning Tree - Category**

- [Boost: Kruskal's algorithm documentation](#)
- [Wikipedia: Kruskal's algorithm](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

## Prim - Family of functions

- [pgr\\_prim](#)
- [pgr\\_primBFS](#)
- [pgr\\_primDD](#)
- [pgr\\_primDFS](#)



Boost Graph Inside

### pgr\_prim

`pgr_prim` — Minimum spanning forest of graph using Prim algorithm.



Boost Graph Inside

## Availability

- Version 3.0.0
  - New **Official** function

## Support

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

## Description

This algorithm finds the minimum spanning forest in a possibly disconnected graph using Prim's algorithm.

## The main characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time:  $\mathcal{O}(E \cdot \log V)$
- EMPTY SET is returned when there are no edges in the graph.

## Signatures

## Summary

```
pgr_prim(edges_sq)
```

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

## Example:

Minimum Spanning Forest of a subgraph

```

SELECT edge, cost FROM pgr_prim(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table WHERE id < 14'
) ORDER BY edge;
edge | cost
-----+-----
 1 | 1
 2 | 1
 3 | 1
 4 | 1
 5 | 1
 6 | 1
 7 | 1
 9 | 1
10 | 1
11 | 1
13 | 1
(11 rows)

```

#### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

Returns SET OF (edge, cost)

Column	Type	Description
<b>edge</b>	BIGINT	Identifier of the edge.
<b>cost</b>	FLOAT	Cost to traverse the edge.

See Also

- [Spanning Tree - Category](#)
- [Prim - Family of functions](#)
- The queries use the [Sample Data](#) network.
- [Boost: Prim's algorithm documentation](#)
- [Wikipedia: Prim's algorithm](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### pgr\_primBFS

`pgr_primBFS` — Prim's algorithm for Minimum Spanning Tree with Depth First Search ordering.



## Availability

- Version 3.0.0
  - New **Official** function

## Support

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

## Description

Visits and extracts the nodes information in Breath First Search ordering of the Minimum Spanning Tree created with 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:  $\mathcal{O}(E \cdot \log V)$
- Returned tree nodes from a root vertex are on Breath First Search order
- Breath First Search Running time:  $\mathcal{O}(E + V)$

## Signatures

```
pgr_primBFS(Edges SQL, Root vid [, max_depth])
pgr_primBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

## Single vertex

```
pgr_primBFS(Edges SQL, Root vid [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

## Example:

The Minimum Spanning Tree having as root vertex  $\{2\}$

```
SELECT * FROM pgr_primBFS(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 1 | 2 | 5 | 4 | 1 | 1
 5 | 2 | 2 | 4 | 3 | 1 | 2
 6 | 2 | 2 | 6 | 5 | 1 | 2
 7 | 2 | 2 | 8 | 7 | 1 | 2
 8 | 2 | 2 | 10 | 10 | 1 | 2
 9 | 3 | 2 | 9 | 9 | 1 | 3
10 | 3 | 2 | 11 | 11 | 1 | 3
11 | 3 | 2 | 7 | 6 | 1 | 3
12 | 3 | 2 | 13 | 14 | 1 | 3
13 | 4 | 2 | 12 | 13 | 1 | 4
(13 rows)
```

## Multiple vertices

```
pgr_primBFS(Edges SQL, Root vids [, max_depth])

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

## Example:

The Minimum Spanning Tree starting on vertices  $\{13, 2\}$  with  $(depth \leq 3)$

```

SELECT * FROM pgr_primBFS(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
ARRAY[13,2], max_depth := 3
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 1 | 2 | 5 | 4 | 1 | 1
 5 | 2 | 2 | 4 | 3 | 1 | 2
 6 | 2 | 2 | 6 | 5 | 1 | 2
 7 | 2 | 2 | 8 | 7 | 1 | 2
 8 | 2 | 2 | 10 | 10 | 1 | 2
 9 | 3 | 2 | 9 | 9 | 1 | 3
10 | 3 | 2 | 11 | 11 | 1 | 3
11 | 3 | 2 | 7 | 6 | 1 | 3
12 | 3 | 2 | 13 | 14 | 1 | 3
13 | 0 | 13 | 13 | -1 | 0 | 0
14 | 1 | 13 | 10 | 14 | 1 | 1
15 | 2 | 13 | 5 | 10 | 1 | 2
16 | 3 | 13 | 2 | 4 | 1 | 3
17 | 3 | 13 | 8 | 7 | 1 | 3
(17 rows)

```

#### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"> <li>Used on <b>Single vertex</b></li> <li>When value is \(\(0\) then gets the spanning forest starting in aleatory nodes for each tree in the forest.</li> </ul>
<b>Root vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. <ul style="list-style-type: none"> <li>Used on <b>Multiple vertices</b></li> <li>\(\(0\) values are ignored</li> <li>For optimization purposes, any duplicated value is ignored.</li> </ul>

#### Optional Parameters

Parameter	Type	Default	Description
<b>max_depth</b>	BIGINT	\(9223372036854775807\)	Upper limit for depth of node in the tree <ul style="list-style-type: none"> <li>When value is <b>Negative</b> then <b>throws error</b></li> </ul>

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICALS:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from \(\(1\)

Column	Type	Description
<b>depth</b>	BIGINT	Depth of the <code>node</code> . <ul style="list-style-type: none"> <li><math>\backslash(0\backslash)</math> when <code>node = start_vid</code>.</li> </ul>
<b>start_vid</b>	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"> <li>In <b>Multiple Vertices</b> results are in ascending order.</li> </ul>
<b>node</b>	BIGINT	Identifier of <code>node</code> reached using <code>edge</code> .
<b>edge</b>	BIGINT	Identifier of the <code>edge</code> used to arrive to <code>node</code> . <ul style="list-style-type: none"> <li><math>\backslash(-1\backslash)</math> when <code>node = start_vid</code>.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse <code>edge</code> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> .

See Also

- [Spanning Tree - Category](#)
- [Prim - Family of functions](#)
- The queries use the [Sample Data](#) network.
- [Boost: Prim's algorithm documentation](#)
- [Wikipedia: Prim's algorithm](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

`pgr_primDD`

`pgr_primDD` — Catchment nodes using Prim's algorithm.



Boost Graph Inside

## Availability

- Version 3.0.0
  - New **Official** function

## Support

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

Description

Using Prim algorithm, extracts the nodes that have aggregate costs less than or equal to the value `Distance` 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:  $\backslash(O(E*\log V)\backslash)$
- Returned tree nodes from a root vertex are on Depth First Search order.
- Depth First Search running time:  $\backslash(O(E + V)\backslash)$

Signatures

## Summary



```
pgr_prim(Edges SQL, root vid, distance)
pgr_prim(Edges SQL, root vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

### Single vertex

```
pgr_primDD(Edges SQL, root vid, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

#### Example:

The Minimum Spanning Tree starting on vertex \(\{2\}\) with \(\text{agg\\_cost} \leq 3.5\)

```
SELECT * FROM pgr_primDD(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2, 3.5
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 2 | 2 | 6 | 5 | 1 | 2
 6 | 3 | 2 | 9 | 9 | 1 | 3
 7 | 3 | 2 | 11 | 11 | 1 | 3
 8 | 1 | 2 | 5 | 4 | 1 | 1
 9 | 2 | 2 | 8 | 7 | 1 | 2
10 | 3 | 2 | 7 | 6 | 1 | 3
11 | 2 | 2 | 10 | 10 | 1 | 2
12 | 3 | 2 | 13 | 14 | 1 | 3
(12 rows)
```

### Multiple vertices

```
pgr_primDD(Edges SQL, root vids, distance)

RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

#### Example:

The Minimum Spanning Tree starting on vertices \(\{13, 2\}\) with \(\text{agg\\_cost} \leq 3.5\);

```
SELECT * FROM pgr_primDD(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  ARRAY[13,2], 3.5
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 2 | 2 | 6 | 5 | 1 | 2
 6 | 3 | 2 | 9 | 9 | 1 | 3
 7 | 3 | 2 | 11 | 11 | 1 | 3
 8 | 1 | 2 | 5 | 4 | 1 | 1
 9 | 2 | 2 | 8 | 7 | 1 | 2
10 | 3 | 2 | 7 | 6 | 1 | 3
11 | 2 | 2 | 10 | 10 | 1 | 2
12 | 3 | 2 | 13 | 14 | 1 | 3
13 | 0 | 13 | 13 | -1 | 0 | 0
14 | 1 | 13 | 10 | 14 | 1 | 1
15 | 2 | 13 | 5 | 10 | 1 | 2
16 | 3 | 13 | 2 | 4 | 1 | 3
17 | 3 | 13 | 8 | 7 | 1 | 3
(17 rows)
```

### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"> <li>• Used on <b>Single vertex</b></li> <li>• When \(\{0\}\) gets the spanning forest starting in aleatory nodes for each tree.</li> </ul>

Parameter	Type	Description
<b>Root vids</b>	ARRAY[ANY-INTEGERS]	Array of identifiers of the root vertices. <ul style="list-style-type: none"> <li>Used on <b>Multiple vertices</b></li> <li>\(0\) values are ignored</li> <li>For optimization purposes, any duplicated value is ignored.</li> </ul>
<b>Distance</b>	ANY-NUMERIC	Upper limit for the inclusion of the node in the result. <ul style="list-style-type: none"> <li>When the value is Negative <b>throws error</b></li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERIC:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT, NUMERIC

Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from \(\1\).
<b>depth</b>	BIGINT	Depth of the <i>node</i> . <ul style="list-style-type: none"> <li>\(0\) when <i>node</i> = <i>start_vid</i>.</li> </ul>
<b>start_vid</b>	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"> <li>In <b>Multiple Vertices</b> results are in ascending order.</li> </ul>
<b>node</b>	BIGINT	Identifier of <i>node</i> reached using <i>edge</i> .
<b>edge</b>	BIGINT	Identifier of the <i>edge</i> used to arrive to <i>node</i> . <ul style="list-style-type: none"> <li>\(-1\) when <i>node</i> = <i>start_vid</i>.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse <i>edge</i> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <i>start_vid</i> to <i>node</i> .

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**



Boost Graph Inside

## Availability

- Version 3.0.0
  - New **Official** function

## Support

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

## Description

Visits and extracts the nodes information in Depth First Search ordering of the Minimum Spanning Tree created using 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:  $\mathcal{O}(E \cdot \log V)$
- Returned tree nodes from a root vertex are on Depth First Search order
- Depth First Search Running time:  $\mathcal{O}(E + V)$

## 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  $\backslash(2\backslash)$

```
SELECT * FROM pgr_primDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2
);
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 2 | 2 | 6 | 5 | 1 | 2
 6 | 3 | 2 | 9 | 9 | 1 | 3
 7 | 3 | 2 | 11 | 11 | 1 | 3
 8 | 4 | 2 | 12 | 13 | 1 | 4
 9 | 1 | 2 | 5 | 4 | 1 | 1
10 | 2 | 2 | 8 | 7 | 1 | 2
11 | 3 | 2 | 7 | 6 | 1 | 3
12 | 2 | 2 | 10 | 10 | 1 | 2
13 | 3 | 2 | 13 | 14 | 1 | 3
(13 rows)
```

## Multiple vertices

```
pgr_primDFS(Edges SQL, Root vids [, max_depth])
```

```
RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
```

### Example:

The Minimum Spanning Tree starting on vertices  $\{13, 2\}$  with  $(depth \leq 3)$

```
SELECT * FROM pgr_primDFS(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  ARRAY[13,2], max_depth := 3
);
```

```
seq | depth | start_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 0 | 2 | 2 | -1 | 0 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 1
 3 | 1 | 2 | 3 | 2 | 1 | 1
 4 | 2 | 2 | 4 | 3 | 1 | 2
 5 | 2 | 2 | 6 | 5 | 1 | 2
 6 | 3 | 2 | 9 | 9 | 1 | 3
 7 | 3 | 2 | 11 | 11 | 1 | 3
 8 | 1 | 2 | 5 | 4 | 1 | 1
 9 | 2 | 2 | 8 | 7 | 1 | 2
10 | 3 | 2 | 7 | 6 | 1 | 3
11 | 2 | 2 | 10 | 10 | 1 | 2
12 | 3 | 2 | 13 | 14 | 1 | 3
13 | 0 | 13 | 13 | -1 | 0 | 0
14 | 1 | 13 | 10 | 14 | 1 | 1
15 | 2 | 13 | 5 | 10 | 1 | 2
16 | 3 | 13 | 2 | 4 | 1 | 3
17 | 3 | 13 | 8 | 7 | 1 | 3
(17 rows)
```

### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"> <li>Used on <b>Single vertex</b></li> <li>When value is <math>\{0\}</math> then gets the spanning forest starting in aleatory nodes for each tree in the forest.</li> </ul>
<b>Root vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. <ul style="list-style-type: none"> <li>Used on <b>Multiple vertices</b></li> <li><math>\{0\}</math> values are ignored</li> <li>For optimization purposes, any duplicated value is ignored.</li> </ul>

### Optional Parameters

Parameter	Type	Default	Description
<b>max_depth</b>	BIGINT	$\{9223372036854775807\}$	Upper limit for depth of node in the tree <ul style="list-style-type: none"> <li>When value is <b>Negative</b> then <b>throws error</b></li> </ul>

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
seq	BIGINT	Sequential value starting from \{1\}.
depth	BIGINT	Depth of the node. <ul style="list-style-type: none"><li>\{0\} when node = start_vid.</li></ul>
start_vid	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"><li>In <b>Multiple Vertices</b> results are in ascending order.</li></ul>
node	BIGINT	Identifier of node reached using edge.
edge	BIGINT	Identifier of the edge used to arrive to node. <ul style="list-style-type: none"><li>\{-1\} when node = start_vid.</li></ul>
cost	FLOAT	Cost to traverse edge.
agg_cost	FLOAT	Aggregate cost from start_vid to node.

See Also

- Spanning Tree - Category
- Prim - Family of functions
- The queries use the **Sample Data** network.
- Boost: Prim's algorithm documentation**
- Wikipedia: Prim's algorithm**

## Indices and tables

- Index**
- Search Page**
- Supported versions:** current(3.0)

## Description

The prim algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník. It is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

This algorithms find the minimum spanning forest in a possibly disconnected graph; in contrast, the most basic form of Prim's algorithm only finds minimum spanning trees in connected graphs. However, running Prim's algorithm separately for each connected component of the graph, then it is called minimum spanning forest.

## The main characteristics are:

- It's implementation is only on **undirected graph**.
- Process is done only on edges with positive costs.
- When the graph is connected
  - The resulting edges make up a tree
- When the graph is not connected,
  - Finds a minimum spanning tree for each connected component.
  - The resulting edges make up a forest.
- Prim's running time:  $O(E \cdot \log V)$



## Note

From boost Graph: "The algorithm as implemented in Boost.Graph does not produce correct results on graphs with parallel edges."

## Inner query

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

Column	Type	Default	Description
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

See Also

- Spanning Tree - Category
- Boost: Prim's algorithm documentation
- Wikipedia: Prim's algorithm

**Indices and tables**

- Index
- Search Page

**Topology - Family of Functions**

The pgRouting's topology of a network, represented with an edge table with source and target attributes and a vertices table associated with it. Depending on the algorithm, you can create a topology or just reconstruct the vertices table, You can analyze the topology, We also provide a function to node an unoded network.

- pgr\_createTopology** - to create a topology based on the geometry.
- pgr\_createVerticesTable** - to reconstruct the vertices table based on the source and target information.
- pgr\_analyzeGraph** - to analyze the edges and vertices of the edge table.
- pgr\_analyzeOneWay** - to analyze directionality of the edges.
- pgr\_nodeNetwork** -to create nodes to a not noded edge table.

**Experimental**



**Warning**

Possible server crash

- These functions might create a server crash



**Warning**

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

- pgr\_extractVertices - Experimental** - Extracts vertices information based on the source and target.

`pgr_createTopology` — Builds a network topology based on the geometry information.

## Availability

- Version 2.0.0
  - Renamed from version 1.x
  - Official** function

## Support

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

## Description

The function returns:

- OK after the network topology has been built and the vertices table created.
- FAIL when the network topology was not built due to an error.

## Signatures

```
varchar pgr_createTopology(text edge_table, double precision tolerance,  
    text the_geom='the_geom', text id='id',  
    text source='source', text target='target',  
    text rows_where='true', boolean clean:=false)
```

## Parameters

The topology creation function accepts the following parameters:

### edge\_table:

text Network table name. (may contain the schema name AS well)

### tolerance:

float8 Snapping tolerance of disconnected edges. (in projection unit)

### the\_geom:

text Geometry column name of the network table. Default value is `the_geom`.

### id:

text Primary key column name of the network table. Default value is `id`.

### source:

text Source column name of the network table. Default value is `source`.

### target:

text Target column name of the network table. Default value is `target`.

### rows\_where:

text Condition to SELECT a subset or rows. Default value is `true` to indicate all rows that `where_source` or `target` have a null value, otherwise the condition is used.

### clean:

boolean Clean any previous topology. Default value is `false`.



## Warning

The `edge_table` will be affected

- The `source` column values will change.
- The `target` column values will change.
  - An index will be created, if it doesn't exist, to speed up the process to the following columns:
    - `id`
    - `the_geom`
    - `source`
    - `target`

The function returns:

- OK after the network topology has been built.
  - Creates a vertices table: `<edge_table>_vertices_pgr`.
  - Fills `id` and `the_geom` columns of the vertices table.
  - Fills the source and target columns of the edge table referencing the `id` of the vertices table.
- FAIL when the network topology was not built due to an error:
  - A required column of the Network table is not found or is not of the appropriate type.
  - The condition is not well formed.
  - The names of source, target or id are the same.

- The SRID of the geometry could not be determined.

## The Vertices Table

The vertices table is a requirement of the `pgr_analyzeGraph` and the `pgr_analyzeOneWay` functions.

The structure of the vertices table is:

### id:

`bigint` Identifier of the vertex.

### cnt:

`integer` Number of vertices in the `edge_table` that reference this vertex. See `pgr_analyzeGraph`.

### chk:

`integer` Indicator that the vertex might have a problem. See `pgr_analyzeGraph`.

### ein:

`integer` Number of vertices in the `edge_table` that reference this vertex AS incoming. See `pgr_analyzeOneWay`.

### eout:

`integer` Number of vertices in the `edge_table` that reference this vertex AS outgoing. See `pgr_analyzeOneWay`.

### the\_geom:

`geometry` Point geometry of the vertex.

Usage when the edge table's columns MATCH the default values:

The simplest way to use `pgr_createTopology` is:

```
SELECT pgr_createTopology('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 18 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)
```

## When the arguments are given in the order described in the parameters:

We get the same result AS the simplest way to use the function.

```
SELECT pgr_createTopology('edge_table', 0.001,
    'the_geom', 'id', 'source', 'target');
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 18 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)
```



## Warning

An error would occur when the arguments are not given in the appropriate order:  
In this example, the column `id` of the table `edge_table` is passed to the function as the geometry column,  
and the geometry column `the_geom` is passed to the function as the `id` column.

```
SELECT pgr_createTopology('edge_table', 0.001,
    'id', 'the_geom');
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'id', 'the_geom', 'source', 'target', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait .....
NOTICE: ----> PGR ERROR in pgr_createTopology: Wrong type of Column id:the_geom
NOTICE: Unexpected error raise_exception
pgr_createtopology
-----
FAIL
(1 row)
```



## When using the named notation

Parameters defined with a default value can be omitted, as long as the value matches the default. And the order of the parameters would not matter.

```
SELECT pgr_createTopology('edge_table', 0.001,
  the_geom:= 'the_geom', id:= 'id', source:= 'source', target:= 'target');
pgr_createtopology
-----
OK
(1 row)
```

```
SELECT pgr_createTopology('edge_table', 0.001,
  source:= 'source', id:= 'id', target:= 'target', the_geom:= 'the_geom');
pgr_createtopology
-----
OK
(1 row)
```

```
SELECT pgr_createTopology('edge_table', 0.001, source:= 'source');
pgr_createtopology
-----
OK
(1 row)
```

## Selecting rows using rows\_where parameter

Selecting rows based on the id.

```
SELECT pgr_createTopology('edge_table', 0.001, rows_where:= 'id < 10');
pgr_createtopology
-----
OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of row with `id = 5`.

```
SELECT pgr_createTopology('edge_table', 0.001,
  rows_where:= 'the_geom && (SELECT st_buffer(the_geom, 0.05) FROM edge_table WHERE id=5)');
pgr_createtopology
-----
OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of the row with `gid = 100` of the table `othertable`.

```
CREATE TABLE otherTable AS (SELECT 100 AS gid, st_point(2.5, 2.5) AS other_geom);
SELECT 1
SELECT pgr_createTopology('edge_table', 0.001,
  rows_where:= 'the_geom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100)');
pgr_createtopology
-----
OK
(1 row)
```

Usage when the edge table's columns DO NOT MATCH the default values:

For the following table

```
CREATE TABLE mytable AS (SELECT id AS gid, the_geom AS mygeom, source AS src, target AS tgt FROM edge_table);
SELECT 18
```

## Using positional notation:

The arguments need to be given in the order described in the parameters.

Note that this example uses clean flag. So it recreates the whole vertices table.

```
SELECT pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', clean := TRUE);
pgr_createtopology
```

```
-----
OK
(1 row)
```



### Warning

An error would occur when the arguments are not given in the appropriate order:  
In this example, the column `gid` of the table `mytable` is passed to the function AS the geometry column,  
and the geometry column `mygeom` is passed to the function AS the id column.

```
SELECT pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt');
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait .....
NOTICE: ----> PGR ERROR in pgr_createTopology: Wrong type of Column id:mygeom
NOTICE: Unexpected error raise_exception
pgr_createtopology
```

```
-----
FAIL
(1 row)
```

### When using the named notation

In this scenario omitting a parameter would create an error because the default values for the column names do not match the column names of the table. The order of the parameters do not matter:

```
SELECT pgr_createTopology('mytable', 0.001, the_geom:='mygeom', id:='gid', source:='src', target:='tgt');
pgr_createtopology
```

```
-----
OK
(1 row)
```

```
SELECT pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom');
pgr_createtopology
```

```
-----
OK
(1 row)
```

### Selecting rows using rows\_where parameter

Based on id:

```
SELECT pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', rows_where:='gid < 10');
pgr_createtopology
```

```
-----
OK
(1 row)
```

```
SELECT pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom', rows_where:='gid < 10');
pgr_createtopology
```

```
-----
OK
(1 row)
```

```
SELECT pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt',
  rows_where:='mygeom && (SELECT st_buffer(mygeom, 1) FROM mytable WHERE gid=5)');
pgr_createtopology
```

```
-----
OK
(1 row)
```

```
SELECT pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='mygeom',
  rows_where:='mygeom && (SELECT st_buffer(mygeom, 1) FROM mytable WHERE gid=5)');
pgr_createtopology
```

```
-----
OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of the row with `gid = 100` of the table `othertable`.

```

SELECT pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt',
  rows_where:=mygeom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100));
pgr_createtopology
-----
OK
(1 row)

SELECT pgr_createTopology('mytable', 0.001, source:=src', id:=gid', target:=tgt', the_geom:=mygeom',
  rows_where:=mygeom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100));
pgr_createtopology
-----
OK
(1 row)

```

## Additional Examples

### Example:

With full output

This example start a clean topology, with 5 edges, and then its incremented to the rest of the edges.

```

SELECT pgr_createTopology('edge_table', 0.001, rows_where:=id < 6', clean := true);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'id < 6', clean := t)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 5 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)

SELECT pgr_createTopology('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 13 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)

```

The example uses the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr\_createVerticesTable** to reconstruct the vertices table based on the source and target information.
- **pgr\_analyzeGraph** to analyze the edges and vertices of the edge table.

## Indices and tables

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

### pgr\_extractVertices - Experimental

pgr\_extractVertices — Extracts the vertices information based on the source and target.



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

### Availability

- Version 3.0.0
  - New **experimental** function

### Support

- Supported versions:** current(3.0)

### Description

This is an auxiliary function for extracting the vertex information of the set of edges of a graph.

- When the edge identifier is given, then it will also calculate the in and out edges

### Signatures

```
pgr_extractVertices(Edges SQL [, dryrun])
RETURNS SETOF (id, in_edges, out_edges, x, y, geom)
```

### Example:

Extracting the vertex information

```
SELECT * FROM pgr_extractVertices(
'SELECT id, the_geom AS geom
FROM edge_table');
 id | in_edges | out_edges | x   | y   | geom
-----+-----+-----+-----+-----+-----
 1 |   {6}   |   {1}    |  0  |  2  | 010100000000000000000000000000000000000000000040
 2 |   {17}  |   {7}    |  0.5| 3.5 | 01010000000000000000000000E03F00000000000000C40
 3 | {6}    | {7}     |  1  |  2  | 01010000000000000000000000F03F0000000000000040
 4 | {17}   |   {1}    | 1.999999999999999 | 3.5 | 010100000068EEFFFFFFFFF3F0000000000000C40
 5 |   {1}   |   {2}    |  2  |  0  | 01010000000000000000000000400000000000000000
 6 | {1}    | {2,4}   |  2  |  1  | 01010000000000000000000040000000000000F03F
 7 | {4,7}  | {8,10}  |  2  |  2  | 010100000000000000000000400000000000000040
 8 | {10}   | {12,14} |  2  |  3  | 01010000000000000000000040000000000000840
 9 | {14}   |   {2}    |  2  |  4  | 010100000000000000000000400000000000001040
10 | {2}    | {3,5}   |  3  |  1  | 01010000000000000000000084000000000000F03F
11 | {5,8}  | {9,11}  |  3  |  2  | 01010000000000000000000084000000000000040
12 | {11,12}| {13}    |  3  |  3  | 01010000000000000000000084000000000000840
13 |   {18} |   {1}    | 3.5| 2.3 | 010100000000000000000000C40666666666666660240
14 | {18}   |   {4}    | 3.5|  4  | 010100000000000000000000C40000000000001040
15 | {3}    | {16}    |  4  |  1  | 01010000000000000000000010400000000000F03F
16 | {9,16} | {15}    |  4  |  2  | 01010000000000000000000010400000000000040
17 | {13,15}|   {3}    |  4  |  3  | 01010000000000000000000010400000000000840
(17 rows)
```

### Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	The set of edges of the graph. It is an <b>Inner Query</b> as described below.
<b>dryrun</b>	TEXT	Don't process and get in a NOTICE the resulting query.

### Inner Query

### When line geometry is known

Column	Type	Description
<b>id</b>	BIGINT	(Optional) identifier of the edge.
<b>geom</b>	LINSTRING	LINSTRING geometry of the edge.

This inner query takes precedence over the next two inner query, therefore other columns are ignored whergeom column appears.

- o Ignored columns:
  - o startpoint
  - o endpoint
  - o source
  - o target

### When vertex geometry is known

To use this inner query the columngeom 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 wherstartpoint and endpoint columns appears.

- o Ignored columns:
  - o source
  - o target

### When identifiers of vertices are known

To use this inner query the columnsgeom, startpoint and endpoint should not be part of the set of columns.

Column	Type	Description
id	BIGINT	(Optional) identifier of the edge.
source	ANY-INTEGERS	Identifier of the first end point vertex of the edge.
target	ANY-INTEGERS	Identifier of the second end point vertex of the edge.

Result Columns

Rreturns set of (id, in\_edges, out\_edges, x, y, geom)

Column	Type	Description
id	BIGINT	Identifier of the first end point vertex of the edge.
in_edges	BIGINT[]	Array of identifiers of the edges that have the vertexid as <i>first end point</i> . <ul style="list-style-type: none"> <li>o NULL When the id is not part of the inner query</li> </ul>
out_edges	BIGINT[]	Array of identifiers of the edges that have the vertexid as <i>second end point</i> . <ul style="list-style-type: none"> <li>o NULL When the id is not part of the inner query</li> </ul>
x	FLOAT	X value of the POINT geometry <ul style="list-style-type: none"> <li>o NULL When no geometry is provided</li> </ul>
y	FLOAT	Y value of the POINT geometry <ul style="list-style-type: none"> <li>o NULL When no geometry is provided</li> </ul>
geom	POINT	Geometry of the POINT <ul style="list-style-type: none"> <li>o NULL When no geometry is provided</li> </ul>

Additional Examples

#### Example 1:

Dryrun execution

To get the query generated used to get the vertex information, usedryrun := true.

The results can be used as base code to make a refinement based on the backend development needs.

```

SELECT * FROM pgr_extractVertices(
'SELECT id, the_geom AS geom FROM edge_table',
dryrun := true);
NOTICE:
WITH

main_sql AS (
SELECT id, the_geom AS geom FROM edge_table
),

the_out AS (
SELECT id::BIGINT AS out_edge, ST_StartPoint(geom) AS geom
FROM main_sql
),

agg_out AS (
SELECT array_agg(out_edge ORDER BY out_edge) AS out_edges, ST_x(geom) AS x, ST_Y(geom) AS y, geom
FROM the_out
GROUP BY geom
),

the_in AS (
SELECT id::BIGINT AS in_edge, ST_EndPoint(geom) AS geom
FROM main_sql
),

agg_in AS (
SELECT array_agg(in_edge ORDER BY in_edge) AS in_edges, ST_x(geom) AS x, ST_Y(geom) AS y, geom
FROM the_in
GROUP BY geom
),

the_points AS (
SELECT in_edges, out_edges, coalesce(agg_out.geom, agg_in.geom) AS geom
FROM agg_out
FULL OUTER JOIN agg_in USING (x, y)
)

SELECT row_number() over(ORDER BY ST_X(geom), ST_Y(geom)) AS id, in_edges, out_edges, ST_X(geom), ST_Y(geom), geom
FROM the_points;
id | in_edges | out_edges | x | y | geom
-----+-----+-----+---+---+-----
(0 rows)

```

## Example 2:

Creating a routing topology

1. Making sure the database does not have the `vertices_table`

```

DROP TABLE IF EXISTS vertices_table;
NOTICE: table "vertices_table" does not exist, skipping
DROP TABLE

```

2. Cleaning up the columns of the routing topology to be created

```

UPDATE edge_table
SET source = NULL, target = NULL,
    x1 = NULL, y1 = NULL,
    x2 = NULL, y2 = NULL;
UPDATE 18

```

3. Creating the vertices table

```

SELECT * INTO vertices_table
FROM pgr_extractVertices('SELECT id, the_geom AS geom FROM edge_table');
SELECT 17

```

4. Inspection of the vertices table

```

SELECT *
FROM vertices_table;
id | in_edges | out_edges | x | y | geom
-----+-----+-----+---+---+-----
1 |  | {6} | 0 | 2 | 01010000000000000000000000000000000000000000000040
2 |  | {17} | 0.5 | 3.5 | 01010000000000000000000000000000E03F0000000000000C40
3 | {6} | {7} | 1 | 2 | 01010000000000000000000000000000F03F0000000000000040
4 | {17} |  | 1.999999999999 | 3.5 | 010100000068EEFFFFFFF3F000000000000000C40
5 |  | {1} | 2 | 0 | 01010000000000000000000000000000400000000000000000
6 | {1} | {2,4} | 2 | 1 | 010100000000000000000000000000004000000000000000F03F
7 | {4,7} | {8,10} | 2 | 2 | 0101000000000000000000000000000040000000000000000040
8 | {10} | {12,14} | 2 | 3 | 010100000000000000000000000000004000000000000000840
9 | {14} |  | 2 | 4 | 0101000000000000000000000000000040000000000000001040
10 | {2} | {3,5} | 3 | 1 | 0101000000000000000000000000000084000000000000000F03F
11 | {5,8} | {9,11} | 3 | 2 | 01010000000000000000000000000000840000000000000000040
12 | {11,12} | {13} | 3 | 3 | 01010000000000000000000000000000840000000000000000840
13 |  | {18} | 3.5 | 2.3 | 01010000000000000000000000000000C40666666666660240
14 | {18} |  | 3.5 | 4 | 01010000000000000000000000000000C40000000000000001040
15 | {3} | {16} | 4 | 1 | 01010000000000000000000000000000104000000000000000F03F
16 | {9,16} | {15} | 4 | 2 | 01010000000000000000000000000000104000000000000000040
17 | {13,15} |  | 4 | 3 | 0101000000000000000000000000000010400000000000000840
(17 rows)

```

### 5. Creating the routing topology on the edge table

Updating the `source` information

```

WITH
  out_going AS (
    SELECT id AS vid, unnest(out_edges) AS eid, x, y
    FROM vertices_table
  )
UPDATE edge_table
SET source = vid, x1 = x, y1 = y
FROM out_going WHERE id = eid;
UPDATE 18

```

Updating the `target` information

```

WITH
  in_coming AS (
    SELECT id AS vid, unnest(in_edges) AS eid, x, y
    FROM vertices_table
  )
UPDATE edge_table
SET target = vid, x2 = x, y2 = y
FROM in_coming WHERE id = eid;
UPDATE 18

```

### 6. Inspection of the routing topology

```

SELECT id, source, target, x1, y1, x2, y2
FROM edge_table;
id | source | target | x1 | y1 | x2 | y2
-----+-----+-----+---+---+---+---
6 | 1 | 3 | 0 | 2 | 1 | 2
17 | 2 | 4 | 0.5 | 3.5 | 1.999999999999 | 3.5
1 | 5 | 6 | 2 | 0 | 2 | 1
4 | 6 | 7 | 2 | 1 | 2 | 2
7 | 3 | 7 | 1 | 2 | 2 | 2
10 | 7 | 8 | 2 | 2 | 2 | 3
14 | 8 | 9 | 2 | 3 | 2 | 4
2 | 6 | 10 | 2 | 1 | 3 | 1
5 | 10 | 11 | 3 | 1 | 3 | 2
8 | 7 | 11 | 2 | 2 | 3 | 2
11 | 11 | 12 | 3 | 2 | 3 | 3
12 | 8 | 12 | 2 | 3 | 3 | 3
18 | 13 | 14 | 3.5 | 2.3 | 3.5 | 4
3 | 10 | 15 | 3 | 1 | 4 | 1
9 | 11 | 16 | 3 | 2 | 4 | 2
16 | 15 | 16 | 4 | 1 | 4 | 2
13 | 12 | 17 | 3 | 3 | 4 | 3
15 | 16 | 17 | 4 | 2 | 4 | 3
(18 rows)

```

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr\_createVerticesTable** to create a topology based on the geometry.

### Indices and tables

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

pgr\_createVerticesTable — Reconstructs the vertices table based on the source and target information.

## Availability

- Version 2.0.0
  - Renamed from version 1.x
  - Official** function

## Support

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

## Description

The function returns:

- OK** after the vertices table has been reconstructed.
- FAIL** when the vertices table was not reconstructed due to an error.

## Signatures

```
pgr_createVerticesTable(edge_table, the_geom, source, target, rows_where)
RETURNS VARCHAR
```

## Parameters

The reconstruction of the vertices table function accepts the following parameters:

### edge\_table:

`text` Network table name. (may contain the schema name as well)

### the\_geom:

`text` Geometry column name of the network table. Default value is `the_geom`.

### source:

`text` Source column name of the network table. Default value is `source`.

### target:

`text` Target column name of the network table. Default value is `target`.

### rows\_where:

`text` Condition to SELECT a subset or rows. Default value is `true` to indicate all rows.



### Warning

The `edge_table` will be affected

- An index will be created, if it doesn't exist, 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 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.

### Example 1:

The simplest way to use pgr\_createVerticesTable

```
SELECT pgr_createVerticesTable('edge_table');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: ----> VERTICES TABLE CREATED WITH 17 VERTICES
NOTICE:                FOR 18 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 18
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)
```

### Additional Examples

#### Example 2:

When the arguments are given in the order described in the parameters:

```
SELECT pgr_createVerticesTable('edge_table', 'the_geom', 'source', 'target');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: ----> VERTICES TABLE CREATED WITH 17 VERTICES
NOTICE:                FOR 18 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 18
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)
```

We get the same result as the simplest way to use the function.



### Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the column source column `source` of the table `mytable` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the source column.

```
SELECT pgr_createVerticesTable('edge_table', 'source', 'the_geom', 'target');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','source','the_geom','target','true')
NOTICE: Performing checks, please wait .....
NOTICE: ----> PGR ERROR in pgr_createVerticesTable: Wrong type of Column source: the_geom
HINT: ----> Expected type of the_geom is integer, smallint or bigint but USER-DEFINED was found
NOTICE: Unexpected error raise_exception
pgr_createverticestable
-----
FAIL
(1 row)
```

### When using the named notation

#### Example 3:

The order of the parameters do not matter:

```

SELECT pgr_createVerticesTable('edge_table',the_geom:='the_geom',source:='source',target:='target');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target',true)
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: ----> VERTICES TABLE CREATED WITH 17 VERTICES
NOTICE:                FOR 18 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 18
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```

#### Example 4:

Using a different ordering

```

SELECT pgr_createVerticesTable('edge_table',source:='source',target:='target',the_geom:='the_geom');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target',true)
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: ----> VERTICES TABLE CREATED WITH 17 VERTICES
NOTICE:                FOR 18 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 18
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```

#### Example 5:

Parameters defined with a default value can be omitted, as long as the value matches the default:

```

SELECT pgr_createVerticesTable('edge_table',source:='source');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target',true)
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: ----> VERTICES TABLE CREATED WITH 17 VERTICES
NOTICE:                FOR 18 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 18
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```

### Selecting rows using rows\_where parameter

#### Example 6:

Selecting rows based on the id.

```

SELECT pgr_createVerticesTable('edge_table',rows_where:='id < 10');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target','id < 10')
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: ----> VERTICES TABLE CREATED WITH 9 VERTICES
NOTICE:                FOR 10 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 10
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```

#### Example 7:

Selecting the rows where the geometry is near the geometry of row with id =5 .

```

SELECT pgr_createVerticesTable('edge_table',
  rows_where:='the_geom && (select st_buffer(the_geom,0.5) FROM edge_table WHERE id=5)');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target','the_geom && (select st_buffer(the_geom,0.5) FROM edge_table WHERE id=5)')
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: -----> VERTICES TABLE CREATED WITH 9 VERTICES
NOTICE:                FOR 9 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 9
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```

### Example 8:

Selecting the rows where the geometry is near the geometry of the row with `gid =100` of the table `othertable`.

```

DROP TABLE IF EXISTS otherTable;
NOTICE: table "othertable" does not exist, skipping
DROP TABLE
CREATE TABLE otherTable AS (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom);
SELECT 1
SELECT pgr_createVerticesTable('edge_table',
  rows_where:='the_geom && (select st_buffer(other_geom,0.5) FROM otherTable WHERE gid=100)');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('edge_table','the_geom','source','target','the_geom && (select st_buffer(other_geom,0.5) FROM otherTable WHERE gid=100)')
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.edge_table_vertices_pgr, please wait...
NOTICE: -----> VERTICES TABLE CREATED WITH 10 VERTICES
NOTICE:                FOR 12 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 12
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```

Usage when the edge table's columns DO NOT MATCH the default values:

Using the following table

```

DROP TABLE IF EXISTS mytable;
NOTICE: table "mytable" does not exist, skipping
DROP TABLE
CREATE TABLE mytable AS (SELECT id AS gid, the_geom AS mygeom, source AS src, target AS tgt FROM edge_table);
SELECT 18

```

### Using positional notation:

#### Example 9:

The arguments need to be given in the order described in the parameters:

```

SELECT pgr_createVerticesTable('mytable', 'mygeom', 'src', 'tgt');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('mytable','mygeom','src','tgt','true')
NOTICE: Performing checks, please wait .....
NOTICE: Populating public.mytable_vertices_pgr, please wait...
NOTICE: -----> VERTICES TABLE CREATED WITH 17 VERTICES
NOTICE:                FOR 18 EDGES
NOTICE: Edges with NULL geometry,source or target: 0
NOTICE:                Edges processed: 18
NOTICE: Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE: -----
pgr_createverticestable
-----
OK
(1 row)

```



### Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the 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 withgid = 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 withgid =100 of the tableothertable.

```
DROP TABLE IF EXISTS otherTable;
DROP TABLE
CREATE TABLE otherTable AS (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom);
SELECT 1
```

```
SELECT pgr_createVerticesTable(
  'mytable', 'mygeom', 'src', 'tgt',
  rows_where:='the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('mytable','mygeom','src','tgt','the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)')
NOTICE: Performing checks, please wait .....
NOTICE: Got column "the_geom" does not exist
NOTICE: ERROR: Condition is not correct, please execute the following query to test your condition
NOTICE: select * from public.mytable WHERE true AND (the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)) limit 1
pgr_createverticestable
-----
FAIL
(1 row)
```

**Example 17:**

TBD

```

SELECT pgr_createVerticesTable(
  'mytable',source:='src',target:='tgt',the_geom:='mygeom',
  rows_where:='the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)');
NOTICE: PROCESSING:
NOTICE: pgr_createVerticesTable('mytable','mygeom','src','tgt','the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)')
NOTICE: Performing checks, please wait .....
NOTICE: Got column "the_geom" does not exist
NOTICE: ERROR: Condition is not correct, please execute the following query to test your condition
NOTICE: select * from public.mytable WHERE true AND (the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTable WHERE gid=100)) limit 1
pgr_createverticestable
-----
FAIL
(1 row)

```

The example uses the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr\_createTopology** <pgr\_create\_topology>` to create a topology based on the geometry.
- **pgr\_analyzeGraph** to analyze the edges and vertices of the edge table.
- **pgr\_analyzeOneWay** to analyze directionality of the edges.

## Indices and tables

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

### pgr\_analyzeGraph

`pgr_analyzeGraph` — Analyzes the network topology.

## Availability

- Version 2.0.0
  - **Official** function

## Support

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

## Description

The function returns:

- OK after the analysis has finished.
- **FAIL** when the analysis was not completed due to an error.

```

varchar pgr_analyzeGraph(text edge_table, double precision tolerance,
  text the_geom:='the_geom', text id:='id',
  text source:='source', text target:='target', text rows_where:='true')

```

## Prerequisites

The edge table to be analyzed must contain a source column and a target column filled with id's of the vertices of the segments and the corresponding vertices table <edge\_table>\_vertices\_pgr that stores the vertices information.

- Use **pgr\_createVerticesTable** to create the vertices table.
- Use **pgr\_createTopology** to create the topology and the vertices table.

## Parameters

The analyze graph function accepts the following parameters:

### edge\_table:

`text` Network table name. (may contain the schema name as well)

### tolerance:

`float8` Snapping tolerance of disconnected edges. (in projection unit)

### the\_geom:

`text` Geometry column name of the network table. Default value is `the_geom`.

### id:

`text` Primary key column name of the network table. Default value is `id`.

### source:

text Source column name of the network table. Default value is `source`.

**target:**

text Target column name of the network table. Default value is `target`.

**rows\_where:**

text Condition to select a subset or rows. Default value is `true` to indicate all rows.

The function returns:

- **OK** after the analysis has finished.
  - Uses the vertices table: `<edge_table>_vertices_pgr`.
  - Fills completely the `cnt` and `chk` columns of the vertices table.
  - Returns the analysis of the section of the network defined by `rows_where`
- **FAIL** when the analysis was not completed due to an error.
  - The vertices table is not found.
  - A required column of the Network table is not found or is not of the appropriate type.
  - The condition is not well formed.
  - The names of source , target or id are the same.
  - The SRID of the geometry could not be determined.

**The Vertices Table**

The vertices table can be created with **`pgr_createVerticesTable`** or **`pgr_createTopology`**

The structure of the vertices table is:

**id:**

bigint Identifier of the vertex.

**cnt:**

integer Number of vertices in the edge\_table that reference this vertex.

**chk:**

integer Indicator that the vertex might have a problem.

**ein:**

integer Number of vertices in the edge\_table that reference this vertex as incoming. See **`pgr_analyzeOneWay`**.

**eout:**

integer Number of vertices in the edge\_table that reference this vertex as outgoing. See **`pgr_analyzeOneWay`**.

**the\_geom:**

geometry Point geometry of the vertex.

Usage when the edge table's columns MATCH the default values:

**The simplest way to use `pgr_analyzeGraph` is:**


```
SELECT pgr_createTopology('edge_table',0.001, clean := true);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := t)
NOTICE: Performing checks, please wait ....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 18 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)

SELECT pgr_analyzeGraph('edge_table',0.001);
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

## When the arguments are given in the order described in the parameters:

```
SELECT pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

We get the same result as the simplest way to use the function.

 **Warning**

An error would occur when the arguments are not given in the appropriate order: In this example, the column `id` of the table `mytable` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the id column.

```
SELECT pgr_analyzeGraph('edge_table',0.001,'id','the_geom','source','target');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'id','the_geom','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Got function st_srid(bigint) does not exist
NOTICE: ERROR: something went wrong when checking for SRID of id in table public.edge_table
pgr_analyzegraph
-----
FAIL
(1 row)
```

## When using the named notation

The order of the parameters do not matter:

```
SELECT pgr_analyzeGraph('edge_table',0.001,the_geom:='the_geom',id:='id',source:='source',target:='target');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```



```

SELECT pgr_analyzeGraph('edge_table',0.001,source:='source',id:='id',target:='target',the_geom:='the_geom');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Parameters defined with a default value can be omitted, as long as the value matches the default:

```

SELECT pgr_analyzeGraph('edge_table',0.001,source:='source');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

### Selecting rows using rows\_where parameter

Selecting rows based on the id. Displays the analysis a the section of the network.

```

SELECT pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 10');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 10')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 4
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Selecting the rows where the geometry is near the geometry of row with `id = 5` .

```

SELECT pgr_analyzeGraph('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(the_geom,0.05) FROM edge_table WHERE id=5)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','the_geom && (SELECT st_buffer(the_geom,0.05) FROM edge_table WHERE id=5)')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 5
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Selecting the rows where the geometry is near the geometry of the row with `gid = 100` of the table `othertable`.

```

CREATE TABLE otherTable AS (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom);
SELECT 1
SELECT pgr_analyzeGraph('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE gid=100)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','the_geom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE gid=100)')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 10
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Usage when the edge table's columns DO NOT MATCH the default values:

For the following table

```

CREATE TABLE mytable AS (SELECT id AS gid, source AS src ,target AS tgt , the_geom AS mygeom FROM edge_table);
SELECT 18
SELECT pgr_createTopology('mytable',0.001,'mygeom','gid','src','tgt', clean := true);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', rows_where := 'true', clean := t)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 18 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.mytable is: public.mytable_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)

```

### Using positional notation:

The arguments need to be given in the order described in the parameters:

```

SELECT pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```



### Warning

An error would occur when the arguments are not given in the appropriate order: In this example, the 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 `gid = 5` .

```
SELECT pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',
rows_where:='mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 1
NOTICE: Dead ends: 5
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

```

SELECT pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom',
  rows_where:='mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE gid=5)')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 1
NOTICE: Dead ends: 5
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Selecting the rows WHERE the geometry is near the place='myhouse' of the table `othertable`. (note the use of quote\_literal)

```

DROP TABLE IF EXISTS otherTable;
DROP TABLE
CREATE TABLE otherTable AS (SELECT 'myhouse'::text AS place, st_point(2.5,2.5) AS other_geom) ;
SELECT 1
SELECT pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',
  rows_where:='mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place=||quote_literal('myhouse')||)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='myhouse')')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 10
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

```

SELECT pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom',
  rows_where:='mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place=||quote_literal('myhouse')||)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt','mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='myhouse')')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 10
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Additional Examples

```
SELECT pgr_createTopology('edge_table',0.001, clean := true);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := t)
NOTICE: Performing checks, please wait ....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 18 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)
```

```
SELECT pgr_analyzeGraph('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

```
SELECT pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 10');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 10')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 4
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

```
SELECT pgr_analyzeGraph('edge_table',0.001,rows_where:='id >= 10');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id >= 10')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 8
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

```

SELECT pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 17');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 17')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 3
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

```

SELECT pgr_createTopology('edge_table', 0.001,rows_where:='id <17', clean := true);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'id <17', clean := t)
NOTICE: Performing checks, please wait ....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 16 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)

```

```

SELECT pgr_analyzeGraph('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 3
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

The examples use the **Sample Data** network.

See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **pgr\_analyzeOneWay** to analyze directionality of the edges.
- **pgr\_createVerticesTable** to reconstruct the vertices table based on the source and target information.
- **pgr\_nodeNetwork** to create nodes to a not noded edge table.

## Indices and tables

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

### pgr\_analyzeOneWay

`pgr_analyzeOneWay` — Analyzes oneway Sstreets and identifies flipped segments.

This function analyzes oneway streets in a graph and identifies any flipped segments.

## Availability

- Version 2.0.0
  - **Official** function

## Support

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

## Description

The analyses of one way segments is pretty simple but can be a powerful tools to identifying some the potential problems created by setting the direction of a segment the wrong way. A node is a *source* if it has edges the exit from that node and no edges enter that node. Conversely, a node is a *sink* if all edges enter the node but none exit that node. For a *source* type node it is logically impossible to exist because no vehicle can exit the node if no vehicle and enter the node. Likewise, if you had a *sink* node you would have an infinite number of vehicle piling up on this node because you can enter it but not leave it.

So why do we care if they are not feasible? Well if the direction of an edge was reversed by mistake we could generate exactly these conditions. Think about a divided highway and on the north bound lane one segment got entered wrong or maybe a sequence of multiple segments got entered wrong or maybe this happened on a round-about. The result would be potentially a *source* and/or a *sink* node.

So by counting the number of edges entering and exiting each node we can identify both *source* and *sink* nodes so that you can look at those areas of your network to make repairs and/or report the problem back to your data vendor.

## Prerequisites

The edge table to be analyzed must contain a source column and a target column filled with id's of the vertices of the segments and the corresponding vertices table <edge\_table>\_vertices\_pgr that stores the vertices information.

- Use **pgr\_createVerticesTable** to create the vertices table.
- Use **pgr\_createTopology** to create the topology and the vertices table.

## Signatures

```
text pgr_analyzeOneWay(geom_table text,
                      text[] s_in_rules, text[] s_out_rules,
                      text[] t_in_rules, text[] t_out_rules,
                      text oneway='oneway', text source='source', text target='target',
                      boolean two_way_if_null=true);
```

## Parameters

### edge\_table:

text Network table name. (may contain the schema name as well)

### s\_in\_rules:

text[] source node **in** rules

### s\_out\_rules:

text[] source node **out** rules

### t\_in\_rules:

text[] target node **in** rules

### t\_out\_rules:

text[] target node **out** rules

### oneway:

text oneway column name name of the network table. Default value is `oneway`.

### source:

text Source column name of the network table. Default value is `source`.

### target:

text Target column name of the network table. Default value is `target`.

### two\_way\_if\_null:

boolean flag to treat oneway NULL values as bi-directional. Default value is `true`.



### Note

It is strongly recommended to use the named notation. See **pgr\_createVerticesTable** or **pgr\_createTopology** for examples.

## The function returns:

- **OK** after the analysis has finished.
  - Uses the vertices table: <edge\_table>\_vertices\_pgr.
  - Fills completely the `ein` and `eout` columns of the vertices table.
- **FAIL** when the analysis was not completed due to an error.
  - The vertices table is not found.
  - A required column of the Network table is not found or is not of the appropriate type.
  - The names of source, target or oneway are the same.



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

## The Vertices Table

The vertices table can be created with `pgr_createVerticesTable` or `pgr_createTopology`

The structure of the vertices table is:

**id:**

`bigint` Identifier of the vertex.

**cnt:**

`integer` Number of vertices in the `edge_table` that reference this vertex. See `pgr_analyzeGgraph`.

**chk:**

`integer` Indicator that the vertex might have a problem. See `pgr_analyzeGraph`.

**ein:**

`integer` Number of vertices in the `edge_table` that reference this vertex as incoming.

**eout:**

`integer` Number of vertices in the `edge_table` that reference this vertex as outgoing.

**the\_geom:**

`geometry` Point geometry of the vertex.

## Additional Examples

```
SELECT pgr_analyzeOneWay('edge_table',
  ARRAY['B', 'TF'],
  ARRAY['B', 'FT'],
  ARRAY['B', 'FT'],
  ARRAY['B', 'TF'],
  oneway:='dir');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeOneWay('edge_table':{'B,TF':{'B,FT':{'B,FT':{'B,TF}'}'}'};dir','source','target',t)
NOTICE: Analyzing graph for one way street errors.
NOTICE: Analysis 25% complete ...
NOTICE: Analysis 50% complete ...
NOTICE: Analysis 75% complete ...
NOTICE: Analysis 100% complete ...
NOTICE: Found 0 potential problems in directionality
pgr_analyzeoneway
-----
OK
(1 row)
```

The queries use the **Sample Data** network.

## See Also

- **Topology - Family of Functions** for an overview of a topology for routing algorithms.
- **Graph Analytics** for an overview of the analysis of a graph.
- `pgr_analyzeGraph` to analyze the edges and vertices of the edge table.
- `pgr_createVerticesTable` to reconstruct the vertices table based on the source and target information.

## Indices and tables

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

### `pgr_nodeNetwork`

`pgr_nodeNetwork` - Nodes an network edge table.

**Author:**

Nicolas Ribot

**Copyright:**

Nicolas Ribot, The source code is released under the MIT-X license.

The function reads edges from a not “noded” network table and writes the “noded” edges into a new table.

```
pgr_nodenetwork(edge_table, tolerance, id, text the_geom, table_ending, rows_where, outall)
RETURNS TEXT
```

## Availability

- Version 2.0.0

- Official function

## Support

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

## Description

### The main characteristics are:

A common problem associated with bringing GIS data into pgRouting is the fact that the data is often not “noded” correctly. This will create invalid topologies, which will result in routes that are incorrect.

What we mean by “noded” is that at every intersection in the road network all the edges will be broken into separate road segments. There are cases like an over-pass and under-pass intersection where you can not traverse from the over-pass to the under-pass, but this function does not have the ability to detect and accommodate those situations.

This function reads the `edge_table` table, that has a primary key column `id` and geometry column named `the_geom` and intersect all the segments in it against all the other segments and then creates a table `edge_table_noded`. It uses the `tolerance` for deciding that multiple nodes within the tolerance are considered the same node.

## Parameters

### edge\_table:

text Network table name. (may contain the schema name as well)

### tolerance:

float8 tolerance for coincident points (in projection unit)

### id:

text Primary key column name of the network table. Default value is `id`.

### the\_geom:

text Geometry column name of the network table. Default value is `the_geom`.

### table\_ending:

text Suffix for the new table's. Default value is `noded`.

The output table will have for `edge_table_noded`

### id:

bigint Unique identifier for the table

### old\_id:

bigint Identifier of the edge in original table

### sub\_id:

integer Segment number of the original edge

### source:

integer Empty source column to be used with `pgr_createTopology` function

### target:

integer Empty target column to be used with `pgr_createTopology` function

### the\_geom:

geometry Geometry column of the noded network

## Examples

Let's create the topology for the data in **Sample Data**

```
SELECT pgr_createTopology('edge_table', 0.001, clean := TRUE);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := t)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 18 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)
```

Now we can analyze the network.

```

SELECT pgr_analyzegraph('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

The analysis tell us that the network has a gap and an intersection. We try to fix the problem using:

```

SELECT pgr_nodeNetwork('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: id: id
NOTICE: the_geom: the_geom
NOTICE: table_ending: noded
NOTICE: rows_where:
NOTICE: outall: f
NOTICE: pgr_nodeNetwork('edge_table', 0.001, 'id', 'the_geom', 'noded', ", f)
NOTICE: Performing checks, please wait .....
NOTICE: Processing, please wait .....
NOTICE: Split Edges: 3
NOTICE: Untouched Edges: 15
NOTICE: Total original Edges: 18
NOTICE: Edges generated: 6
NOTICE: Untouched Edges: 15
NOTICE: Total New segments: 21
NOTICE: New Table: public.edge_table_noded
NOTICE: -----
pgr_nodenetwork
-----
OK
(1 row)

```

Inspecting the generated table, we can see that edges 13,14 and 18 has been segmented

```

SELECT old_id, sub_id FROM edge_table_noded ORDER BY old_id, sub_id;
old_id | sub_id
-----+-----
 1 | 1
 2 | 1
 3 | 1
 4 | 1
 5 | 1
 6 | 1
 7 | 1
 8 | 1
 9 | 1
10 | 1
11 | 1
12 | 1
13 | 1
13 | 2
14 | 1
14 | 2
15 | 1
16 | 1
17 | 1
18 | 1
18 | 2
(21 rows)

```

We can create the topology of the new network

```

SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table_noded', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait .....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 21 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table_noded is: public.edge_table_noded_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)

```

Now let's analyze the new topology

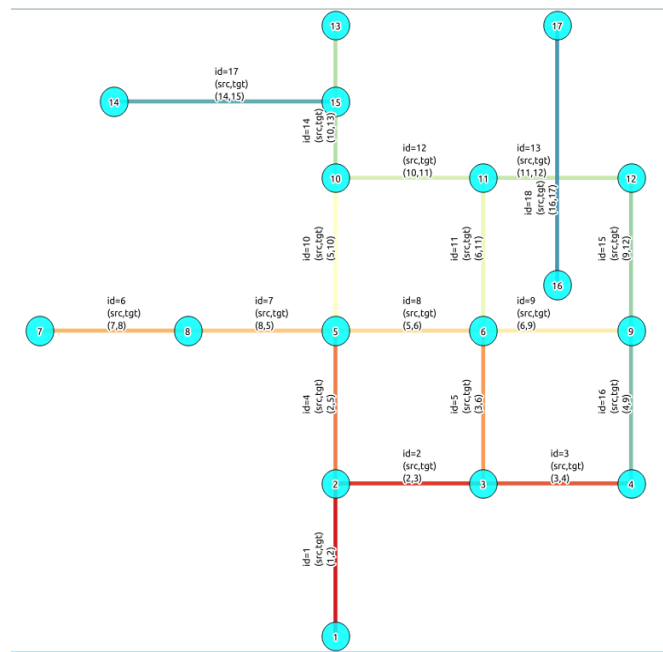
```

SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table_noded', 0.001, 'the_geom', 'id', 'source', 'target', 'true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 6
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
NOTICE: -----
pgr_analyzegraph
-----
OK
(1 row)

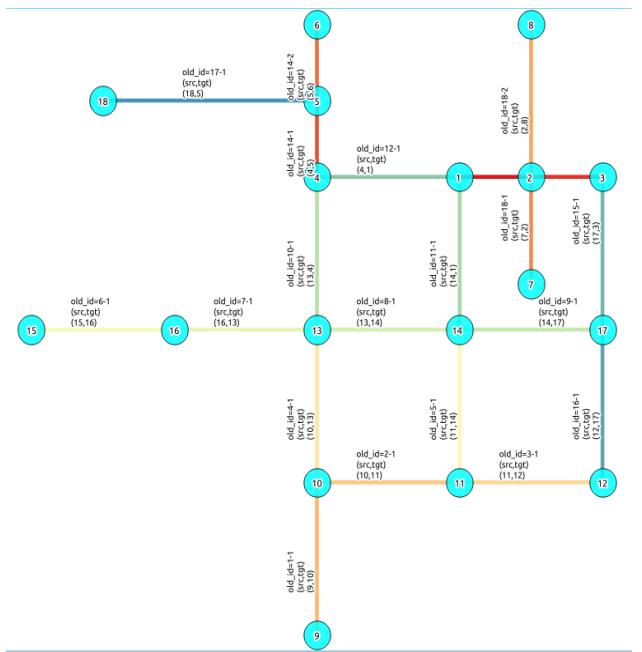
```

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	<ul style="list-style-type: none"> <li>Edges with 1 dead end: 1,6,24</li> <li>Edges with 2 dead ends 17,18</li> </ul>	Edges with 1 dead end: 1-1 , 6-1,14-2, 18-1 17-1 18-2
	Edge 17's right node is a dead end because there is no other edge sharing that same node. (cnt=1)	

	Before	After
Isolated segments	two isolated segments: 17 and 18 both they have 2 dead ends	No Isolated segments <ul style="list-style-type: none"> <li>Edge 17 now shares a node with edges 14-1 and 14-2</li> <li>Edges 18-1 and 18-2 share a node with edges 13-1 and 13-2</li> </ul>
Gaps	There is a gap between edge 17 and 14 because edge 14 is near to the right node of edge 17	Edge 14 was segmented Now edges: 14-1 14-2 17 share the same node The tolerance value was taken in account
Intersections	Edges 13 and 18 were intersecting	Edges were segmented, So, now in the interection's point there is a node and the following edges share it: 13-1 13-2 18-1 18-2

Now, we are going to include the segments 13-1, 13-2 14-1, 14-2 ,18-1 and 18-2 into our edge-table, copying the data for dir,cost,and reverse cost with tho following steps:

- Add a column old\_id into edge\_table, this column is going to keep track the id of the original edge
- Insert only the segmented edges, that is, the ones whose max(sub\_id) >1

```
alter table edge_table drop column if exists old_id;
NOTICE: column "old_id" of relation "edge_table" does not exist, skipping
ALTER TABLE
alter table edge_table add column old_id integer;
ALTER TABLE
insert into edge_table (old_id, dir, cost, reverse_cost, the_geom)
  (with
    segmented as (select old_id,count(*) as i from edge_table_noded group by old_id)
  select segments.old_id, dir, cost, reverse_cost, segments.the_geom
    from edge_table as edges join edge_table_noded as segments on (edges.id = segments.old_id)
   where edges.id in (select old_id from segmented where i>1) );
INSERT 0 6
```

We recreate the topology:

```
SELECT pgr_createTopology('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_createTopology('edge_table', 0.001, 'the_geom', 'id', 'source', 'target', rows_where := 'true', clean := f)
NOTICE: Performing checks, please wait ....
NOTICE: Creating Topology, Please wait...
NOTICE: -----> TOPOLOGY CREATED FOR 6 edges
NOTICE: Rows with NULL geometry or NULL id: 0
NOTICE: Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE: -----
pgr_createtopology
-----
OK
(1 row)
```

To get the same analysis results as the topology of edge\_table\_noded, we do the following query:

```
SELECT pgr_analyzegraph('edge_table', 0.001, rows_where:= 'id not in (select old_id from edge_table where old_id is not null)');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id not in (select old_id from edge_table where old_id is not null)')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 6
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 0
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)
```

To get the same analysis results as the original edge\_table, we do the following query:

```

SELECT pgr_analyzegraph('edge_table', 0.001, rows_where:='old_id is null');
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','old_id is null')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 2
NOTICE: Dead ends: 7
NOTICE: Potential gaps found near dead ends: 1
NOTICE: Intersections detected: 1
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

Or we can analyze everything because, maybe edge 18 is an overpass, edge 14 is an under pass and there is also a street level junction, and the same happens with edges 17 and 13.

```

SELECT pgr_analyzegraph('edge_table', 0.001);
NOTICE: PROCESSING:
NOTICE: pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE: Performing checks, please wait ...
NOTICE: Analyzing for dead ends. Please wait...
NOTICE: Analyzing for gaps. Please wait...
NOTICE: Analyzing for isolated edges. Please wait...
NOTICE: Analyzing for ring geometries. Please wait...
NOTICE: Analyzing for intersections. Please wait...
NOTICE: ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE: Isolated segments: 0
NOTICE: Dead ends: 3
NOTICE: Potential gaps found near dead ends: 0
NOTICE: Intersections detected: 5
NOTICE: Ring geometries: 0
pgr_analyzegraph
-----
OK
(1 row)

```

See Also

**Topology - Family of Functions** for an overview of a topology for routing algorithms. **pgr\_analyzeOneWay** to analyze directionality of the edges. **pgr\_createTopology** to create a topology based on the geometry. **pgr\_analyzeGraph** to analyze the edges and vertices of the edge table.

## Indices and tables

- [Index](#)
- [Search Page](#)

See Also

## Indices and tables

- [Index](#)
- [Search Page](#)

## Traveling Sales Person - Family of functions

- **pgr\_TSP** - When input is given as matrix cell information.
- **pgr\_TSPeuclidean** - When input are coordinates.

**pgr\_TSP**

- **pgr\_TSP** - Using *Simulated Annealing* approximation algorithm

## Availability: 2.0.0

- Version 2.3.0
  - Signature change
    - Old signature no longer supported
- Version 2.0.0
  - **Official** function

## Support

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

## Description

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question:

*Given a list of cities and the distances between each pair of cities, which is the shortest possible route that visits each city exactly once and returns to the origin city?*

See **Simulated Annealing Algorithm** for a complete description of this implementation

## Signatures

## Summary

```
pgr_TSP(Matrix SQL,  
  [start_id], [end_id],  
  [max_processing_time],  
  [tries_per_temperature], [max_changes_per_temperature], [max_consecutive_non_changes],  
  [initial_temperature], [final_temperature], [cooling_factor],  
  [randomize])  
RETURNS SETOF (seq, node, cost, agg_cost)
```

## Example:

Not having a random execution

```
SELECT * FROM pgr_TSP(  
  $$  
  SELECT * FROM pgr_dijkstraCostMatrix(  
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',  
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 14),  
    directed := false)  
  $$,  
  randomize := false);  
seq | node | cost | agg_cost  
-----  
1 | 1 | 1 | 0  
2 | 2 | 1 | 1  
3 | 3 | 1 | 2  
4 | 4 | 1 | 3  
5 | 9 | 1 | 4  
6 | 6 | 1 | 5  
7 | 11 | 1 | 6  
8 | 12 | 2 | 7  
9 | 10 | 1 | 9  
10 | 13 | 4 | 10  
11 | 7 | 1 | 14  
12 | 8 | 1 | 15  
13 | 5 | 2 | 16  
14 | 1 | 0 | 18  
(14 rows)
```

## Parameters

Parameter	Description
<b>Matrix SQL</b>	an SQL query, described in the <b>Inner query</b>

## Optional Parameters

Parameter	Type	Default	Description
<b>start_vid</b>	BIGINT	0	The greedy part of the implementation will use this identifier.
<b>end_vid</b>	BIGINT	0	Last visiting vertex before returning to start_vid.
<b>max_processing_time</b>	FLOAT	+infinity	Stop the annealing processing when the value is reached.
<b>tries_per_temperature</b>	INTEGER	500	Maximum number of times a neighbor(s) is searched in each temperature.
<b>max_changes_per_temperature</b>	INTEGER	60	Maximum number of times the solution is changed in each temperature.
<b>max_consecutive_non_changes</b>	INTEGER	100	Maximum number of consecutive times the solution is not changed in each temperature.
<b>initial_temperature</b>	FLOAT	100	Starting temperature.
<b>final_temperature</b>	FLOAT	0.1	Ending temperature.
<b>cooling_factor</b>	FLOAT	0.9	Value between between 0 and 1 (not including) used to calculate the next temperature.

Parameter	Type	Default	Description
<b>randomize</b>	BOOLEAN	true	Choose the random seed <ul style="list-style-type: none"> <li>• true: Use current time as seed</li> <li>• false: Use <i>I</i> as seed. Using this value will get the same results with the same data in each execution.</li> </ul>

Inner query

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

Column	Type	Description
<b>start_vid</b>	BIGINT	Identifier of the starting vertex.
<b>end_vid</b>	BIGINT	Identifier of the ending vertex.
<b>agg_cost</b>	FLOAT	Cost for going from start_vid to end_vid

Can be Used with **Cost Matrix - Category** functions with *directed := false*.

If using *directed := true*, the resulting non symmetric matrix must be converted to symmetric by fixing the non symmetric values according to your application needs.

Result Columns

Returns SET OF (seq, node, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INTEGER	Row sequence.
<b>node</b>	BIGINT	Identifier of the node/coordinate/point.
<b>cost</b>	FLOAT	Cost to traverse from the current <code>node</code> to the next <code>node</code> in the path sequence. <ul style="list-style-type: none"> <li>• 0 for the last row in the path sequence.</li> </ul>
<b>agg_cost</b>	FLOAT	Aggregate cost from the <code>node</code> at <code>seq = 1</code> to the current node. <ul style="list-style-type: none"> <li>• 0 for the first row in the path sequence.</li> </ul>

Additional Examples

**Example:**

Start from vertex \ (7)

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_dijkstraCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 14),
    directed := false
  )
  $$,
  start_id := 7,
  randomize := false
);
seq | node | cost | agg_cost
-----+-----+-----+-----
 1 |  7 |  1 |    0
 2 |  8 |  1 |    1
 3 |  5 |  1 |    2
 4 |  2 |  1 |    3
 5 |  1 |  2 |    4
 6 |  3 |  1 |    6
 7 |  4 |  1 |    7
 8 |  9 |  1 |    8
 9 | 12 |  1 |    9
10 | 11 |  1 |   10
11 | 10 |  1 |   11
12 | 13 |  3 |   12
13 |  6 |  3 |   15
14 |  7 |  0 |   18
(14 rows)
```

**Example:**

Using with points of interest.

To generate a symmetric matrix:

- the **side** information of pointsOfInterest is ignored by not including it in the query
- and **directed := false**



```

SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_withPointsCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction from pointsOfInterest',
    array[-1, 3, 5, 6, -6], directed := false)
  $$,
  start_id := 5,
  randomize := false
);
seq | node | cost | agg_cost
-----+-----+-----+-----
 1 |  5 | 0.3 |    0
 2 | -6 | 1.3 |   0.3
 3 | -1 | 1.6 |   1.6
 4 |  3 |  1 |   3.2
 5 |  6 |  1 |   4.2
 6 |  5 |  0 |   5.2
(6 rows)

```

The queries use the **Sample Data** network.

See Also

- [Traveling Sales Person - Family of functions](#)
- [Wikipedia: Traveling Salesman Problem](#)
- [Wikipedia: Simulated annealing](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

**pgr\_TSPeuclidean**

`pgr_TSPeuclidean` - Using *Simulated Annealing* approximation algorithm

## Availability

- Version 3.0.0
  - Name change from `pgr_euclediantSP`
- Version 2.3.0
  - New **Official** function

## Support

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

## Description

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question:

*Given a list of cities and the distances between each pair of cities, which is the shortest possible route that visits each city exactly once and returns to the origin city?*

See **Simulated Annealing Algorithm** for a complete description of this implementation

## Signatures

## Summary

```

pgr_TSPeuclidean(Coordinates SQL,
  [start_id], [end_id],
  [max_processing_time],
  [tries_per_temperature], [max_changes_per_temperature], [max_consecutive_non_changes],
  [initial_temperature], [final_temperature], [cooling_factor],
  [randomize])
RETURNS SETOF (seq, node, cost, agg_cost)

```

## Example:

Not having a random execution

```

SELECT * FROM pgr_TSPeucclidean(
$$
SELECT id, st_X(the_geom) AS x, st_Y(the_geom) AS y FROM edge_table_vertices_pgr
$$,
randomize := false);
seq | node | cost | agg_cost
-----+-----+-----+-----
1 | 1 | 1.41421356237 | 0
2 | 3 | 1 | 1.41421356237
3 | 4 | 1 | 2.41421356237
4 | 9 | 0.583095189485 | 3.41421356237
5 | 16 | 0.583095189485 | 3.99730875186
6 | 6 | 1 | 4.58040394134
7 | 11 | 1 | 5.58040394134
8 | 12 | 1.11803398875 | 6.58040394134
9 | 17 | 1.5 | 7.69843793009
10 | 13 | 0.5 | 9.19843793009
11 | 15 | 0.5 | 9.69843793009
12 | 10 | 1.58113883008 | 10.1984379301
13 | 14 | 1.58113883008 | 11.7795767602
14 | 7 | 1 | 13.3607155903
15 | 8 | 1 | 14.3607155903
16 | 5 | 1 | 15.3607155903
17 | 2 | 1 | 16.3607155903
18 | 1 | 0 | 17.3607155903
(18 rows)

```

Parameters

Parameter	Description
<b>Coordinates SQL</b>	an SQL query, described in the <b>Inner query</b>

Optional Parameters

Parameter	Type	Default	Description
<b>start_vid</b>	BIGINT	0	The greedy part of the implementation will use this identifier.
<b>end_vid</b>	BIGINT	0	Last visiting vertex before returning to start_vid.
<b>max_processing_time</b>	FLOAT	+infinity	Stop the annealing processing when the value is reached.
<b>tries_per_temperature</b>	INTEGER	500	Maximum number of times a neighbor(s) is searched in each temperature.
<b>max_changes_per_temperature</b>	INTEGER	60	Maximum number of times the solution is changed in each temperature.
<b>max_consecutive_non_changes</b>	INTEGER	100	Maximum number of consecutive times the solution is not changed in each temperature.
<b>initial_temperature</b>	FLOAT	100	Starting temperature.
<b>final_temperature</b>	FLOAT	0.1	Ending temperature.
<b>cooling_factor</b>	FLOAT	0.9	Value between between 0 and 1 (not including) used to calculate the next temperature.
<b>randomize</b>	BOOLEAN	true	Choose the random seed <ul style="list-style-type: none"> <li>true: Use current time as seed</li> <li>false: Use 1 as seed. Using this value will get the same results with the same data in each execution.</li> </ul>

Inner query

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

Column	Type	Description
<b>id</b>	BIGINT	(optional) Identifier of the coordinate. <ul style="list-style-type: none"> <li>When missing the coordinates will receive an <b>id</b> starting from 1, in the order given.</li> </ul>
<b>x</b>	FLOAT	X value of the coordinate.
<b>y</b>	FLOAT	Y value of the coordinate.

Result Columns

Returns SET OF (seq, node, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INTEGER	Row sequence.
<b>node</b>	BIGINT	Identifier of the node/coordinate/point.
<b>cost</b>	FLOAT	Cost to traverse from the current <b>node</b> to the next <b>node</b> in the path sequence. <ul style="list-style-type: none"> <li>0 for the last row in the path sequence.</li> </ul>

Column	Type	Description
<b>agg_cost</b>	FLOAT	Aggregate cost from the <code>node</code> at <code>seq = 1</code> to the current node. <ul style="list-style-type: none"> <li>0 for the first row in the path sequence.</li> </ul>

#### Additional Examples

##### Example:

Try  $\backslash(3)$  times per temperature with cooling factor of  $(0.5)$ , not having a random execution

```
SELECT* from pgr_TSPeuclidean(
$$
SELECT id, st_X(the_geom) AS x, st_Y(the_geom) AS y FROM edge_table_vertices_pgr
$$,
tries_per_temperature := 3,
cooling_factor := 0.5,
randomize := false);
seq | node | cost | agg_cost
-----+-----+-----+-----
1 | 1 | 1.41421356237 | 0
2 | 3 | 1 | 1.41421356237
3 | 4 | 1 | 2.41421356237
4 | 9 | 0.583095189485 | 3.41421356237
5 | 16 | 0.583095189485 | 3.99730875186
6 | 6 | 1 | 4.58040394134
7 | 5 | 1 | 5.58040394134
8 | 8 | 1 | 6.58040394134
9 | 7 | 1.58113883008 | 7.58040394134
10 | 14 | 1.5 | 9.16154277143
11 | 15 | 0.5 | 10.6615427714
12 | 13 | 1.5 | 11.1615427714
13 | 17 | 1.11803398875 | 12.6615427714
14 | 12 | 1 | 13.7795767602
15 | 11 | 1 | 14.7795767602
16 | 10 | 2 | 15.7795767602
17 | 2 | 1 | 17.7795767602
18 | 1 | 0 | 18.7795767602
(18 rows)
```

##### Example:

Skipping the Simulated Annealing & showing some process information

```
SET client_min_messages TO DEBUG1;
SET
SELECT* from pgr_TSPeuclidean(
$$
SELECT id, st_X(the_geom) AS x, st_Y(the_geom) AS y FROM edge_table_vertices_pgr
$$,
tries_per_temperature := 0,
randomize := false);
DEBUG: Processing Information
Initializing tsp class ---> tsp.greedyInitial ---> tsp.annealing ---> OK

Cycle(100) total changes =0 0 were because delta energy < 0
Total swaps: 3
Total slides: 0
Total reverses: 0
Times best tour changed: 4
Best cost reached = 18.7796
seq | node | cost | agg_cost
-----+-----+-----+-----
1 | 1 | 1.41421356237 | 0
2 | 3 | 1 | 1.41421356237
3 | 4 | 1 | 2.41421356237
4 | 9 | 0.583095189485 | 3.41421356237
5 | 16 | 0.583095189485 | 3.99730875186
6 | 6 | 1 | 4.58040394134
7 | 5 | 1 | 5.58040394134
8 | 8 | 1 | 6.58040394134
9 | 7 | 1.58113883008 | 7.58040394134
10 | 14 | 1.5 | 9.16154277143
11 | 15 | 0.5 | 10.6615427714
12 | 13 | 1.5 | 11.1615427714
13 | 17 | 1.11803398875 | 12.6615427714
14 | 12 | 1 | 13.7795767602
15 | 11 | 1 | 14.7795767602
16 | 10 | 2 | 15.7795767602
17 | 2 | 1 | 17.7795767602
18 | 1 | 0 | 18.7795767602
(18 rows)
```

The queries use the **Sample Data** network.

See Also

- **Traveling Sales Person - Family of functions**
- **Wikipedia: Traveling Salesman Problem**

- [Wikipedia: Simulated annealing](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

## Previous versions of this page

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

## Table of Contents

- **General Information**
  - **Problem Definition**
  - **Origin**
  - **Characteristics**
- **Simulated Annealing Algorithm**
  - **pgRouting Implementation**
  - **Choosing parameters**
  - **Description of the Control Parameters**
- **Description of the return columns**
  - **See Also**

### General Information

### Problem Definition

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question:

*Given a list of cities and the distances between each pair of cities, which is the shortest possible route that visits each city exactly once and returns to the origin city?*

### Origin

The traveling sales person problem was studied in the 18th century by mathematicians

**Sir William Rowan Hamilton** and **Thomas Penyngton Kirkman**.

A discussion about the work of Hamilton & Kirkman can be found in the book **Graph Theory (Biggs et al. 1976)**.

- ISBN-13: 978-0198539162
- ISBN-10: 0198539169

It is believed that the general form of the TSP have been first studied by Kalr Menger in Vienna and Harvard. The problem was later promoted by Hassler, Whitney & Merrill at Princeton. A detailed description about the connection between Menger & Whitney, and the development of the TSP can be found in **On the history of combinatorial optimization (till 1960)**

### Characteristics

- The travel costs are symmetric:
  - traveling costs from city A to city B are just as much as traveling from B to A.
- This problem is an NP-hard optimization problem.
- To calculate the number of different tours through  $n$  cities:
  - Given a starting city,
  - There are  $(n-1)$  choices for the second city,
  - And  $(n-2)$  choices for the third city, etc.
  - Multiplying these together we get  $(n-1)! = (n-1) (n-2) \dots 1$ .
  - Now since our travel costs do not depend on the direction we take around the tour:
    - this number by 2
    - $((n-1)!/2)$ .

### Simulated Annealing Algorithm

The simulated annealing algorithm was originally inspired from the process of annealing in metal work.

Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties.

### Pseudocode

Given an initial solution, the simulated annealing process, will start with a high temperature and gradually cool down until the desired temperature is reached.

For each temperature, a neighbouring new solution **newSolution** is calculated. The higher the temperature the higher the probability of accepting the new solution as a possible better solution.

Once the desired temperature is reached, the best solution found is returned

```
Solution = initial_solution;

temperature = initial_temperature;
while (temperature > final_temperature) {

    do tries_per_temperature times {
        newSolution = neighbour(solution);
        If P(E(solution), E(newSolution), T) >= random(0, 1)
            solution = newSolution;
    }

    temperature = temperature * cooling_factor;
}

Output: the best solution
```

### pgRouting Implementation

pgRouting's implementation adds some extra parameters to allow some exit controls within the simulated annealing process.

- **max\_changes\_per\_temperature:**
  - Limits the number of changes in the solution per temperature
  - Count is reset to 0 when **temperature** changes
  - Count is increased by 1 when **solution** changes
- **max\_consecutive\_non\_changes:**
  - Limits the number of consecutive non changes per temperature
  - Count is reset to 0 when **solution** changes
  - Count is increased by 1 when **solution** changes
- **max\_processing\_time:**
  - Limits the time the simulated annealing is performed.

```
Solution = initial_solution;

temperature = initial_temperature;
WHILE (temperature > final_temperature) {

    DO tries_per_temperature times {
        newSolution = neighbour(solution);
        If Probability(E(solution), E(newSolution), T) >= random(0, 1)
            solution = newSolution;

        BREAK DO WHEN:
            max_changes_per_temperature is reached
            OR max_consecutive_non_changes is reached
    }

    temperature = temperature * cooling_factor;
    BREAK WHILE WHEN:
        no changes were done in the current temperature
        OR max_processing_time has being reached
}

Output: the best solution found
```

### Choosing parameters

There is no exact rule on how the parameters have to be chose, it will depend on the special characteristics of the problem.

- If the computational time is crucial, then limit execution time with **max\_processing\_time**.
- Make the **tries\_per\_tempture** depending on the number of cities  $(n)$ , for example:
  - Useful to estimate the time it takes to do one cycle: use 1
    - this will help to set a reasonable **max\_processing\_time**
  - $(n * (n-1))$
  - $(500 * n)$
- For a faster decreasing the temperature set **cooling\_factor** to a smaller number, and set to a higher number for a slower decrease.
- When for the same given data the same results are needed, set **randomize** to *false*.
  - When estimating how long it takes to do one cycle: use *false*

A recommendation is to play with the values and see what fits to the particular data.

### Description of the Control Parameters

The control parameters are optional, and have a default value.

Parameter	Type	Default	Description
<b>start_vid</b>	BIGINT	0	The greedy part of the implementation will use this identifier.
<b>end_vid</b>	BIGINT	0	Last visiting vertex before returning to start_vid.
<b>max_processing_time</b>	FLOAT	+infinity	Stop the annealing processing when the value is reached.
<b>tries_per_temperature</b>	INTEGER	500	Maximum number of times a neighbor(s) is searched in each temperature.
<b>max_changes_per_temperature</b>	INTEGER	60	Maximum number of times the solution is changed in each temperature.
<b>max_consecutive_non_changes</b>	INTEGER	100	Maximum number of consecutive times the solution is not changed in each temperature.
<b>initial_temperature</b>	FLOAT	100	Starting temperature.
<b>final_temperature</b>	FLOAT	0.1	Ending temperature.
<b>cooling_factor</b>	FLOAT	0.9	Value between between 0 and 1 (not including) used to calculate the next temperature.
<b>randomize</b>	BOOLEAN	true	Choose the random seed <ul style="list-style-type: none"> <li>• true: Use current time as seed</li> <li>• false: Use 1 as seed. Using this value will get the same results with the same data in each execution.</li> </ul>

#### Description of the return columns

Returns SET OF (seq, node, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INTEGER	Row sequence.
<b>node</b>	BIGINT	Identifier of the node/coordinate/point.
<b>cost</b>	FLOAT	Cost to traverse from the current <code>node</code> to the next <code>node</code> in the path sequence. <ul style="list-style-type: none"> <li>• 0 for the last row in the path sequence.</li> </ul>
<b>agg_cost</b>	FLOAT	Aggregate cost from the <code>node</code> at <code>seq = 1</code> to the current node. <ul style="list-style-type: none"> <li>• 0 for the first row in the path sequence.</li> </ul>

#### See Also

#### References

- [Wikipedia: Traveling Salesman Problem](#)
- [Wikipedia: Simulated annealing](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### Spanning Tree - Category

- [Kruskal - Family of functions](#)
- [Prim - Family of functions](#)

A spanning tree of an undirected graph is a tree that includes all the vertices of G with the minimum possible number of edges.

For a disconnected graph, there there is no single tree, but a spanning forest, consisting of a spanning tree of each connected component.

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

#### See Also

- [Boost: Prim's algorithm documentation](#)
- [Wikipedia: Prim's algorithm](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

## K shortest paths - Category

- **pgr\_KSP** - Yen's algorithm based on pgr\_dijkstra

## Proposed



### Warning

Proposed functions for next mayor release.

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

- **pgr\_withPointsKSP - Proposed** - Yen's algorithm based on pgr\_withPoints

## Previous versions of this page

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

## Indices and tables

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

## pgr\_trsp - Turn Restriction Shortest Path (TRSP)

`pgr_trsp` — Returns the shortest path with support for turn restrictions.

## Availability

- Version 2.1.0
  - New *Via* **prototypes**
    - `pgr_trspViaVertices`
    - `pgr_trspViaEdges`
- Version 2.0.0
  - **Official** function

## Support

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

## Description

The turn restricted shortest 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 an 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` comma 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 edge.

Returns set of:

#### seq:

row sequence

#### id1:

route ID

#### id2:

node ID

#### id3:

edge ID (-1 for the last row)

#### cost:

cost to traverse from `id2` using `id3`

#### Additional Examples

**Example:**

Without turn restrictions

```
SELECT * FROM pgr_trsp(
  'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
  7, 12, false, false
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 7 | 6 | 1
1 | 8 | 7 | 1
2 | 5 | 8 | 1
3 | 6 | 9 | 1
4 | 9 | 15 | 1
5 | 12 | -1 | 0
(6 rows)
```

**Example:**

With turn restrictions

Then a query with turn restrictions is created as:

```
SELECT * FROM pgr_trsp(
  'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
  2, 7, false, false,
  'SELECT to_cost, target_id::int4,
  from_edge || coalesce(", " || via_path, "") AS via_path
  FROM restrictions'
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 2 | 4 | 1
1 | 5 | 10 | 1
2 | 10 | 12 | 1
3 | 11 | 11 | 1
4 | 6 | 8 | 1
5 | 5 | 7 | 1
6 | 8 | 6 | 1
7 | 7 | -1 | 0
(8 rows)
```

```
SELECT * FROM pgr_trsp(
  'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
  7, 11, false, false,
  'SELECT to_cost, target_id::int4,
  from_edge || coalesce(", " || via_path, "") AS via_path
  FROM restrictions'
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 7 | 6 | 1
1 | 8 | 7 | 1
2 | 5 | 8 | 1
3 | 6 | 9 | 1
4 | 9 | 15 | 1
5 | 12 | 13 | 1
6 | 11 | -1 | 0
(7 rows)
```

An example query using vertex ids and via points:

```
SELECT * FROM pgr_trspViaVertices(
  'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
  ARRAY[2,7,11]::INTEGER[],
  false, false,
  'SELECT to_cost, target_id::int4, from_edge ||
  coalesce(", " || via_path, "") AS via_path FROM restrictions');
seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+-----
1 | 1 | 2 | 4 | 1
2 | 1 | 5 | 10 | 1
3 | 1 | 10 | 12 | 1
4 | 1 | 11 | 11 | 1
5 | 1 | 6 | 8 | 1
6 | 1 | 5 | 7 | 1
7 | 1 | 8 | 6 | 1
8 | 2 | 7 | 6 | 1
9 | 2 | 8 | 7 | 1
10 | 2 | 5 | 8 | 1
11 | 2 | 6 | 9 | 1
12 | 2 | 9 | 15 | 1
13 | 2 | 12 | 13 | 1
14 | 2 | 11 | -1 | 0
(14 rows)
```

An example query using edge ids and vias:

```
SELECT * FROM pgr_trspViaEdges(
  'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost,
  reverse_cost FROM edge_table',
  ARRAY[2,7,11)::INTEGER[],
  ARRAY[0.5, 0.5, 0.5)::FLOAT[],
  true,
  true,
  'SELECT to_cost, target_id::int4, FROM_edge ||
  coalesce("",||via_path,"") AS via_path FROM restrictions');
seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+-----
 1 | 1 | -1 | 2 | 0.5
 2 | 1 | 2 | 4 | 1
 3 | 1 | 5 | 8 | 1
 4 | 1 | 6 | 9 | 1
 5 | 1 | 9 | 16 | 1
 6 | 1 | 4 | 3 | 1
 7 | 1 | 3 | 5 | 1
 8 | 1 | 6 | 8 | 1
 9 | 1 | 5 | 7 | 1
10 | 2 | 5 | 8 | 1
11 | 2 | 6 | 9 | 1
12 | 2 | 9 | 16 | 1
13 | 2 | 4 | 3 | 1
14 | 2 | 3 | 5 | 1
15 | 2 | 6 | 11 | 0.5
(15 rows)
```

The queries use the **Sample Data** network.

**Known Issues**

Introduction

pgr\_trsp code has issues that are not being fixed yet, but as time passes and new functionality is added to pgRouting with wrappers to **hide** the issues, not to fix them.

For clarity on the queries:

- \_pgr\_trsp (internal\_function) is the original code
- pgr\_trsp (lower case) represents the wrapper calling the original code
- pgr\_TRSP (upper case) represents the wrapper calling the replacement function, depending on the function, it can be:
  - pgr\_dijkstra
  - pgr\_dijkstraVia
  - pgr\_withPoints
  - \_pgr\_withPointsVia (internal function)

The restrictions

The restriction used in the examples does not have to do anything with the graph:

- No vertex has id: 25, 32 or 33
- No edge has id: 25, 32 or 33

A restriction is assigned as:

```
SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path;
to_cost | target_id | via_path
-----+-----+-----
 100 | 25 | 32, 33
(1 row)
```

The back end code has that same restriction as follows

```
SELECT 1 AS id, 100::float AS cost, 25::INTEGER AS target_id, ARRAY[33, 32, 25] AS path;
id | cost | target_id | path
-----+-----+-----+-----
 1 | 100 | 25 | {33,32,25}
(1 row)
```

therefore the shortest path expected are as if there was no restriction involved

The “Vertices” signature version

```
pgr_trsp(sql text, source integer, target integer,
  directed boolean, has_rcost boolean [, restrict_sql text]);
```

Different ways to represent 'no path found`

- Sometimes represents with EMPTY SET a no path found
- Sometimes represents with Error a no path found

### Returning EMPTY SET to represent no path found

```
SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 15, true, true
);
seq | id1 | id2 | cost
-----+-----+-----+-----
(0 rows)
```

pgr\_trsp calls **pgr\_dijkstra** when there are no restrictions which returns *EMPTY SET* when a path is not found

```
SELECT * FROM pgr_dijkstra(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 15
);
seq | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
(0 rows)
```

### Throwing EXCEPTION to represent no path found

```
SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 15, true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR: Error computing path: Path Not Found
```

pgr\_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, which will throw an EXCEPTION to represent no path found.

Routing from/to same location

When routing from location  $\{(1)\}$  to the same location  $\{(1)\}$ , no path is needed to reach the destination, its already there. Therefore is expected to return an *EMPTY SET* or an *EXCEPTION* depending on the parameters

- Sometimes represents with EMPTY SET no path found (expected)
- Sometimes represents with EXCEPTION no path found (expected)
- Sometimes finds a path (not expected)

### Returning expected EMPTY SET to represent no path found

```
SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 1, true, true
);
seq | id1 | id2 | cost
-----+-----+-----+-----
(0 rows)
```

pgr\_trsp calls **pgr\_dijkstra** when there are no restrictions which returns the expected to return *EMPTY SET* to represent no path found.

### Returning expected EXCEPTION to represent no path found

```
SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  14, 14, true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR: Error computing path: Path Not Found
```

In this case pgr\_trsp calls the original code when there are restrictions, even if they have nothing to do with the graph, in this case that code throws the expected EXCEPTION

### Returning unexpected path

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 1, true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 1 | 1 | 1
1 | 2 | 4 | 1
2 | 5 | 8 | 1
3 | 6 | 9 | 1
4 | 9 | 16 | 1
5 | 4 | 3 | 1
6 | 3 | 2 | 1
7 | 2 | 1 | 1
8 | 1 | -1 | 0
(9 rows)

```

In this case `pgr_trsp` calls the original code when there are restrictions, even if they have nothing to do with the graph, in this case that code finds an unexpected path.

User contradictions

`pgr_trsp` unlike other pgRouting functions does not autodetect the existence of `reverse_cost` column. Therefore it has `has_rcost` parameter to check the existence of `reverse_cost` column. Contradictions happen:

- When the `reverse_cost` is missing, and the flag `has_rcost` is set to true
- When the `reverse_cost` exists, and the flag `has_rcost` is set to false

### When the `reverse_cost` is missing, and the flag `has_rcost` is set to true.

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table$$,
  2, 3, false, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR: Error, reverse_cost is used, but query didn't return 'reverse_cost' column

```

An EXCEPTION is thrown.

### When the `reverse_cost` exists, and the flag `has_rcost` is set to false

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  2, 3, false, false,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 2 | 4 | 1
1 | 5 | 8 | 1
2 | 6 | 5 | 1
3 | 3 | -1 | 0
(4 rows)

```

The `reverse_cost` column will be effectively removed and will cost execution time

The “Edges” signature version

```

pgr_trsp(sql text, source_edge integer, source_pos float8,
  target_edge integer, target_pos float8,
  directed boolean, has_rcost boolean [,restrict_sql text]);

```

Different ways to represent ‘no path found’

- Sometimes represents with EMPTY SET a no path found
- Sometimes represents with EXCEPTION a no path found

### Returning EMPTY SET to represent no path found

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 17, 0.5, true, true
);
seq | id1 | id2 | cost
-----+-----+-----+-----
(0 rows)

```

pgr\_trsp calls **pgr\_withPoints - Proposed** when there are no restrictions which returns *EMPTY SET* when a path is not found

### Throwing **EXCEPTION** to represent no path found

```
SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 17, 0.5, true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR: Error computing path: Path Not Found
```

pgr\_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, which will throw an **EXCEPTION** to represent no path found.

Paths with equal number of vertices and edges

A path is made of  $N$  vertices and  $N - 1$  edges.

- Sometimes returns  $N$  vertices and  $N - 1$  edges.
- Sometimes returns  $N - 1$  vertices and  $N - 1$  edges.

### Returning $N$ vertices and $N - 1$ edges.

```
SELECT * FROM pgr_TRSP(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 1, 0.8, true, true
);
seq | id1 | id2 | cost
-----+-----+-----+-----
  0 | -1 |  1 | 0.3
  1 | -2 | -1 |  0
(2 rows)
```

pgr\_trsp calls **pgr\_withPoints - Proposed** when there are no restrictions which returns the correct number of rows that will include all the vertices. The last row will have a **-1** on the edge column to indicate the edge number is invalidu for that row.

### Returning $N - 1$ vertices and $N - 1$ edges.

```
SELECT * FROM pgr_TRSP(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 1, 0.8, true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----+-----
  0 | -1 |  1 | 0.3
(1 row)
```

pgr\_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, and will not return the last vertex of the path.

Routing from/to same location

When routing from the same edge and position to the same edge and position, no path is needed to reach the destination, its already there. Therefore is expected to return an *EMPTY SET* or an *EXCEPTION* depending on the parameters, non of which is happening.

### A path with 2 vertices and edge cost 0

```
SELECT * FROM pgr_TRSP(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 1, 0.5, true, true
);
seq | id1 | id2 | cost
-----+-----+-----+-----
  0 | -1 |  1 |  0
  1 | -2 | -1 |  0
(2 rows)
```

pgr\_trsp calls **pgr\_withPoints - Proposed** setting the first  $((edge, position))$  with a differencnt point id from the second  $((edge, position))$  making them different points. But the cost using the edge, is  $(0)$ .

### A path with 1 vertices and edge cost 0

```

SELECT * FROM pgr_TRSP(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 1, 0.5, true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----
0 | -1 | 1 | 0
(1 row)

```

pgr\_trsp use the original code when there are restrictions, even if they have nothing to do with the graph, and will not have the row for the vertex \(-2\).

User contradictions

pgr\_trsp unlike other pgRouting functions does not autodetect the existence of reverse\_cost column. Therefore it has has\_rcost parameter to check the existence of reverse\_cost column. Contradictions happen:

- When the reverse\_cost is missing, and the flag has\_rcost is set to true
- When the reverse\_cost exists, and the flag has\_rcost is set to false

**When the reverse\_cost is missing, and the flag has\_rcost is set to true.**

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table$$,
  1, 0.5, 1, 0.8, false, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
ERROR: Error, reverse_cost is used, but query didn't return 'reverse_cost' column

```

An EXCEPTION is thrown.

**When the reverse\_cost exists, and the flag has\_rcost is set to false**

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  1, 0.5, 1, 0.8, false, false,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----
0 | -1 | 1 | 0.3
(1 row)

```

The reverse\_cost column will be effectively removed and will cost execution time

Using a points of interest table

Given a set of points of interest:

```

SELECT * FROM pointsOfInterest;
pid | x | y | edge_id | side | fraction | the_geom | newpoint
-----+-----+-----+-----+-----+-----+-----+-----
1 | 1.8 | 0.4 | 1 | l | 0.4 | 0101000000CDCCCCCCCCFC3F9A999999999D93F | 01010000000000000000000000409A999999999D93F
2 | 4.2 | 2.4 | 15 | r | 0.4 | 0101000000CDCCCCCCCC1040333333333330340 | 0101000000000000000000000104033333333330340
3 | 2.6 | 3.2 | 12 | l | 0.6 | 0101000000GDCCCCCCCC04409A99999999990940 | 0101000000CDCCCCCCCC0440000000000000840
4 | 0.3 | 1.8 | 6 | r | 0.3 | 0101000000333333333333D33FCDCCCCCCCCFC3F | 0101000000333333333333D33F0000000000000040
5 | 2.9 | 1.8 | 5 | l | 0.8 | 01010000003333333333330740CDCCCCCCCCFC3F | 01010000000000000000000000840CDCCCCCCCCFC3F
6 | 2.2 | 1.7 | 4 | b | 0.7 | 01010000009A999999999014033333333333FB3F | 01010000000000000000000000403333333333FB3F
(6 rows)

```

Using pgr\_trsp

```

SELECT * FROM pgr_TRSP(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  (SELECT edge_id::INTEGER FROM pointsOfInterest WHERE pid = 1),
  (SELECT fraction FROM pointsOfInterest WHERE pid = 1),
  (SELECT edge_id::INTEGER FROM pointsOfInterest WHERE pid = 6),
  (SELECT fraction FROM pointsOfInterest WHERE pid = 6),
  true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | -1 | 1 | 0.6
1 | 2 | 4 | 0.7
(2 rows)

```

On `pgr_trsp`, to be able to use the table information:

- Each parameter has to be extracted explicitly from the table
- Regardless of the point pid original value
  - will always be -1 for the first point
  - will always be -2 for the second point
    - the row reaching point -2 will not be shown

### Using `pgr_withPoints` - Proposed

```

SELECT * FROM pgr_withPoints(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  $$SELECT pid, edge_id, fraction FROM pointsOfInterest$$,
  -1, -6
);
seq | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
1 | 1 | -1 | 1 | 0.6 | 0
2 | 2 | 2 | 4 | 0.7 | 0.6
3 | 3 | -6 | -1 | 0 | 1.3
(3 rows)

```

Suggestion: use `pgr_withPoints` - Proposed when there are no turn restrictions:

- Results are more complete
- Column names are meaningful

Routing from a vertex to a point

Solving a shortest path from vertex\{6\} to pid 1 using a points of interest table

### Using `pgr_trsp`

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  8, 1,
  (SELECT edge_id::INTEGER FROM pointsOfInterest WHERE pid = 1),
  (SELECT fraction FROM pointsOfInterest WHERE pid = 1),
  true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 6 | 8 | 1
1 | 5 | 4 | 1
2 | 2 | 1 | 0.6
(3 rows)

```

- Vertex 6 is on edge 8 at 1 fraction

```

SELECT * FROM pgr_trsp(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  11, 0,
  (SELECT edge_id::INTEGER FROM pointsOfInterest WHERE pid = 1),
  (SELECT fraction FROM pointsOfInterest WHERE pid = 1),
  true, true,
  $$SELECT 100::float AS to_cost, 25::INTEGER AS target_id, '32, 33'::TEXT AS via_path$$
);
seq | id1 | id2 | cost
-----+-----+-----+-----
0 | 6 | 8 | 1
1 | 5 | 4 | 1
2 | 2 | 1 | 0.6
(3 rows)

```



- Vertex 6 is also edge 11 at 0 fraction

### Using **pgr\_withPoints - Proposed**

```
SELECT * FROM pgr_withPoints(
  $$SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table$$,
  $$SELECT pid, edge_id, fraction FROM pointsOfInterest$$,
  6, -1
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	6	8	1	0
2	2	5	4	1	1
3	3	2	1	0.6	2
4	4	-1	-1	0	2.6

(4 rows)

Suggestion: use **pgr\_withPoints - Proposed** when there are no turn restrictions:

- No need to choose where the vertex is located.
- Results are more complete
- Column names are meaningful

prototypes

`pgr_trspViaVertices` and `pgr_trspViaEdges` were added to `pgRouting` as prototypes

These functions use the `pgr_trsp` functions inheriting all the problems mentioned above. When there are no restrictions and have a routing “via” problem with vertices:

- pgr\_dijkstraVia - Proposed**

See Also


### Indices and tables

- [Index](#)
- [Search Page](#)

### Cost - Category

- pgr\_aStarCost**
- pgr\_dijkstraCost**

### Proposed



**Warning**

Proposed functions for next mayor release.

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

- pgr\_withPointsCost - Proposed**

### Previous versions of this page

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

### General Information

Characteristics

The main Characteristics are:

- Each function works as part of the family it belongs to.

- It does not return a path.
- Returns the sum of the costs of the resulting path(s) for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of  $(start\_vid, end\_vid, agg\_cost)$ .
  - When the starting vertex and ending vertex are the same, there is no path.
    - The  $agg\_cost$  in the non included values  $(v, v)$  is 0.
  - When the starting vertex and ending vertex are the different and there is no path.
    - The  $agg\_cost$  in the non included values  $(u, v)$  is  $(-\infty)$ .
- Let be the case the values returned are stored in a table, so the unique index would be the pair  $(start\_vid, end\_vid)$ .
- Depending on the function and its parameters, the results can be symmetric.
  - The  $agg\_cost$  of  $(u, v)$  is the same as for  $(v, u)$ .
- Any duplicated value in the  $start\_vids$  or in  $end\_vids$  are ignored.
- The returned values are ordered:
  - $start\_vid$  ascending
  - $end\_vid$  ascending

See Also

## Indices and tables

- [Index](#)
- [Search Page](#)

## Cost Matrix - Category

- [pgr\\_aStarCostMatrix](#)
- [pgr\\_dijkstraCostMatrix](#)

## proposed



### Warning

Proposed functions for next mayor release.

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

## [pgr\\_withPointsCostMatrix](#) - proposed

### [pgr\\_withPointsCostMatrix](#) - proposed

[pgr\\_withPointsCostMatrix](#) - Calculates the shortest path and returns only the aggregate cost of the shortest path(s) found, for the combination of points given.



### Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGGER 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 2.2.0
  - New **proposed** function
- Supported versions:** current(**3.0**)
- Unsupported versions:** **2.6 2.5 2.4 2.3**

## Description

- TBD**

## Signatures

## Summary

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vids [, directed] [, driving_side])  
RETURNS SET OF (start_vid, end_vid, agg_cost)
```



### Note

There is no **details** flag, unlike the other members of the withPoints family of functions.

## Using default

The minimal signature:

- Is for a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vid)  
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

### Example:

Cost matrix for points  $\{\{1, 6\}\}$  and vertices  $\{\{3, 6\}\}$  on a **directed** graph

```
SELECT * FROM pgr_withPointsCostMatrix(  
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',  
  'SELECT pid, edge_id, fraction from pointsOfInterest',  
  array[-1, 3, 6, -6]);  
start_vid | end_vid | agg_cost
```

start_vid	end_vid	agg_cost
-6	-1	1.3
-6	3	4.3
-6	6	1.3
-1	-6	1.3
-1	3	5.6
-1	6	2.6
3	-6	1.7
3	-1	1.6
3	6	1
6	-6	1.3
6	-1	2.6
6	3	3

(12 rows)

## Complete Signature

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vids,  
  directed:=true, driving_side:='b')  
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

### Example:

Cost matrix for points  $\{\{1, 6\}\}$  and vertices  $\{\{3, 6\}\}$  on an **undirected** graph

- Returning a **symmetrical** cost matrix
- Using the default **side** value on the **points\_sql** query
- Using the default **driving\_side** value

```
SELECT * FROM pgr_withPointsCostMatrix(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
'SELECT pid, edge_id, fraction from pointsOfInterest',
array[-1, 3, 6, -6], directed := false);
start_vid | end_vid | agg_cost
```

```
-----+-----+-----
-6 | -1 | 1.3
-6 | 3 | 1.7
-6 | 6 | 1.3
-1 | -6 | 1.3
-1 | 3 | 1.6
-1 | 6 | 2.6
3 | -6 | 1.7
3 | -1 | 1.6
3 | 6 | 1
6 | -6 | 1.3
6 | -1 | 2.6
6 | 3 | 1
```

(12 rows)

## Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	Edges SQL query as described above.
<b>points_sql</b>	TEXT	Points SQL query as described above.
<b>start_vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of starting vertices. When negative: is a point's pid.
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>driving_side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: <ul style="list-style-type: none"> <li>In the right or left or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>

Returns SET OF (start\_vid, end\_vid, agg\_cost)

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

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICALS:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Description of the Points SQL query

### points\_sql:

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

Column	Type	Description
--------	------	-------------

Column	Type	Description
<b>pid</b>	ANY-INTEGER	(optional) Identifier of the point. <ul style="list-style-type: none"> <li>• If column present, it can not be NULL.</li> <li>• If column not present, a sequential identifier will be given automatically.</li> </ul>
<b>edge_id</b>	ANY-INTEGER	Identifier of the “closest” edge to the point.
<b>fraction</b>	ANY-NUMERICAL	Value in <0,1> that indicates the relative position from the first end point of the edge.
<b>side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: <ul style="list-style-type: none"> <li>• In the right, left of the edge or</li> <li>• If it doesn't matter with 'b' or NULL.</li> <li>• If column not present 'b' is considered.</li> </ul>

Where:

**ANY-INTEGER:**

smallint, int, bigint

**ANY-NUMERICAL:**

smallint, int, bigint, real, float

Additional Examples

**Example:**

**pgr\_TSP** using `pgr_withPointsCostMatrix` for points  $\{\{1, 6\}\}$  and vertices  $\{\{3, 6\}\}$  on an **undirected** graph

```
SELECT * FROM pgr_TSP(
  $$
  SELECT * FROM pgr_withPointsCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction from pointsOfInterest',
    array[-1, 3, 6, -6], directed := false);
  $$,
  randomize := false
);
seq | node | cost | agg_cost
-----+-----+-----+-----
 1 | -6 | 1.3 | 0
 2 | -1 | 1.6 | 1.3
 3 | 3 | 1 | 2.9
 4 | 6 | 1.3 | 3.9
 5 | -6 | 0 | 5.2
(5 rows)
```

See Also

- [pgr\\_withPoints - Proposed](#)
- [Cost Matrix - Category](#)
- [pgr\\_TSP](#)
- *sampledata* network.

**Indices and tables**

- [Index](#)
- [Search Page](#)

**Previous versions of this page**

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

**General Information**

Synopsis

**Traveling Sales Person - Family of functions** needs as input a symmetric cost matrix and no edge( $u, v$ ) must value  $\infty$ .

This collection of functions will return a cost matrix in form of a table.

Characteristics

The main Characteristics are:

- Can be used as input to **pgr\_TSP**.



- 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

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

## Description

Returns the polygon part of an alpha shape.

## Characteristics

- Input is a *geometry* and returns a *geometry*
- Uses PostGis ST\_DelaunyTriangles
- Instead of using CGAL's definition of *alpha* it use the `spoon_radius`
  - $\backslash(\text{spoon\_radius} = \backslash\text{sqrt alpha}\backslash)$
- A Triangle area is considered part of the alpha shape when  $\backslash(\text{circumcenter}\backslash \text{radius} < \text{spoon}\backslash\text{radius}\backslash)$
- When the total number of points is less than 3, returns an EMPTY geometry

## Signatures

## Summary

```
pgr_alphaShape(geometry, [spoon_radius])
RETURNS geometry
```

## Example: passing a geometry collection with spoon radius $\backslash(1.5\backslash)$ using the return variable `geom`

```
SELECT ST_Area(pgr_alphaShape((SELECT ST_Collect(the_geom) FROM edge_table_vertices_pgr), 1.5));
st_area
-----
  9.75
(1 row)
```

## Parameters

Parameter	Type	Default	Description
<b>geometry</b>	<code>geometry</code>		Geometry with at least $\backslash(3\backslash)$ points
<b>spoon_radius</b>	<code>FLOAT</code>		The radius of the spoon

## Return Value

Kind of geometry	Description
GEOMETRY	A Geometry collection of Polygons
COLLECTION	

## See Also

- **pgr\_drivingDistance**
- **Sample Data** network.
- **ST\_ConcaveHull**

## Indices and tables

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

## Previous versions of this page

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

See Also

## Indices and tables

- [Index](#)
- [Search Page](#)

See Also

## Indices and tables

- [Index](#)
- [Search Page](#)

### All Pairs - Family of Functions

- [pgr\\_floydWarshall](#) - Floyd-Warshall's algorithm.
- [pgr\\_johnson](#) - Johnson's algorithm

### aStar - Family of functions

- [pgr\\_aStar](#) - A\* algorithm for the shortest path.
- [pgr\\_aStarCost](#) - Get the aggregate cost of the shortest paths.
- [pgr\\_aStarCostMatrix](#) - Get the cost matrix of the shortest paths.

### Bidirectional A\* - Family of functions

- [pgr\\_bdAStar](#) - Bidirectional A\* algorithm for obtaining paths.
- [pgr\\_bdAStarCost](#) - Bidirectional A\* algorithm to calculate the cost of the paths.
- [pgr\\_bdAStarCostMatrix](#) - Bidirectional A\* algorithm to calculate a cost matrix of paths.

### Bidirectional Dijkstra - Family of functions

- [pgr\\_bdDijkstra](#) - Bidirectional Dijkstra algorithm for the shortest paths.
- [pgr\\_bdDijkstraCost](#) - Bidirectional Dijkstra to calculate the cost of the shortest paths
- [pgr\\_bdDijkstraCostMatrix](#) - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

### Components - Family of functions

- [pgr\\_connectedComponents](#) - Connected components of an undirected graph.
- [pgr\\_strongComponents](#) - Strongly connected components of a directed graph.
- [pgr\\_biconnectedComponents](#) - Biconnected components of an undirected graph.
- [pgr\\_articulationPoints](#) - Articulation points of an undirected graph.
- [pgr\\_bridges](#) - Bridges of an undirected graph.

### Contraction - Family of functions

- [pgr\\_contraction](#)

### Dijkstra - Family of functions

- [pgr\\_dijkstra](#) - Dijkstra's algorithm for the shortest paths.
- [pgr\\_dijkstraCost](#) - Get the aggregate cost of the shortest paths.
- [pgr\\_dijkstraCostMatrix](#) - Use [pgr\\_dijkstra](#) to create a costs matrix.
- [pgr\\_drivingDistance](#) - Use [pgr\\_dijkstra](#) to calculate catchment information.
- [pgr\\_KSP](#) - Use Yen algorithm with [pgr\\_dijkstra](#) to get the K shortest paths.

### Flow - Family of functions

- [pgr\\_maxFlow](#) - Only the Max flow calculation using Push and Relabel algorithm.
- [pgr\\_boykovKolmogorov](#) - Boykov and Kolmogorov with details of flow on edges.
- [pgr\\_edmondsKarp](#) - Edmonds and Karp algorithm with details of flow on edges.
- [pgr\\_pushRelabel](#) - Push and relabel algorithm with details of flow on edges.
- Applications
  - [pgr\\_edgeDisjointPaths](#) - Calculates edge disjoint paths between two groups of vertices.
  - [pgr\\_maxCardinalityMatch](#) - Calculates a maximum cardinality matching in a graph.

### Kruskal - Family of functions

- [pgr\\_kruskal](#)
- [pgr\\_kruskalBFS](#)
- [pgr\\_kruskalDD](#)
- [pgr\\_kruskalDFS](#)

### Prim - Family of functions



- **pgr\_prim**
- **pgr\_primBFS**
- **pgr\_primDD**
- **pgr\_primDFS**

### Topology - Family of Functions

- **pgr\_createTopology** - to create a topology based on the geometry.
- **pgr\_createVerticesTable** - to reconstruct the vertices table based on the source and target information.
- **pgr\_analyzeGraph** - to analyze the edges and vertices of the edge table.
- **pgr\_analyzeOneWay** - to analyze directionality of the edges.
- **pgr\_nodeNetwork** - to create nodes to a not noded edge table.

### Traveling Sales Person - Family of functions

- **pgr\_TSP** - When input is given as matrix cell information.
- **pgr\_TSPeuclidean** - When input are coordinates.

**pgr\_trsp** - Turn Restriction Shortest Path (TRSP) - Turn Restriction Shortest Path (TRSP)

## Functions by categories

---

### Cost - Category

- **pgr\_aStarCost**
- **pgr\_dijkstraCost**

### Cost Matrix - Category

- **pgr\_aStarCostMatrix**
- **pgr\_dijkstraCostMatrix**

### Driving Distance - Category

- **pgr\_drivingDistance** - Driving Distance based on Dijkstra's algorithm
- **pgr\_primDD** - Driving Distance based on Prim's algorithm
- **pgr\_kruskalDD** - Driving Distance based on Kruskal's algorithm
- Post processing
  - **pgr\_alphaShape** - Alpha shape computation

### K shortest paths - Category

- **pgr\_KSP** - Yen's algorithm based on pgr\_dijkstra

### Spanning Tree - Category

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

## Available Functions but not official pgRouting functions

---

- **Proposed Functions**
- **Experimental Functions**

### Proposed Functions



#### Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGGER 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 sequence of vertices.

#### withPoints - Family of functions

- **pgr\_withPoints - Proposed** - Route from/to points anywhere on the graph.
- **pgr\_withPointsCost - Proposed** - Costs of the shortest paths.
- **pgr\_withPointsCostMatrix - proposed** - Costs of the shortest paths.
- **pgr\_withPointsKSP - Proposed** - K shortest paths.
- **pgr\_withPointsDD - Proposed** - Driving distance.

#### categories

##### Cost - Category

- **pgr\_withPointsCost - Proposed**

##### Cost Matrix - Category

- **pgr\_withPointsCostMatrix - proposed**

##### Driving Distance - Category

- **pgr\_withPointsDD - Proposed** - Driving Distance based on pgr\_withPoints

##### K shortest paths - Category

- **pgr\_withPointsKSP - Proposed** - Yen's algorithm based on pgr\_withPoints

#### withPoints - Family of functions

When points are also given as input:



#### Warning

Proposed functions for next mayor release.

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

- **pgr\_withPoints - Proposed** - Route from/to points anywhere on the graph.
- **pgr\_withPointsCost - Proposed** - Costs of the shortest paths.
- **pgr\_withPointsCostMatrix - proposed** - Costs of the shortest paths.
- **pgr\_withPointsKSP - Proposed** - K shortest paths.
- **pgr\_withPointsDD - Proposed** - Driving distance.

#### pgr\_withPoints - Proposed

`pgr_withPoints` - Returns the shortest path in a graph with additional temporary vertices.



#### Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGGER 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 2.2.0
  - New **proposed** function

### Support

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

### Description

Modify the graph to include points defined by points\_sql. Using Dijkstra algorithm, find the shortest path(s)

### The main characteristics are:

- Process is done only on edges with positive costs.
- Vertices of the graph are:
  - positive** when it belongs to the edges\_sql
  - negative** when it belongs to the points\_sql
- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path. - The agg\_cost the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path: - The agg\_cost the non included values (u, v) is  $\infty$
- 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:  $\mathcal{O}(|start\_vids| \times (V \log V + E))$

### Signatures

### Summary

```
pgr_withPoints(edges_sql, points_sql, from_vid, to_vid [, directed] [, driving_side] [, details])
pgr_withPoints(edges_sql, points_sql, from_vid, to_vids [, directed] [, driving_side] [, details])
pgr_withPoints(edges_sql, points_sql, from_vids, to_vid [, directed] [, driving_side] [, details])
pgr_withPoints(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, [start_vid,] [end_vid,] node, edge, cost, agg_cost)
```

### Using defaults

```
pgr_withPoints(edges_sql, points_sql, from_vid, to_vid)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

### Example:

From point  $\backslash(1)$  to point  $\backslash(3)$

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of points\_sql query.

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, -3);
seq | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 |    1 |   1 |  -1 |  1 |    0.6 |    0
 2 |    2 |   2 |   2 |  4 |    1 |    0.6
 3 |    3 |   5 |  10 |  1 |    1.6
 4 |    4 |   4 |  10 |  12 |  0.6 |    2.6
 5 |    5 |   5 |  -3 |  -1 |    0 |    3.2
(5 rows)
```

### One to One

```
pgr_withPoints(edges_sql, points_sql, from_vid, to_vid [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

**Example:**

From point \{1\} to vertex \{3\} with details of passing points

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 3,
  details := true);
seq | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 | 1 | -1 | 1 | 0.6 | 0
 2 | 2 | 2 | 4 | 0.7 | 0.6
 3 | 3 | -6 | 4 | 0.3 | 1.3
 4 | 4 | 5 | 8 | 1 | 1.6
 5 | 5 | 6 | 9 | 1 | 2.6
 6 | 6 | 9 | 16 | 1 | 3.6
 7 | 7 | 4 | 3 | 1 | 4.6
 8 | 8 | 3 | -1 | 0 | 5.6
(8 rows)
```

**One to Many**

```
pgr_withPoints(edges_sql, points_sql, from_vid, to_vids [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost)
```

**Example:**

From point \{1\} to point \{3\} and vertex \{5\}

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, ARRAY[-3,5]);
seq | path_seq | end_pid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | -3 | -1 | 1 | 0.6 | 0
 2 | 2 | -3 | 2 | 4 | 1 | 0.6
 3 | 3 | -3 | 5 | 10 | 1 | 1.6
 4 | 4 | -3 | 10 | 12 | 0.6 | 2.6
 5 | 5 | -3 | -3 | -1 | 0 | 3.2
 6 | 1 | 5 | -1 | 1 | 0.6 | 0
 7 | 2 | 5 | 2 | 4 | 1 | 0.6
 8 | 3 | 5 | 5 | -1 | 0 | 1.6
(8 rows)
```

**Many to One**

```
pgr_withPoints(edges_sql, points_sql, from_vids, to_vid [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost)
```

**Example:**

From point \{1\} and vertex \{2\} to point \{3\}

```
SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], -3);
seq | path_seq | start_pid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | -1 | -1 | 1 | 0.6 | 0
 2 | 2 | -1 | 2 | 4 | 1 | 0.6
 3 | 3 | -1 | 5 | 10 | 1 | 1.6
 4 | 4 | -1 | 10 | 12 | 0.6 | 2.6
 5 | 5 | -1 | -3 | -1 | 0 | 3.2
 6 | 1 | 2 | 2 | 4 | 1 | 0
 7 | 2 | 2 | 5 | 10 | 1 | 1
 8 | 3 | 2 | 10 | 12 | 0.6 | 2
 9 | 4 | 2 | -3 | -1 | 0 | 2.6
(9 rows)
```

**Many to Many**

```
pgr_withPoints(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
```

**Example:**

From point \{1\} and vertex \{2\} to point \{3\} and vertex \{7\}

```

SELECT * FROM pgr_withPoints(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], ARRAY[-3,7]);
seq | path_seq | start_pid | end_pid | node | edge | cost | agg_cost

```

```

-----+-----+-----+-----+-----+-----+-----+-----
1 | 1 | -1 | -3 | -1 | 1 | 0.6 | 0
2 | 2 | -1 | -3 | 2 | 4 | 1 | 0.6
3 | 3 | -1 | -3 | 5 | 10 | 1 | 1.6
4 | 4 | -1 | -3 | 10 | 12 | 0.6 | 2.6
5 | 5 | -1 | -3 | -3 | -1 | 0 | 3.2
6 | 1 | -1 | 7 | -1 | 1 | 0.6 | 0
7 | 2 | -1 | 7 | 2 | 4 | 1 | 0.6
8 | 3 | -1 | 7 | 5 | 7 | 1 | 1.6
9 | 4 | -1 | 7 | 8 | 6 | 1 | 2.6
10 | 5 | -1 | 7 | 7 | -1 | 0 | 3.6
11 | 1 | 2 | -3 | 2 | 4 | 1 | 0
12 | 2 | 2 | -3 | 5 | 10 | 1 | 1
13 | 3 | 2 | -3 | 10 | 12 | 0.6 | 2
14 | 4 | 2 | -3 | -3 | -1 | 0 | 2.6
15 | 1 | 2 | 7 | 2 | 4 | 1 | 0
16 | 2 | 2 | 7 | 5 | 7 | 1 | 1
17 | 3 | 2 | 7 | 8 | 6 | 1 | 2
18 | 4 | 2 | 7 | 7 | -1 | 0 | 3
(18 rows)

```

#### Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	Edges SQL query as described above.
<b>points_sql</b>	TEXT	Points SQL query as described above.
<b>start_vid</b>	ANY-INTEGERS	Starting vertex identifier. When negative: is a point's pid.
<b>end_vid</b>	ANY-INTEGERS	Ending vertex identifier. When negative: is a point's pid.
<b>start_vids</b>	ARRAY[ANY-INTEGERS]	Array of identifiers of starting vertices. When negative: is a point's pid.
<b>end_vids</b>	ARRAY[ANY-INTEGERS]	Array of identifiers of ending vertices. When negative: is a point's pid.
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>driving_side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: <ul style="list-style-type: none"> <li>In the right or left or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>
<b>details</b>	BOOLEAN	(optional). When <code>true</code> the results will include the points in <code>points_sql</code> that are in the path. Default is <code>false</code> which ignores other points of the <code>points_sql</code> .

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Description of the Points SQL query

#### points\_sql:

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

Column	Type	Description
<b>pid</b>	ANY-INTEGER	(optional) Identifier of the point. <ul style="list-style-type: none"> <li>• If column present, it can not be NULL.</li> <li>• If column not present, a sequential identifier will be given automatically.</li> </ul>
<b>edge_id</b>	ANY-INTEGER	Identifier of the “closest” edge to the point.
<b>fraction</b>	ANY-NUMERICAL	Value in <0,1> that indicates the relative position from the first end point of the edge.
<b>side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: <ul style="list-style-type: none"> <li>• In the right, left of the edge or</li> <li>• If it doesn't matter with 'b' or NULL.</li> <li>• If column not present 'b' is considered.</li> </ul>

Where:

**ANY-INTEGER:**

smallint, int, bigint

**ANY-NUMERICAL:**

smallint, int, bigint, real, float

**Result Columns**

Column	Type	Description
<b>seq</b>	INTEGER	Row sequence.
<b>path_seq</b>	INTEGER	Path sequence that indicates the relative position on the path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. When negative: is a point's pid.
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. When negative: is a point's pid.
<b>node</b>	BIGINT	Identifier of the node: <ul style="list-style-type: none"> <li>• A positive value indicates the node is a vertex of edges_sql.</li> <li>• A negative value indicates the node is a point of points_sql.</li> </ul>
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next <code>node</code> in the path sequence. <ul style="list-style-type: none"> <li>• -1 for the last row in the path sequence.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next <code>node</code> in the path sequence. <ul style="list-style-type: none"> <li>• 0 for the last row in the path sequence.</li> </ul>
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_pid</code> to <code>node</code> . <ul style="list-style-type: none"> <li>• 0 for the first row in the path sequence.</li> </ul>

**Additional Examples**

**Example:**

Which path (if any) passes in front of point\((6)\) or vertex\((6)\) with **right** side driving topology.

```

SELECT ((' || start_pid || ' => ' || end_pid || ') at ' || path_seq || 'th step:'):TEXT AS path_at,
CASE WHEN edge = -1 THEN 'visits'
ELSE 'passes in front of'
END as status,
CASE WHEN node < 0 THEN 'Point'
ELSE 'Vertex'
END as is_a,
abs(node) as id
FROM pgr_withPoints(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
'SELECT pid, edge_id, fraction, side from pointsOfInterest',
ARRAY[1,-1], ARRAY[-2,-3,-6,3,6],
driving_side := 'r',
details := true)
WHERE node IN (-6,6);

```

path_at	status	is_a	id
(-1 => -6) at 4th step:	visits	Point	6
(-1 => -3) at 4th step:	passes in front of	Point	6
(-1 => -2) at 4th step:	passes in front of	Point	6
(-1 => -2) at 6th step:	passes in front of	Vertex	6
(-1 => 3) at 4th step:	passes in front of	Point	6
(-1 => 3) at 6th step:	passes in front of	Vertex	6
(-1 => 6) at 4th step:	passes in front of	Point	6
(-1 => 6) at 6th step:	visits	Vertex	6
(1 => -6) at 3th step:	visits	Point	6
(1 => -3) at 3th step:	passes in front of	Point	6
(1 => -2) at 3th step:	passes in front of	Point	6
(1 => -2) at 5th step:	passes in front of	Vertex	6
(1 => 3) at 3th step:	passes in front of	Point	6
(1 => 3) at 5th step:	passes in front of	Vertex	6
(1 => 6) at 3th step:	passes in front of	Point	6
(1 => 6) at 5th step:	visits	Vertex	6

(16 rows)

### Example:

Which path (if any) passes in front of point\6) or vertex\6) with **left** side driving topology.

```

SELECT ((' || start_pid || ' => ' || end_pid || ') at ' || path_seq || 'th step:'):TEXT AS path_at,
CASE WHEN edge = -1 THEN 'visits'
ELSE 'passes in front of'
END as status,
CASE WHEN node < 0 THEN 'Point'
ELSE 'Vertex'
END as is_a,
abs(node) as id
FROM pgr_withPoints(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
'SELECT pid, edge_id, fraction, side from pointsOfInterest',
ARRAY[1,-1], ARRAY[-2,-3,-6,3,6],
driving_side := 'l',
details := true)
WHERE node IN (-6,6);

```

path_at	status	is_a	id
(-1 => -6) at 3th step:	visits	Point	6
(-1 => -3) at 3th step:	passes in front of	Point	6
(-1 => -2) at 3th step:	passes in front of	Point	6
(-1 => -2) at 5th step:	passes in front of	Vertex	6
(-1 => 3) at 3th step:	passes in front of	Point	6
(-1 => 3) at 5th step:	passes in front of	Vertex	6
(-1 => 6) at 3th step:	passes in front of	Point	6
(-1 => 6) at 5th step:	visits	Vertex	6
(1 => -6) at 4th step:	visits	Point	6
(1 => -3) at 4th step:	passes in front of	Point	6
(1 => -2) at 4th step:	passes in front of	Point	6
(1 => -2) at 6th step:	passes in front of	Vertex	6
(1 => 3) at 4th step:	passes in front of	Point	6
(1 => 3) at 6th step:	passes in front of	Vertex	6
(1 => 6) at 4th step:	passes in front of	Point	6
(1 => 6) at 6th step:	visits	Vertex	6

(16 rows)

### Example:

From point\1) and vertex\2) to point\3) to vertex\7) on an **undirected** graph, with details.

```

SELECT * FROM pgr_withPoints(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
'SELECT pid, edge_id, fraction, side from pointsOfInterest',
ARRAY[-1,2], ARRAY[-3,7],
directed := false,
details := true);
seq | path_seq | start_pid | end_pid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----+-----

```

1	1	-1	-3	-1	1	0.6	0
2	2	-1	-3	2	4	0.7	0.6
3	3	-1	-3	-6	4	0.3	1.3
4	4	-1	-3	5	10	1	1.6
5	5	-1	-3	10	12	0.6	2.6
6	6	-1	-3	-3	-1	0	3.2
7	1	-1	7	-1	1	0.6	0
8	2	-1	7	2	4	0.7	0.6
9	3	-1	7	-6	4	0.3	1.3
10	4	-1	7	5	7	1	1.6
11	5	-1	7	8	6	0.7	2.6
12	6	-1	7	-4	6	0.3	3.3
13	7	-1	7	7	-1	0	3.6
14	1	2	-3	2	4	0.7	0
15	2	2	-3	-6	4	0.3	0.7
16	3	2	-3	5	10	1	1
17	4	2	-3	10	12	0.6	2
18	5	2	-3	-3	-1	0	2.6
19	1	2	7	2	4	0.7	0
20	2	2	7	-6	4	0.3	0.7
21	3	2	7	5	7	1	1
22	4	2	7	8	6	0.7	2
23	5	2	7	-4	6	0.3	2.7
24	6	2	7	7	-1	0	3

(24 rows)

The queries use the **Sample Data** network

See Also

- [withPoints - Family of functions](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

## pgr\_withPointsCost - Proposed

`pgr_withPointsCost` - Calculates the shortest path and returns only the aggregate cost of the shortest path(s) found, for the combination of points given.



### Warning

Proposed functions for next mayor release.

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



Boost Graph Inside

## Availability

- Version 2.2.0
  - New **proposed** function

## Support



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

#### Description

Modify the graph to include points defined by `points_sql`. Using Dijkstra algorithm, return only the aggregate cost of the shortest path(s) found.

The main characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of vertices in the modified graph.
- Vertices of the graph are:
  - positive** when it belongs to the `edges_sql`
  - negative** when it belongs to the `points_sql`
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of  $(start\_vid, end\_vid, agg\_cost)$ .
  - When the starting vertex and ending vertex are the same, there is no path.
    - The  $agg\_cost$  in the non included values  $(v, v)$  is 0
  - When the starting vertex and ending vertex are the different and there is no path.
    - The  $agg\_cost$  in the non included values  $(u, v)$  is  $-\infty$
- If the values returned are stored in a table, the unique index would be the pair:  $(start\_vid, end\_vid)$ .
- For **undirected** graphs, the results are **symmetric**.
  - The  $agg\_cost$  of  $(u, v)$  is the same as for  $(v, u)$ .
- For optimization purposes, any duplicated value in the `start_vids` or `end_vids` is ignored.
- The returned values are ordered:
  - `start_vid` ascending
  - `end_vid` ascending
- Running time:  $O(|start\_vids| * (V \log V + E))$

#### Signatures

#### Summary

```
pgr_withPointsCost(edges_sql, points_sql, from_vid, to_vid [, directed] [, driving_side])
pgr_withPointsCost(edges_sql, points_sql, from_vid, to_vids [, directed] [, driving_side])
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vid [, directed] [, driving_side])
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```



#### Note

There is no **details** flag, unlike the other members of the `withPoints` family of functions.

#### Using defaults

```
pgr_withPointsCost(edges_sql, points_sql, start_vid, end_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

#### Example:

From point  $(1)$  to point  $(3)$

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, -3);
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 3.2
(1 row)
```

#### One to One

```
pgr_withPointsCost(edges_sql, points_sql, from_vid, to_vid [, directed] [, driving_side])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

**Example:**

From point \{1\} to vertex \{3\} on an **undirected** graph.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 3,
  directed := false);
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | 3 | 1.6
(1 row)
```

**One to Many**

```
pgr_withPointsCost(edges_sql, points_sql, from_vid, to_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

**Example:**

From point \{1\} to point \{3\} and vertex \{5\} on a **directed** graph.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, ARRAY[-3,5]);
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 3.2
-1 | 5 | 1.6
(2 rows)
```

**Many to One**

```
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vid [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

**Example:**

From point \{1\} and vertex \{2\} to point \{3\} on a **directed** graph.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], -3);
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 3.2
2 | -3 | 2.6
(2 rows)
```

**Many to Many**

```
pgr_withPointsCost(edges_sql, points_sql, from_vids, to_vids [, directed] [, driving_side])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

**Example:**

From point \{1\} and vertex \{2\} to point \{3\} and vertex \{7\} on a **directed** graph.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], ARRAY[-3,7]);
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 3.2
-1 | 7 | 3.6
2 | -3 | 2.6
2 | 7 | 3
(4 rows)
```

**Parameters**

Parameter	Type	Description
<b>edges_sql</b>	TEXT	Edges SQL query as described above.
<b>points_sql</b>	TEXT	Points SQL query as described above.

Parameter	Type	Description
<b>start_vid</b>	ANY-INTEGER	Starting vertex identifier. When negative: is a point's pid.
<b>end_vid</b>	ANY-INTEGER	Ending vertex identifier. When negative: is a point's pid.
<b>start_vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of starting vertices. When negative: is a point's pid.
<b>end_vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of ending vertices. When negative: is a point's pid.
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>driving_side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: <ul style="list-style-type: none"> <li>In the right or left or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Description of the Points SQL query

#### points\_sql:

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

Column	Type	Description
<b>pid</b>	ANY-INTEGER	(optional) Identifier of the point. <ul style="list-style-type: none"> <li>If column present, it can not be NULL.</li> <li>If column not present, a sequential identifier will be given automatically.</li> </ul>
<b>edge_id</b>	ANY-INTEGER	Identifier of the "closest" edge to the point.
<b>fraction</b>	ANY-NUMERICAL	Value in <0,1> that indicates the relative position from the first end point of the edge.
<b>side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: <ul style="list-style-type: none"> <li>In the right, left of the edge or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>

Where:

#### ANY-INTEGER:

smallint, int, bigint

#### ANY-NUMERICAL:

smallint, int, bigint, real, float

#### Result Columns

Column	Type	Description
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. When negative: is a point's pid.

Column	Type	Description
<code>end_vid</code>	BIGINT	Identifier of the ending point. When negative: is a point's pid.
<code>agg_cost</code>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>end_vid</code> .

#### Additional Examples

##### Example:

From point \{1\} and vertex \{2\} to point \{3\} and vertex \{7\}, with **right** side driving topology

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], ARRAY[-3,7],
  driving_side := 'l');
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 3.2
-1 | 7 | 3.6
2 | -3 | 2.6
2 | 7 | 3
(4 rows)
```

##### Example:

From point \{1\} and vertex \{2\} to point \{3\} and vertex \{7\}, with **left** side driving topology

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], ARRAY[-3,7],
  driving_side := 'r');
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 4
-1 | 7 | 4.4
2 | -3 | 2.6
2 | 7 | 3
(4 rows)
```

##### Example:

From point \{1\} and vertex \{2\} to point \{3\} and vertex \{7\}, does not matter driving side.

```
SELECT * FROM pgr_withPointsCost(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  ARRAY[-1,2], ARRAY[-3,7],
  driving_side := 'b');
start_pid | end_pid | agg_cost
-----+-----+-----
-1 | -3 | 3.2
-1 | 7 | 3.6
2 | -3 | 2.6
2 | 7 | 3
(4 rows)
```

The queries use the **Sample Data** network.

#### See Also

- [withPoints - Family of functions](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### pgr\_withPointsKSP - Proposed

`pgr_withPointsKSP` - Find the K shortest paths using Yen's algorithm.



#### Warning

Proposed functions for next mayor release.

- They are not officially in the current release.
- They will likely officially be part of the next mayor release:
  - The functions make use of ANY-INTEGGER and ANY-NUMERICAL

- Name might not change. (But still can)
- Signature might not change. (But still can)
- Functionality might not change. (But still can)
- pgTap tests have being done. But might need more.
- Documentation might need refinement.



Boost Graph Inside

## Availability

- Version 2.2.0
  - New **proposed** function

## Support

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

## Description

Modifies the graph to include the points defined in the `points_sql` and using Yen algorithm, finds the  $\backslash(K)$  shortest paths.

## Signatures

## Summary

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K [, directed] [, heap_paths] [, driving_side] [, details])
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

## Using defaults

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K)
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

## Example:

From point  $\backslash(1)$  to point  $\backslash(2)$  in  $\backslash(2)$  cycles

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of the query.
- No **heap paths** are returned.

```
SELECT * FROM pgr_withPointsKSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, -2, 2);
```

```
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 1 | -1 | 1 | 0.6 | 0
 2 | 1 | 2 | 2 | 4 | 1 | 0.6
 3 | 1 | 3 | 5 | 8 | 1 | 1.6
 4 | 1 | 4 | 6 | 9 | 1 | 2.6
 5 | 1 | 5 | 9 | 15 | 0.4 | 3.6
 6 | 1 | 6 | -2 | -1 | 0 | 4
 7 | 2 | 1 | -1 | 1 | 0.6 | 0
 8 | 2 | 2 | 2 | 4 | 1 | 0.6
 9 | 2 | 3 | 5 | 8 | 1 | 1.6
10 | 2 | 4 | 6 | 11 | 1 | 2.6
11 | 2 | 5 | 11 | 13 | 1 | 3.6
12 | 2 | 6 | 12 | 15 | 0.6 | 4.6
13 | 2 | 7 | -2 | -1 | 0 | 5.2
(13 rows)
```

## Complete Signature

Finds the  $\backslash(K)$  shortest paths depending on the optional parameters setup.

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K [, directed] [, heap_paths] [, driving_side] [, details])
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

### Example:

From point \{1\} to vertex \{6\} in \{2\} cycles with details.

```
SELECT * FROM pgr_withPointsKSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 6, 2, details := true);
```

```
seq | path_id | path_seq | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 1 | -1 | 1 | 0.6 | 0
 2 | 1 | 2 | 2 | 2 | 4 | 0.7 | 0.6
 3 | 1 | 3 | -6 | 4 | 0.3 | 1.3
 4 | 1 | 4 | 5 | 8 | 1 | 1.6
 5 | 1 | 5 | 6 | -1 | 0 | 2.6
 6 | 2 | 1 | -1 | 1 | 0.6 | 0
 7 | 2 | 2 | 2 | 4 | 0.7 | 0.6
 8 | 2 | 3 | -6 | 4 | 0.3 | 1.3
 9 | 2 | 4 | 5 | 10 | 1 | 1.6
10 | 2 | 5 | 10 | 12 | 0.6 | 2.6
11 | 2 | 6 | -3 | 12 | 0.4 | 3.2
12 | 2 | 7 | 11 | 13 | 1 | 3.6
13 | 2 | 8 | 12 | 15 | 0.6 | 4.6
14 | 2 | 9 | -2 | 15 | 0.4 | 5.2
15 | 2 | 10 | 9 | 9 | 1 | 5.6
16 | 2 | 11 | 6 | -1 | 0 | 6.6
```

(16 rows)

### Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	Edges SQL query as described above.
<b>points_sql</b>	TEXT	Points SQL query as described above.
<b>start_pid</b>	ANY-INTEGER	Starting point id.
<b>end_pid</b>	ANY-INTEGER	Ending point id.
<b>K</b>	INTEGER	Number of shortest paths.
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>heap_paths</b>	BOOLEAN	(optional). When <code>true</code> the paths calculated to get the shortest paths will be returned also. Default is <code>false</code> only the K shortest paths are returned.
<b>driving_side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the driving side is: <ul style="list-style-type: none"> <li>In the right or left or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>
<b>details</b>	BOOLEAN	(optional). When <code>true</code> the results will include the driving distance to the points with in the <code>distance</code> . Default is <code>false</code> which ignores other points of the <code>points_sql</code> .

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICALS:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Description of the Points SQL query

### points\_sql:

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

Column	Type	Description
<b>pid</b>	ANY-INTEGERS	(optional) Identifier of the point. <ul style="list-style-type: none"> <li>If column present, it can not be NULL.</li> <li>If column not present, a sequential identifier will be given automatically.</li> </ul>
<b>edge_id</b>	ANY-INTEGERS	Identifier of the “closest” edge to the point.
<b>fraction</b>	ANY-NUMERICAL	Value in <0,1> that indicates the relative position from the first end point of the edge.
<b>side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: <ul style="list-style-type: none"> <li>In the right, left of the edge or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>

Where:

#### ANY-INTEGERS:

smallint, int, bigint

#### ANY-NUMERICAL:

smallint, int, bigint, real, float

### Result Columns

Column	Type	Description
<b>seq</b>	INTEGER	Row sequence.
<b>path_seq</b>	INTEGER	Relative position in the path of node and edge. Has value 1 for the beginning of a path.
<b>path_id</b>	INTEGER	Path identifier. The ordering of the paths: For two paths i, j if $i < j$ then $agg\_cost(i) \leq agg\_cost(j)$ .
<b>node</b>	BIGINT	Identifier of the node in the path. Negative values are the identifiers of a point.
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <ul style="list-style-type: none"> <li>-1 for the last row in the path sequence.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next <code>node</code> in the path sequence. <ul style="list-style-type: none"> <li>0 for the last row in the path sequence.</li> </ul>
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_pid</code> to <code>node</code> . <ul style="list-style-type: none"> <li>0 for the first row in the path sequence.</li> </ul>

### Additional Examples

#### Example:

Left side driving topology from point \{1\} to point \{2\} in \{2\} cycles, with details

```
SELECT * FROM pgr_withPointsKSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, -2, 2,
  driving_side := 'l', details := true);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 1 | 1 | -1 | 0.6 | 0
2 | 1 | 2 | 2 | 4 | 0.7 | 0.6
3 | 1 | 3 | -6 | 4 | 0.3 | 1.3
4 | 1 | 4 | 5 | 8 | 1 | 1.6
5 | 1 | 5 | 6 | 9 | 1 | 2.6
6 | 1 | 6 | 9 | 15 | 1 | 3.6
7 | 1 | 7 | 12 | 15 | 0.6 | 4.6
8 | 1 | 8 | -2 | -1 | 0 | 5.2
9 | 2 | 1 | -1 | 1 | 0.6 | 0
10 | 2 | 2 | 2 | 4 | 0.7 | 0.6
11 | 2 | 3 | -6 | 4 | 0.3 | 1.3
12 | 2 | 4 | 5 | 8 | 1 | 1.6
13 | 2 | 5 | 6 | 11 | 1 | 2.6
14 | 2 | 6 | 11 | 13 | 1 | 3.6
15 | 2 | 7 | 12 | 15 | 0.6 | 4.6
16 | 2 | 8 | -2 | -1 | 0 | 5.2
(16 rows)
```

#### Example:

Right side driving topology from point \{(1)\} to point \{(2)\} in \{(2)\} cycles, with heap paths and details

```
SELECT * FROM pgr_withPointsKSP(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, -2, 2,
  heap_paths := true, driving_side := 'r', details := true);
seq | path_id | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----
 1 | 1 | 1 | 1 | -1 | 1 | 0.4 | 0
 2 | 1 | 2 | 1 | 1 | 1 | 0.4 | 0.4
 3 | 1 | 3 | 2 | 4 | 0.7 | 1.4
 4 | 1 | 4 | -6 | 4 | 0.3 | 2.1
 5 | 1 | 5 | 5 | 8 | 1 | 2.4
 6 | 1 | 6 | 6 | 9 | 1 | 3.4
 7 | 1 | 7 | 9 | 15 | 0.4 | 4.4
 8 | 1 | 8 | -2 | -1 | 0 | 4.8
 9 | 2 | 1 | -1 | 1 | 0.4 | 0
10 | 2 | 2 | 1 | 1 | 1 | 0.4 | 0.4
11 | 2 | 3 | 2 | 4 | 0.7 | 1.4
12 | 2 | 4 | -6 | 4 | 0.3 | 2.1
13 | 2 | 5 | 5 | 8 | 1 | 2.4
14 | 2 | 6 | 6 | 11 | 1 | 3.4
15 | 2 | 7 | 11 | 13 | 1 | 4.4
16 | 2 | 8 | 12 | 15 | 1 | 5.4
17 | 2 | 9 | 9 | 15 | 0.4 | 6.4
18 | 2 | 10 | -2 | -1 | 0 | 6.8
19 | 3 | 1 | -1 | 1 | 0.4 | 0
20 | 3 | 2 | 1 | 1 | 1 | 0.4 | 0.4
21 | 3 | 3 | 2 | 4 | 0.7 | 1.4
22 | 3 | 4 | -6 | 4 | 0.3 | 2.1
23 | 3 | 5 | 5 | 10 | 1 | 2.4
24 | 3 | 6 | 10 | 12 | 0.6 | 3.4
25 | 3 | 7 | -3 | 12 | 0.4 | 4
26 | 3 | 8 | 11 | 13 | 1 | 4.4
27 | 3 | 9 | 12 | 15 | 1 | 5.4
28 | 3 | 10 | 9 | 15 | 0.4 | 6.4
29 | 3 | 11 | -2 | -1 | 0 | 6.8
(29 rows)
```

The queries use the **Sample Data** network.

See Also

- [withPoints - Family of functions](#)

Indices and tables

- [Index](#)
- [Search Page](#)

**pgr\_withPointsDD - Proposed**

`pgr_withPointsDD` - Returns the driving distance from a starting point.



#### Warning

Proposed functions for next mayor release.

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



Boost Graph Inside

Availability

- Version 2.2.0



- New **proposed** function

## Support

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

## Description

Modify the graph to include points and using Dijkstra algorithm, extracts all the nodes and points that have costs less than or equal to the value `distance` from the starting point. The edges extracted will conform the corresponding spanning tree.

## Signatures

## Summary

```
pgr_withPointsDD(edges_sql, points_sql, from_vids, distance [, directed] [, driving_side] [, details] [, equicost])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

## Using defaults

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of the query.

```
pgr_withPointsDD(edges_sql, points_sql, start_vid, distance)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

## Example:

From point `\(1\)` with `\(agg\_cost <= 3.8\)`

- For a **directed** graph.
- The driving side is set as **b** both. So arriving/departing to/from the point(s) can be in any direction.
- No **details** are given about distance of other points of the query.

```
SELECT * FROM pgr_withPointsDD(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 3.8);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 |  -1 |  -1 |  0 |    0
 2 |  1 |  1 | 0.4 |   0.4
 3 |  2 |  1 | 0.6 |   0.6
 4 |  5 |  4 |  1 |   1.6
 5 |  6 |  8 |  1 |   2.6
 6 |  8 |  7 |  1 |   2.6
 7 | 10 | 10 |  1 |   2.6
 8 |  7 |  6 |  1 |   3.6
 9 |  9 |  9 |  1 |   3.6
10 | 11 | 11 |  1 |   3.6
11 | 13 | 14 |  1 |   3.6
(11 rows)
```

## Single vertex

Finds the driving distance depending on the optional parameters setup.

```
pgr_withPointsDD(edges_sql, points_sql, from_vid, distance [, directed] [, driving_side] [, details])
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

## Example:

Right side driving topology, from point `\(1\)` with `\(agg\_cost <= 3.8\)`

```

SELECT * FROM pgr_withPointsDD(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
'SELECT pid, edge_id, fraction, side from pointsOfInterest',
-1, 3.8,
driving_side := 'r',
details := true);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | -1 | -1 | 0 | 0
 2 | 1 | 1 | 0.4 | 0.4
 3 | 2 | 1 | 1 | 1.4
 4 | -6 | 4 | 0.7 | 2.1
 5 | 5 | 4 | 0.3 | 2.4
 6 | 6 | 8 | 1 | 3.4
 7 | 8 | 7 | 1 | 3.4
 8 | 10 | 10 | 1 | 3.4
(8 rows)

```

## Multiple vertices

Finds the driving distance depending on the optional parameters setup.

```

pgr_withPointsDD(edges_sql, points_sql, from_vids, distance [, directed] [, driving_side] [, details] [, equicost])
RETURNS SET OF (seq, node, edge, cost, agg_cost)

```

### Parameters

Parameter	Type	Description
<b>edges_sql</b>	TEXT	Edges SQL query as described above.
<b>points_sql</b>	TEXT	Points SQL query as described above.
<b>start_vid</b>	ANY-INTEGER	Starting point id
<b>distance</b>	ANY-NUMERICAL	Distance from the start_pid
<b>directed</b>	BOOLEAN	(optional). When <code>false</code> the graph is considered as Undirected. Default is <code>true</code> which considers the graph as Directed.
<b>driving_side</b>	CHAR	(optional). Value in ['b', 'r', 'l', NULL] indicating if the driving side is: <ul style="list-style-type: none"> <li>In the right or left or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>
<b>details</b>	BOOLEAN	(optional). When <code>true</code> the results will include the driving distance to the points with in the <code>distance</code> . Default is <code>false</code> which ignores other points of the <code>points_sql</code> .
<b>equicost</b>	BOOLEAN	(optional). When <code>true</code> the nodes will only appear in the closest start_v list. Default is <code>false</code> which resembles several calls using the single starting point signatures. Tie brakes are arbitrary.

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICALS:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Description of the Points SQL query

**points\_sql:**

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

Column	Type	Description
<b>pid</b>	ANY-INTEGER	(optional) Identifier of the point. <ul style="list-style-type: none"> <li>If column present, it can not be NULL.</li> <li>If column not present, a sequential identifier will be given automatically.</li> </ul>
<b>edge_id</b>	ANY-INTEGER	Identifier of the “closest” edge to the point.
<b>fraction</b>	ANY-NUMERICAL	Value in <0,1> that indicates the relative position from the first end point of the edge.
<b>side</b>	CHAR	(optional) Value in ['b', 'r', 'l', NULL] indicating if the point is: <ul style="list-style-type: none"> <li>In the right, left of the edge or</li> <li>If it doesn't matter with 'b' or NULL.</li> <li>If column not present 'b' is considered.</li> </ul>

Where:

**ANY-INTEGER:**

smallint, int, bigint

**ANY-NUMERICAL:**

smallint, int, bigint, real, float

**Result Columns**

Column	Type	Description
<b>seq</b>	INT	row sequence.
<b>node</b>	BIGINT	Identifier of the node within the Distance from <code>start_pid</code> . If <code>details =: true</code> a negative value is the identifier of a point.
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <ul style="list-style-type: none"> <li>-1 when <code>start_vid = node</code>.</li> </ul>
<b>cost</b>	FLOAT	Cost to traverse <code>edge</code> . <ul style="list-style-type: none"> <li>0 when <code>start_vid = node</code>.</li> </ul>
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_vid</code> to <code>node</code> . <ul style="list-style-type: none"> <li>0 when <code>start_vid = node</code>.</li> </ul>

**Additional Examples**

**Examples for queries marked as `directed with cost and reverse_cost` columns.**

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

**Example:**

Left side driving topology from point \{1\} with \{(agg\\_cost <= 3.8)\}, with details

```
SELECT * FROM pgr_withPointsDD(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  'SELECT pid, edge_id, fraction, side from pointsOfInterest',
  -1, 3.8,
  driving_side := 'l',
  details := true);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | -1 | -1 | 0 | 0
 2 | 2 | 1 | 0.6 | 0.6
 3 | -6 | 4 | 0.7 | 1.3
 4 | 5 | 4 | 0.3 | 1.6
 5 | 1 | 1 | 1 | 1.6
 6 | 6 | 8 | 1 | 2.6
 7 | 8 | 7 | 1 | 2.6
 8 | 10 | 10 | 1 | 2.6
 9 | -3 | 12 | 0.6 | 3.2
10 | -4 | 6 | 0.7 | 3.3
11 | 7 | 6 | 0.3 | 3.6
12 | 9 | 9 | 1 | 3.6
13 | 11 | 11 | 1 | 3.6
14 | 13 | 14 | 1 | 3.6
(14 rows)
```

**Example:**

From point \{1\} with \{(agg\\_cost <= 3.8)\}, does not matter driving side, with details

```

SELECT * FROM pgr_withPointsDD(
'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
'SELECT pid, edge_id, fraction, side from pointsOfInterest',
-1, 3.8,
driving_side := 'b',
details := true);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
 1 | -1 | -1 | 0 | 0
 2 | 1 | 1 | 0.4 | 0.4
 3 | 2 | 1 | 0.6 | 0.6
 4 | -6 | 4 | 0.7 | 1.3
 5 | 5 | 4 | 0.3 | 1.6
 6 | 6 | 8 | 1 | 2.6
 7 | 8 | 7 | 1 | 2.6
 8 | 10 | 10 | 1 | 2.6
 9 | -3 | 12 | 0.6 | 3.2
10 | -4 | 6 | 0.7 | 3.3
11 | 7 | 6 | 0.3 | 3.6
12 | 9 | 9 | 1 | 3.6
13 | 11 | 11 | 1 | 3.6
14 | 13 | 14 | 1 | 3.6
(14 rows)

```

The queries use the **Sample Data** network.

See Also

- o **pgr\_drivingDistance** - Driving distance using dijkstra.
- o **pgr\_alphaShape** - Alpha shape computation.

**Indices and tables**

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

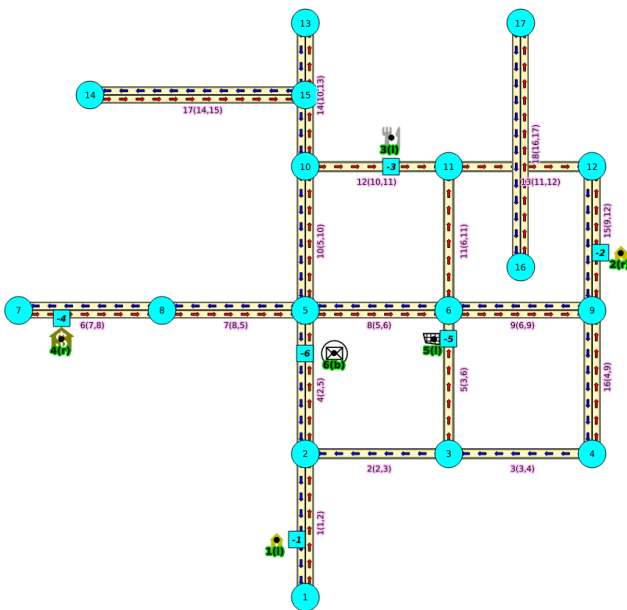
**Previous versions of this page**

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

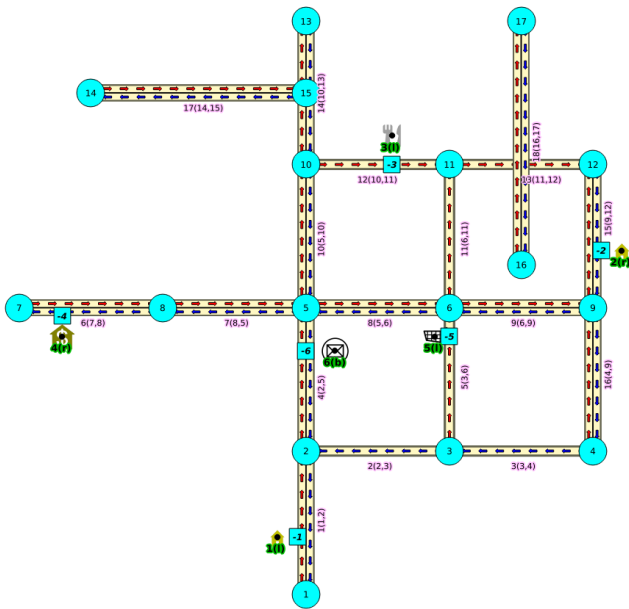
**Images**

The squared vertices are the temporary vertices, The temporary vertices are added according to the driving side, The following images visually show the differences on how depending on the driving side the data is interpreted.

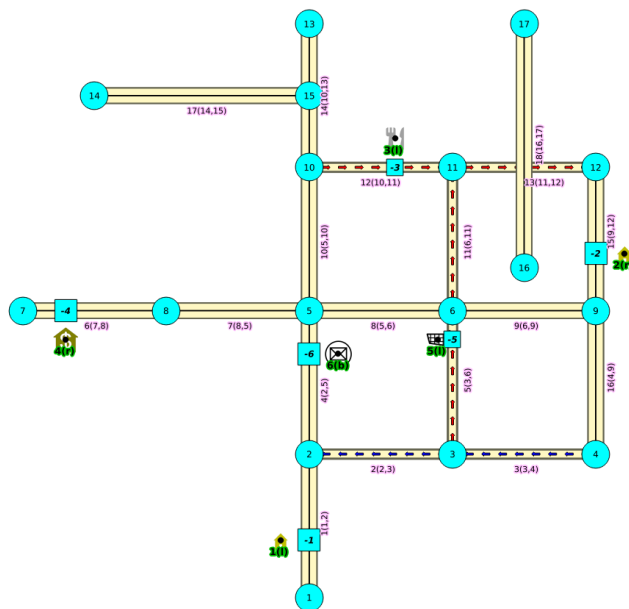
**Right driving side**



**Left driving side**



doesn't matter the driving side



## Introduction

This family of functions was thought for routing vehicles, but might as well work for some other application that we can not think of.

The with points family of function give you the ability to route between arbitrary points located outside the original graph.

When given a point identified with a *pid* that its being mapped to and edge with an *identifiedge\_id*, with a *fraction* along that edge (from the source to the target of the edge) and some additional information about which *side* of the edge the point is on, then routing from arbitrary points more accurately reflect routing vehicles in road networks,

I talk about a family of functions because it includes different functionalities.

- `pgr_withPoints` is `pgr_dijkstra` based
- `pgr_withPointsCost` is `pgr_dijkstraCost` based
- `pgr_withPointsKSP` is `pgr_ksp` based
- `pgr_withPointsDD` is `pgr_drivingDistance` based

In all this functions we have to take care of as many aspects as possible:

- Must work for routing:
  - Cars (directed graph)
  - Pedestrians (undirected graph)
- Arriving at the point:
  - In either side of the street.

- Compulsory arrival on the side of the street where the point is located.
- Countries with:
  - Right side driving
  - Left side driving
- Some points are:
  - Permanent, for example the set of points of clients stored in a table in the data base
  - Temporal, for example points given through a web application
- The numbering of the points are handled with negative sign.
  - Original point identifiers are to be positive.
  - Transformation to negative is done internally.
  - For results for involving vertices identifiers
    - positive sign is a vertex of the original graph
    - negative sign is a point of the temporary points

The reason for doing this is to avoid confusion when there is a vertex with the same number as identifier as the points identifier.

### Graph & edges

- Let  $(G_d(V,E))$  where  $(V)$  is the set of vertices and  $(E)$  is the set of edges be the original directed graph.
  - An edge of the original *edges\_sql* is  $((id, source, target, cost, reverse\_cost))$  will generate internally
    - $((id, source, target, cost))$
    - $((id, target, source, reverse\_cost))$

### Point Definition

- A point is defined by the quadruplet:  $((pid, eid, fraction, side))$ 
  - pid** is the point identifier
  - eid** is an edge id of the *edges\_sql*
  - fraction** represents where the edge *eid* will be cut.
  - side** Indicates the side of the edge where the point is located.

### Creating Temporary Vertices in the Graph

For edge (15, 9,12 10, 20), & lets insert point (2, 12, 0.3, r)

### On a right hand side driving network

From first image above:

- We can arrive to the point only via vertex 9.
- It only affects the edge (15, 9,12, 10) so that edge is removed.
- Edge (15, 12,9, 20) is kept.
- Create new edges:
  - (15, 9,-1, 3) edge from vertex 9 to point 1 has cost 3
  - (15, -1,12, 7) edge from point 1 to vertex 12 has cost 7

### On a left hand side driving network

From second image above:

- We can arrive to the point only via vertex 12.
- It only affects the edge (15, 12,9 20) so that edge is removed.
- Edge (15, 9,12, 10) is kept.
- Create new edges:
  - (15, 12,-1, 14) edge from vertex 12 to point 1 has cost 14
  - (15, -1,9, 6) edge from point 1 to vertex 9 has cost 6

### Remember:

that fraction is from vertex 9 to vertex 12

### When driving side does not matter

From third image above:

- We can arrive to the point either via vertex 12 or via vertex 9
- Edge (15, 12,9 20) is removed.
- Edge (15, 9,12, 10) is removed.
- Create new edges:
  - (15, 12,-1, 14) edge from vertex 12 to point 1 has cost 14
  - (15, -1,9, 6) edge from point 1 to vertex 9 has cost 6

- (15, 9,-1, 3) edge from vertex 9 to point 1 has cost 3
- (15, -1,12, 7) edge from point 1 to vertex 12 has cost 7

See Also

#### Indices and tables

- [Index](#)
- [Search Page](#)

See Also

- [Experimental Functions](#)

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### Experimental Functions



##### Warning

Possible server crash

- These functions might create a server crash



##### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGERS 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

#### Families

##### Flow - Family of functions

- [pgr\\_maxFlowMinCost](#) - **Experimental** - Details of flow and cost on edges.
- [pgr\\_maxFlowMinCost\\_Cost](#) - **Experimental** - Only the Min Cost calculation.

##### Chinese Postman Problem - Family of functions (Experimental)

- [pgr\\_chinesePostman](#) - **Experimental**
- [pgr\\_chinesePostmanCost](#) - **Experimental**

##### Topology - Family of Functions

- [pgr\\_extractVertices](#) - **Experimental** - Extracts vertices information based on the source and target.

##### Transformation - Family of functions (Experimental)

- [pgr\\_lineGraph](#) - **Experimental** - Transformation algorithm for generating a Line Graph.
- [pgr\\_lineGraphFull](#) - **Experimental** - Transformation algorithm for generating a Line Graph out of each vertex in the input graph.

Chinese Postman Problem - Family of functions (Experimental)

- **pgr\_chinesePostman - Experimental**
- **pgr\_chinesePostmanCost - Experimental**

### pgr\_chinesePostman - Experimental

pgr\_chinesePostman — Calculates the shortest circuit path which contains every edge in a directed graph and starts and ends on the same vertex.



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

### Availability

- Version 3.0.0
  - New **experimental** function

### Support

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

### Description

The main characteristics are:

- Process is done only on edges with **positive** costs.
- Running time:  $\mathcal{O}(E * (E + V * \log V))$
- Graph must be connected.
- Returns `EMPTY SET` on a disconnected graph

### Signatures

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

### Example:



```

SELECT * FROM pgr_chinesePostman(
'SELECT id,
source, target,
cost, reverse_cost FROM edge_table where id < 17'
);
seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----
1 | 1 | 1 | 1 | 0
2 | 2 | 4 | 1 | 1
3 | 5 | 4 | 1 | 2
4 | 2 | 4 | 1 | 3
5 | 5 | 7 | 1 | 4
6 | 8 | 6 | 1 | 5
7 | 7 | 6 | 1 | 6
8 | 8 | 7 | 1 | 7
9 | 5 | 8 | 1 | 8
10 | 6 | 8 | 1 | 9
11 | 5 | 10 | 1 | 10
12 | 10 | 10 | 1 | 11
13 | 5 | 10 | 1 | 12
14 | 10 | 14 | 1 | 13
15 | 13 | 14 | 1 | 14
16 | 10 | 12 | 1 | 15
17 | 11 | 13 | 1 | 16
18 | 12 | 15 | 1 | 17
19 | 9 | 9 | 1 | 18
20 | 6 | 9 | 1 | 19
21 | 9 | 15 | 1 | 20
22 | 12 | 15 | 1 | 21
23 | 9 | 16 | 1 | 22
24 | 4 | 3 | 1 | 23
25 | 3 | 5 | 1 | 24
26 | 6 | 11 | 1 | 25
27 | 11 | 13 | 1 | 26
28 | 12 | 15 | 1 | 27
29 | 9 | 16 | 1 | 28
30 | 4 | 16 | 1 | 29
31 | 9 | 16 | 1 | 30
32 | 4 | 3 | 1 | 31
33 | 3 | 2 | 1 | 32
34 | 2 | 1 | 1 | 33
35 | 1 | -1 | 0 | 34
(35 rows)

```

#### Parameters

Column	Type	Default	Description
<b>edges_sql</b>	TEXT		The edges SQL query as described in <b>Inner query</b> .

#### Inner query

An Edges SQL that represents a **directed** graph with the following columns

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

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
<b>seq</b>	INT	Sequential value starting from <b>1</b> .

Column	Type	Description
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

See Also

- [Chinese Postman Problem - Family of functions \(Experimental\)](#)

## Indices and tables

- [Index](#)
- [Search Page](#)

## pgr\_chinesePostmanCost - Experimental

`pgr_chinesePostmanCost` — Calculates the minimum costs of a circuit path which contains every edge in a directed graph and starts and ends on the same vertex.



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Availability

- Version 3.0.0
  - New **experimental** function

## Support

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

## Description

The main characteristics are:

- Process is done only on edges with **positive** costs.
- Running time:  $\mathcal{O}(E * (E + V * \log V))$
- Graph must be connected.
- [TBD] Return value when the graph is disconnected

## Signatures

```
pgr_chinesePostmanCost(edges_sql)
RETURNS FLOAT
```

### Example:

```
SELECT * FROM pgr_chinesePostmanCost(
'SELECT id,
source, target,
cost, reverse_cost FROM edge_table where id < 17'
);
pgr_chinesePostmanCost
-----
34
(1 row)
```

### Parameters

Column	Type	Default	Description
<code>edges_sql</code>	TEXT		The edges SQL query as described in <b>Inner query</b> .

### Inner query

An Edges SQL that represents a **directed** graph with the following columns

Column	Type	Default	Description
<code>id</code>	ANY-INTEGERS		Identifier of the edge.
<code>source</code>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<code>target</code>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<code>cost</code>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<code>reverse_cost</code>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Result Columns

Type	Description
FLOAT	Minimum costs of a circuit path.

See Also

- Chinese Postman Problem - Family of functions (Experimental)

### Indices and tables

- Index
- Search Page



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.

- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Versions of this page

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

## Description

The main characteristics are:

- Process is done only on edges with **positive** costs.
- Running time:  $\mathcal{O}(E * (E + V * \log V))$
- Graph must be connected.

## Parameters

Column	Type	Default	Description
<b>edges_sql</b>	TEXT		The edges SQL query as described in <b>Inner query</b> .

## Inner query

An Edges SQL that represents a **directed** graph with the following columns

Column	Type	Default	Description
<b>id</b>	ANY-INTEGGER		Identifier of the edge.
<b>source</b>	ANY-INTEGGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>◦ When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>◦ When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

### ANY-INTEGGER:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## See Also

## Indices and tables

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

## categories

### Vehicle Routing Functions - Category (Experimental)

- Pickup and delivery problem
  - **pg\_r\_pickDeliver - Experimental** - Pickup & Delivery using a Cost Matrix
  - **pg\_r\_pickDeliverEuclidean - Experimental** - Pickup & Delivery with Euclidean distances
- Distribution problem

- **pgr\_vrpOneDepot - Experimental** - From a single depot, distributes orders

## Vehicle Routing Functions - Category (Experimental)



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - 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

- **Vehicle Routing Functions - Category (Experimental)**
  - **Introduction**
    - **Characteristics**
  - **Pick & Delivery**
  - **Parameters**
    - **Pick & deliver**
  - **Inner Queries**
    - **Pick & Deliver Orders SQL**
    - **Pick & Deliver Vehicles SQL**
    - **Pick & Deliver Matrix SQL**
  - **Results**
    - **Description of the result (TODO Disussion: Euclidean & Matrix)**
    - **Description of the result (TODO Disussion: Euclidean & Matrix)**
  - **Handling Parameters**
    - **Capacity and Demand Units Handling**
    - **Locations**
    - **Time Handling**
    - **Factor Handling**
  - **See Also**

## 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-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Availability

- Version 3.0.0
  - New **experimental** function

## Support

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

## Synopsis

Problem: Distribute and optimize the pickup-delivery pairs into a fleet of vehicles.

- Optimization problem is NP-hard.
- pickup and Delivery with time windows.
- All vehicles are equal.
  - Same Starting location.
  - Same Ending location which is the same as Starting location.
  - All vehicles travel at the same speed.
- A customer is for doing a pickup or doing a deliver.
  - has an open time.
  - has a closing time.
  - has a service time.
  - has an (x, y) location.
- There is a customer where to deliver a pickup.
  - travel time between customers is distance / speed
  - pickup and delivery pair is done with the same vehicle.
  - A pickup is done before the delivery.

## Characteristics

- All trucks depart at time 0.
- No multiple time windows for a location.
- Less vehicle used is considered better.
- Less total duration is better.
- Less wait time is better.
- the algorithm will raise an exception when
  - If there is a pickup-deliver pair than violates time window
  - The speed, max\_cycles, ma\_capacity have illegal values
- Six different initial will be optimized - the best solution found will be result

## Signature

```
pgr_pickDeliver(orders_sql, vehicles_sql, matrix_sql [, factor, max_cycles, initial_sol])  
RETURNS SET OF (seq, vehicle_number, vehicle_id, stop, order_id, stop_type, cargo,  
travel_time, arrival_time, wait_time, service_time, departure_time)
```

## Parameters

The parameters are:

orders\_sql, vehicles\_sql, matrix\_sql [, factor, max\_cycles, initial\_sol]

Column	Type	Default	Description
<b>orders_sql</b>	TEXT		<b>Pick &amp; Deliver Orders SQL</b> query containing the orders to be processed.
<b>vehicles_sql</b>	TEXT		<b>Pick &amp; Deliver Vehicles SQL</b> query containing the vehicles to be used.
<b>matrix_sql</b>	TEXT		<b>Pick &amp; Deliver Matrix SQL</b> query containing the distance or travel times.
<b>factor</b>	NUMERIC	1	Travel time multiplier. See <b>Factor Handling</b>
<b>max_cycles</b>	INTEGER	10	Maximum number of cycles to perform on the optimization.
<b>initial_sol</b>	INTEGER	4	Initial solution to be used. <ul style="list-style-type: none"> <li>○ 1 One order per truck</li> <li>○ 2 Push front order.</li> <li>○ 3 Push back order.</li> <li>○ 4 Optimize insert.</li> <li>○ 5 Push back order that allows more orders to be inserted at the back</li> <li>○ 6 Push front order that allows more orders to be inserted at the front</li> </ul>

#### Pick & Deliver Orders SQL

A *SELECT* statement that returns the following columns:

id, demand  
p\_node\_id, p\_open, p\_close, [p\_service, ]  
d\_node\_id, d\_open, d\_close, [d\_service, ]

where:

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the pick-delivery order pair.
<b>demand</b>	ANY-NUMERICAL		Number of units in the order
<b>p_open</b>	ANY-NUMERICAL		The time, relative to 0, when the pickup location opens.
<b>p_close</b>	ANY-NUMERICAL		The time, relative to 0, when the pickup location closes.
<b>d_service</b>	ANY-NUMERICAL	0	The duration of the loading at the pickup location.
<b>d_open</b>	ANY-NUMERICAL		The time, relative to 0, when the delivery location opens.
<b>d_close</b>	ANY-NUMERICAL		The time, relative to 0, when the delivery location closes.
<b>d_service</b>	ANY-NUMERICAL	0	The duration of the loading at the delivery location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Description
<b>p_node_id</b>	ANY-INTEGER	The node identifier of the pickup, must match a node identifier in the matrix table.
<b>d_node_id</b>	ANY-INTEGER	The node identifier of the delivery, must match a node identifier in the matrix table.

#### Pick & Deliver Vehicles SQL

A *SELECT* statement that returns the following columns:

id, capacity  
start\_node\_id, start\_open, start\_close [, start\_service, ]  
[ end\_node\_id, end\_open, end\_close, end\_service ]

where:

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the pick-delivery order pair.
<b>capacity</b>	ANY-NUMERICAL		Number of units in the order
<b>speed</b>	ANY-NUMERICAL	1	Average speed of the vehicle.
<b>start_open</b>	ANY-NUMERICAL		The time, relative to 0, when the starting location opens.
<b>start_close</b>	ANY-NUMERICAL		The time, relative to 0, when the starting location closes.
<b>start_service</b>	ANY-NUMERICAL	0	The duration of the loading at the starting location.
<b>end_open</b>	ANY-NUMERICAL	start_open	The time, relative to 0, when the ending location opens.
<b>end_close</b>	ANY-NUMERICAL	start_close	The time, relative to 0, when the ending location closes.
<b>end_service</b>	ANY-NUMERICAL	start_service	The duration of the loading at the ending location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Default	Description
<b>start_node_id</b>	ANY-INTEGERS		The node identifier of the starting location, must match a node identifier in the matrix table.
<b>end_node_id</b>	ANY-INTEGERS	start_node_id	The node identifier of the ending location, must match a node identifier in the matrix table.

Pick & Deliver Matrix SQL

A *SELECT* statement that returns the following columns:



**Warning**

TODO

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

**Example**

This example use the following data: [TODO put link](#)

```
SELECT * FROM pgr_pickDeliver(
  $$ SELECT * FROM orders ORDER BY id $$,
  $$ SELECT * FROM vehicles ORDER BY id $$,
  $$ SELECT * from pgr_dijkstraCostMatrix(
    'SELECT * FROM edge_table ',
    (SELECT array_agg(id) FROM (SELECT p_node_id AS id FROM orders
    UNION
    SELECT d_node_id FROM orders
    UNION
    SELECT start_node_id FROM vehicles) a)
  $$
);
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 | 6 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2 | 1 | 1 | 2 | 5 | 3 | 30 | 1 | 1 | 1 | 3 | 5
3 | 1 | 1 | 3 | 3 | 11 | 3 | 0 | 2 | 7 | 0 | 3 | 10
4 | 1 | 1 | 4 | 2 | 9 | 2 | 20 | 2 | 12 | 0 | 2 | 14
5 | 1 | 1 | 5 | 3 | 4 | 2 | 0 | 1 | 15 | 0 | 3 | 18
6 | 1 | 1 | 6 | 6 | 6 | -1 | 0 | 2 | 20 | 0 | 0 | 20
7 | 2 | 1 | 1 | 1 | 6 | -1 | 0 | 0 | 0 | 0 | 0 | 0
8 | 2 | 1 | 2 | 2 | 3 | 1 | 10 | 3 | 3 | 0 | 3 | 6
9 | 2 | 1 | 3 | 3 | 8 | 1 | 0 | 3 | 9 | 0 | 3 | 12
10 | 2 | 1 | 4 | 6 | 6 | -1 | 0 | 2 | 14 | 0 | 0 | 14
11 | -2 | 0 | 0 | -1 | -1 | -1 | -1 | 16 | -1 | 1 | 17 | 34
(11 rows)
```

**See Also**


- [Vehicle Routing Functions - Category \(Experimental\)](#)
- The queries use the **Sample Data** network.

**Indices and tables**

- [Index](#)
- [Search Page](#)

**pgr\_pickDeliverEuclidean - Experimental**


pgr\_pickDeliverEuclidean - Pickup and delivery Vehicle Routing Problem



**Warning**

Possible server crash

- These functions might create a server crash



**Warning**

Experimental functions



- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Availability

- Version 3.0.0
  - Replaces `pgr_gsoc_vrppdtw`
  - New **experimental** function

## Support

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

## Synopsis

Problem: Distribute and optimize the pickup-delivery pairs into a fleet of vehicles.

- Optimization problem is NP-hard.
- Pickup and Delivery:
  - capacitated
  - with time windows.
- The vehicles
  - have (x, y) start and ending locations.
  - have a start and ending service times.
  - have opening and closing times for the start and ending locations.
- An order is for doing a pickup and a a deliver.
  - has (x, y) pickup and delivery locations.
  - has opening and closing times for the pickup and delivery locations.
  - has a pickup and deliver service times.
- There is a customer where to deliver a pickup.
  - travel time between customers is distance / speed
  - pickup and delivery pair is done with the same vehicle.
  - A pickup is done before the delivery.

## Characteristics

- No multiple time windows for a location.
- Less vehicle used is considered better.
- Less total duration is better.
- Less wait time is better.
- Six different optional different initial solutions
  - the best solution found will be result

## Signature

```
pgr_pickDeliverEuclidean(orders_sql, vehicles_sql [,factor, max_cycles, initial_sol])
RETURNS SET OF (seq, vehicle_seq, vehicle_id, stop_seq, stop_type, order_id,
cargo, travel_time, arrival_time, wait_time, service_time, departure_time)
```

## Parameters

The parameters are:

```
orders_sql, vehicles_sql [,factor, max_cycles, initial_sol]
```

Where:

Column	Type	Default	Description
<b>orders_sql</b>	TEXT		<b>Pick &amp; Deliver Orders SQL</b> query containing the orders to be processed.
<b>vehicles_sql</b>	TEXT		<b>Pick &amp; Deliver Vehicles SQL</b> query containing the vehicles to be used.
<b>factor</b>	NUMERIC	1	(Optional) Travel time multiplier. See <b>Factor Handling</b>
<b>max_cycles</b>	INTEGER	10	(Optional) Maximum number of cycles to perform on the optimization.
<b>initial_sol</b>	INTEGER	4	(Optional) Initial solution to be used.

- 1 One order per truck
- 2 Push front order.
- 3 Push back order.
- 4 Optimize insert.
- 5 Push back order that allows more orders to be inserted at the back
- 6 Push front order that allows more orders to be inserted at the front

#### Pick & Deliver Orders SQL

A *SELECT* statement that returns the following columns:

```
id, demand
p_x, p_y, p_open, p_close, [p_service, ]
d_x, d_y, d_open, d_close, [d_service, ]
```

Where:

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the pick-delivery order pair.
<b>demand</b>	ANY-NUMERICAL		Number of units in the order
<b>p_open</b>	ANY-NUMERICAL		The time, relative to 0, when the pickup location opens.
<b>p_close</b>	ANY-NUMERICAL		The time, relative to 0, when the pickup location closes.
<b>d_service</b>	ANY-NUMERICAL	0	The duration of the loading at the pickup location.
<b>d_open</b>	ANY-NUMERICAL		The time, relative to 0, when the delivery location opens.
<b>d_close</b>	ANY-NUMERICAL		The time, relative to 0, when the delivery location closes.
<b>d_service</b>	ANY-NUMERICAL	0	The duration of the loading at the delivery location.

For the euclidean implementation, pick up and delivery  $((x,y))$  locations are needed:

Column	Type	Description
<b>p_x</b>	ANY-NUMERICAL	$x$ value of the pick up location
<b>p_y</b>	ANY-NUMERICAL	$y$ value of the pick up location
<b>d_x</b>	ANY-NUMERICAL	$x$ value of the delivery location
<b>d_y</b>	ANY-NUMERICAL	$y$ value of the delivery location

#### Pick & Deliver Vehicles SQL

A *SELECT* statement that returns the following columns:

```
id, capacity
start_x, start_y, start_open, start_close [ start_service, ]
[ end_x, end_y, end_open, end_close, end_service ]
```

where:

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the pick-delivery order pair.
<b>capacity</b>	ANY-NUMERICAL		Number of units in the order
<b>speed</b>	ANY-NUMERICAL	1	Average speed of the vehicle.
<b>start_open</b>	ANY-NUMERICAL		The time, relative to 0, when the starting location opens.
<b>start_close</b>	ANY-NUMERICAL		The time, relative to 0, when the starting location closes.
<b>start_service</b>	ANY-NUMERICAL	0	The duration of the loading at the starting location.
<b>end_open</b>	ANY-NUMERICAL	<i>start_open</i>	The time, relative to 0, when the ending location opens.
<b>end_close</b>	ANY-NUMERICAL	<i>start_close</i>	The time, relative to 0, when the ending location closes.
<b>end_service</b>	ANY-NUMERICAL	<i>start_service</i>	The duration of the loading at the ending location.

For the euclidean implementation, starting and ending  $((x,y))$  locations are needed:

Column	Type	Default	Description
<b>start_x</b>	ANY-NUMERICAL		\(x\) value of the coordinate of the starting location.
<b>start_y</b>	ANY-NUMERICAL		\(y\) value of the coordinate of the starting location.
<b>end_x</b>	ANY-NUMERICAL	<i>start_x</i>	\(x\) value of the coordinate of the ending location.
<b>end_y</b>	ANY-NUMERICAL	<i>start_y</i>	\(y\) value of the coordinate of the ending location.

Description of the result (TODO Disussion: Euclidean & Matrix)

RETURNS SET OF  
 (seq, vehicle\_seq, vehicle\_id, stop\_seq, stop\_type,  
 travel\_time, arrival\_time, wait\_time, service\_time, departure\_time)  
 UNION  
 (summary row)

Column	Type	Description
<b>seq</b>	INTEGER	Sequential value starting from <b>1</b> .
<b>vehicle_seq</b>	INTEGER	Sequential value starting from <b>1</b> for current vehicles. The \((n_{th})\) vehicle in the solution.
<b>vehicle_id</b>	BIGINT	Current vehicle identifier.
<b>stop_seq</b>	INTEGER	Sequential value starting from <b>1</b> for the stops made by the current vehicle. The \((m_{th})\) stop of the current vehicle.
<b>stop_type</b>	INTEGER	Kind of stop location the vehicle is at: <ul style="list-style-type: none"> <li>• <b>1</b>: Starting location</li> <li>• <b>2</b>: Pickup location</li> <li>• <b>3</b>: Delivery location</li> <li>• <b>6</b>: Ending location</li> </ul>
<b>order_id</b>	BIGINT	Pickup-Delivery order pair identifier. <ul style="list-style-type: none"> <li>• <b>-1</b>: When no order is involved on the current stop location.</li> </ul>
<b>cargo</b>	FLOAT	Cargo units of the vehicle when leaving the stop.
<b>travel_time</b>	FLOAT	Travel time from previous <code>stop_seq</code> to current <code>stop_seq</code> . <ul style="list-style-type: none"> <li>• <b>0</b> When <code>stop_type = 1</code></li> </ul>
<b>arrival_time</b>	FLOAT	Previous <code>departure_time</code> plus current <code>travel_time</code> .
<b>wait_time</b>	FLOAT	Time spent waiting for current <i>location</i> to open.
<b>service_time</b>	FLOAT	Service time at current <i>location</i> .
<b>departure_time</b>	FLOAT	\(arrival\_time + wait\_time + service\_time\). <ul style="list-style-type: none"> <li>• When <code>stop_type = 6</code> has the <i>total_time</i> used for the current vehicle.</li> </ul>

### Summary Row

**Warning**

TODO: Review the summary

Column	Type	Description
<b>seq</b>	INTEGER	Continues the Sequential value
<b>vehicle_seq</b>	INTEGER	<b>-2</b> to indicate is a summary row
<b>vehicle_id</b>	BIGINT	<i>Total Capacity Violations</i> in the solution.
<b>stop_seq</b>	INTEGER	<i>Total Time Window Violations</i> in the solution.
<b>stop_type</b>	INTEGER	<b>-1</b>
<b>order_id</b>	BIGINT	<b>-1</b>
<b>cargo</b>	FLOAT	<b>-1</b>
<b>travel_time</b>	FLOAT	<i>total_travel_time</i> The sum of all the <i>travel_time</i>
<b>arrival_time</b>	FLOAT	<b>-1</b>
<b>wait_time</b>	FLOAT	<i>total_waiting_time</i> The sum of all the <i>wait_time</i>
<b>service_time</b>	FLOAT	<i>total_service_time</i> The sum of all the <i>service_time</i>
<b>departure_time</b>	FLOAT	<i>total_solution_time</i> = \((total\_travel\_time + total\_wait\_time + total\_service\_time)\).

Where:

**ANY-INTEGER:**  
 SMALLINT, INTEGER, BIGINT

## ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Example

This example use the following data: [TODO put link](#)

```
SELECT * FROM pgr_pickDeliverEuclidean(
  'SELECT * FROM orders ORDER BY id',
  'SELECT * from vehicles'
);
```

seq	vehicle_seq	vehicle_id	stop_seq	stop_type	order_id	cargo	travel_time	arrival_time	wait_time	service_time	departure_time
1	1	1	1	1	-1	0	0	0	0	0	0
2	1	1	2	2	3	30	1	1	3	5	
3	1	1	3	3	3	0	1.41421356237	6.41421356237	0	3	9.41421356237
4	1	1	4	2	2	20	1.41421356237	10.8284271247	0	2	12.8284271247
5	1	1	5	3	2	0	1	13.8284271247	0	3	16.8284271247
6	1	1	6	6	-1	0	1.41421356237	18.2426406871	0	0	18.2426406871
7	2	1	1	1	-1	0	0	0	0	0	0
8	2	1	2	2	1	10	1	1	3	5	
9	2	1	3	3	1	0	2.2360679775	7.2360679775	0	3	10.2360679775
10	2	1	4	6	-1	0	2	12.2360679775	0	0	12.2360679775
11	-2	0	0	-1	-1	-1	11.4787086646	-1	2	17	30.4787086646

(11 rows)


### See Also

- [Vehicle Routing Functions - Category \(Experimental\)](#)
- The queries use the [Sample Data](#) network.


### Indices and tables

- [Index](#)
- [Search Page](#)

### pgr\_vrpOneDepot - Experimental

**Warning**  
Possible server crash

- These functions might create a server crash

**Warning**  
Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

### No documentation available

### Availability

- Version 2.1.0
  - New **experimental** function

### Support

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

**Description**

- TBD

**Signatures**

- TBD

**Parameters**

- TBD

**Inner query**

- TBD

**Result Columns**

- TBD

**Additional Example:**

```

BEGIN;
BEGIN
SET client_min_messages TO NOTICE;
SET
SELECT * FROM pgr_vrpOneDepot(
'SELECT * FROM solomon_100_RC_101',
'SELECT * FROM vrp_vehicles',
'SELECT * FROM vrp_distance',
1);
oid | opos | vid | tarrival | tdepart
-----+-----+-----+-----+-----
-1 | 1 | 1 | 0 | 0
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
);

COPY solomon_100_RC_101
(id, x, y, order_unit, open_time, close_time, service_time) FROM stdin;
1 40.000000 50.000000 0 0 240 0
2 25.000000 85.000000 20 145 175 10
3 22.000000 75.000000 30 50 80 10
4 22.000000 85.000000 10 109 139 10
5 20.000000 80.000000 40 141 171 10
6 20.000000 85.000000 20 41 71 10
7 18.000000 75.000000 20 95 125 10
8 15.000000 75.000000 20 79 109 10
9 15.000000 80.000000 10 91 121 10
10 10.000000 35.000000 20 91 121 10
11 10.000000 40.000000 30 119 149 10
\

DROP TABLE IF EXISTS vrp_vehicles cascade;
CREATE TABLE vrp_vehicles (
  vehicle_id integer not null primary key,
  capacity integer,
  case_no integer
);

copy vrp_vehicles (vehicle_id, capacity, case_no) from stdin;
1 200 5
2 200 5
3 200 5
\

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](https://en.wikipedia.org/wiki/Vehicle_routing_problem)

## Indices and tables

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

## Previous versions of this page

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

## Introduction

Vehicle Routing Problems *VRP* are **NP-hard** optimization problem, it generalises the travelling salesman problem (TSP).

- The objective of the VRP is to minimize the total route cost.
- There are several variants of the VRP problem,

**pgRouting does not try to implement all variants.**

## Characteristics

- Capacitated Vehicle Routing Problem *CVRP* where The vehicles have limited carrying capacity of the goods.
- Vehicle Routing Problem with Time Windows *VRPTW* where the locations have time windows within which the vehicle's visits must be made.
- Vehicle Routing Problem with Pickup and Delivery *VRPPD* where a number of goods need to be moved from certain pickup locations to other delivery locations.

## Limitations

- No multiple time windows for a location.

- Less vehicle used is considered better.
- Less total duration is better.
- Less wait time is better.

### Pick & Delivery

Problem: *CVRPPDTW* Capacitated Pick and Delivery Vehicle Routing problem with Time Windows

- Times are relative to 0
- The vehicles
  - have start and ending service duration times.
  - have opening and closing times for the start and ending locations.
  - have a capacity.
- The orders
  - Have pick up and delivery locations.
  - Have opening and closing times for the pickup and delivery locations.
  - Have pickup and delivery duration service times.
  - have a demand request for moving goods from the pickup location to the delivery location.
- Time based calculations:
  - Travel time between customers is  $\lfloor (\text{distance} / \text{speed}) \rfloor$
  - Pickup and delivery order pair is done by the same vehicle.
  - A pickup is done before the delivery.

### Parameters

#### Pick & deliver

Both implementations use the following same parameters:

Column	Type	Default	Description
<b>orders_sql</b>	TEXT		<b>Pick &amp; Deliver Orders SQL</b> query containing the orders to be processed.
<b>vehicles_sql</b>	TEXT		<b>Pick &amp; Deliver Vehicles SQL</b> query containing the vehicles to be used.
<b>factor</b>	NUMERIC	1	(Optional) Travel time multiplier. See <b>Factor Handling</b>
<b>max_cycles</b>	INTEGER	10	(Optional) Maximum number of cycles to perform on the optimization.
<b>initial_sol</b>	INTEGER	4	(Optional) Initial solution to be used. <ul style="list-style-type: none"> <li>• 1 One order per truck</li> <li>• 2 Push front order.</li> <li>• 3 Push back order.</li> <li>• 4 Optimize insert.</li> <li>• 5 Push back order that allows more orders to be inserted at the back</li> <li>• 6 Push front order that allows more orders to be inserted at the front</li> </ul>

The non euclidean implementation, additionally has:

Column	Type	Description
<b>matrix_sql</b>	TEXT	<b>Pick &amp; Deliver Matrix SQL</b> query containing the distance or travel times.

### Inner Queries

- **Pick & Deliver Orders SQL**
- **Pick & Deliver Vehicles SQL**
- **Pick & Deliver Matrix SQL**

### return columns

- **Description of return columns**
- **Description of the return columns for Euclidean version**

#### Pick & Deliver Orders SQL

In general, the columns for the orders SQL is the same in both implementation of pick and delivery:

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the pick-delivery order pair.
<b>demand</b>	ANY-NUMERICAL		Number of units in the order
<b>p_open</b>	ANY-NUMERICAL		The time, relative to 0, when the pickup location opens.
<b>p_close</b>	ANY-NUMERICAL		The time, relative to 0, when the pickup location closes.

Column	Type	Default	Description
<b>d_service</b>	ANY-NUMERICAL	0	The duration of the loading at the pickup location.
<b>d_open</b>	ANY-NUMERICAL		The time, relative to 0, when the delivery location opens.
<b>d_close</b>	ANY-NUMERICAL		The time, relative to 0, when the delivery location closes.
<b>d_service</b>	ANY-NUMERICAL	0	The duration of the loading at the delivery location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Description
<b>p_node_id</b>	ANY-INTEGER	The node identifier of the pickup, must match a node identifier in the matrix table.
<b>d_node_id</b>	ANY-INTEGER	The node identifier of the delivery, must match a node identifier in the matrix table.

For the euclidean implementation, pick up and delivery  $((x,y))$  locations are needed:

Column	Type	Description
<b>p_x</b>	ANY-NUMERICAL	$x$ value of the pick up location
<b>p_y</b>	ANY-NUMERICAL	$y$ value of the pick up location
<b>d_x</b>	ANY-NUMERICAL	$x$ value of the delivery location
<b>d_y</b>	ANY-NUMERICAL	$y$ value of the delivery location

#### Pick & Deliver Vehicles SQL

In general, the columns for the vehicles\_sql is the same in both implementation of pick and delivery:

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the pick-delivery order pair.
<b>capacity</b>	ANY-NUMERICAL		Number of units in the order
<b>speed</b>	ANY-NUMERICAL	1	Average speed of the vehicle.
<b>start_open</b>	ANY-NUMERICAL		The time, relative to 0, when the starting location opens.
<b>start_close</b>	ANY-NUMERICAL		The time, relative to 0, when the starting location closes.
<b>start_service</b>	ANY-NUMERICAL	0	The duration of the loading at the starting location.
<b>end_open</b>	ANY-NUMERICAL	<i>start_open</i>	The time, relative to 0, when the ending location opens.
<b>end_close</b>	ANY-NUMERICAL	<i>start_close</i>	The time, relative to 0, when the ending location closes.
<b>end_service</b>	ANY-NUMERICAL	<i>start_service</i>	The duration of the loading at the ending location.

For the non euclidean implementation, the starting and ending identifiers are needed:

Column	Type	Default	Description
<b>start_node_id</b>	ANY-INTEGER		The node identifier of the starting location, must match a node identifier in the matrix table.
<b>end_node_id</b>	ANY-INTEGER	<i>start_node_id</i>	The node identifier of the ending location, must match a node identifier in the matrix table.

For the euclidean implementation, starting and ending  $((x,y))$  locations are needed:

Column	Type	Default	Description
<b>start_x</b>	ANY-NUMERICAL		$x$ value of the coordinate of the starting location.
<b>start_y</b>	ANY-NUMERICAL		$y$ value of the coordinate of the starting location.
<b>end_x</b>	ANY-NUMERICAL	<i>start_x</i>	$x$ value of the coordinate of the ending location.
<b>end_y</b>	ANY-NUMERICAL	<i>start_y</i>	$y$ value of the coordinate of the ending location.

#### Pick & Deliver Matrix SQL



#### Warning

TODO

#### Results

Description of the result (TODO Disussion: Euclidean & Matrix)



RETURNS SET OF  
 (seq, vehicle\_seq, vehicle\_id, stop\_seq, stop\_type,  
 travel\_time, arrival\_time, wait\_time, service\_time, departure\_time)  
 UNION  
 (summary row)

Column	Type	Description
<b>seq</b>	INTEGER	Sequential value starting from <b>1</b> .
<b>vehicle_seq</b>	INTEGER	Sequential value starting from <b>1</b> for current vehicles. The $(n_{th})$ vehicle in the solution.
<b>vehicle_id</b>	BIGINT	Current vehicle identifier.
<b>stop_seq</b>	INTEGER	Sequential value starting from <b>1</b> for the stops made by the current vehicle. The $(m_{th})$ stop of the current vehicle.
<b>stop_type</b>	INTEGER	Kind of stop location the vehicle is at: <ul style="list-style-type: none"> <li>• <b>1</b>: Starting location</li> <li>• <b>2</b>: Pickup location</li> <li>• <b>3</b>: Delivery location</li> <li>• <b>6</b>: Ending location</li> </ul>
<b>order_id</b>	BIGINT	Pickup-Delivery order pair identifier. <ul style="list-style-type: none"> <li>• <b>-1</b>: When no order is involved on the current stop location.</li> </ul>
<b>cargo</b>	FLOAT	Cargo units of the vehicle when leaving the stop.
<b>travel_time</b>	FLOAT	Travel time from previous <code>stop_seq</code> to current <code>stop_seq</code> . <ul style="list-style-type: none"> <li>• <b>0</b> When <code>stop_type = 1</code></li> </ul>
<b>arrival_time</b>	FLOAT	Previous <code>departure_time</code> plus current <code>travel_time</code> .
<b>wait_time</b>	FLOAT	Time spent waiting for current <code>location</code> to open.
<b>service_time</b>	FLOAT	Service time at current <code>location</code> .
<b>departure_time</b>	FLOAT	$(arrival\_time + wait\_time + service\_time)$ . <ul style="list-style-type: none"> <li>• When <code>stop_type = 6</code> has the <code>total_time</code> used for the current vehicle.</li> </ul>

#### Summary Row



#### Warning

TODO: Review the summary

Column	Type	Description
<b>seq</b>	INTEGER	Continues the Sequential value
<b>vehicle_seq</b>	INTEGER	<b>-2</b> to indicate is a summary row
<b>vehicle_id</b>	BIGINT	<i>Total Capacity Violations</i> in the solution.
<b>stop_seq</b>	INTEGER	<i>Total Time Window Violations</i> in the solution.
<b>stop_type</b>	INTEGER	<b>-1</b>
<b>order_id</b>	BIGINT	<b>-1</b>
<b>cargo</b>	FLOAT	<b>-1</b>
<b>travel_time</b>	FLOAT	<i>total_travel_time</i> The sum of all the <code>travel_time</code>
<b>arrival_time</b>	FLOAT	<b>-1</b>
<b>wait_time</b>	FLOAT	<i>total_waiting_time</i> The sum of all the <code>wait_time</code>
<b>service_time</b>	FLOAT	<i>total_service_time</i> The sum of all the <code>service_time</code>
<b>departure_time</b>	FLOAT	<i>total_solution_time</i> = $(total\_travel\_time + total\_wait\_time + total\_service\_time)$ .

#### Description of the result (TODO Disussion: Euclidean & Matrix)

RETURNS SET OF  
 (seq, vehicle\_seq, vehicle\_id, stop\_seq, stop\_type,  
 travel\_time, arrival\_time, wait\_time, service\_time, departure\_time)  
 UNION  
 (summary row)

Column	Type	Description
<b>seq</b>	INTEGER	Sequential value starting from <b>1</b> .
<b>vehicle_seq</b>	INTEGER	Sequential value starting from <b>1</b> for current vehicles. The $(n_{th})$ vehicle in the solution.
<b>vehicle_id</b>	BIGINT	Current vehicle identifier.

Column	Type	Description
<b>stop_seq</b>	INTEGER	Sequential value starting from <b>1</b> for the stops made by the current vehicle. The $m_{th}$ stop of the current vehicle.
<b>stop_type</b>	INTEGER	Kind of stop location the vehicle is at: <ul style="list-style-type: none"> <li>1: Starting location</li> <li>2: Pickup location</li> <li>3: Delivery location</li> <li>6: Ending location</li> </ul>
<b>order_id</b>	BIGINT	Pickup-Delivery order pair identifier. <ul style="list-style-type: none"> <li>-1: When no order is involved on the current stop location.</li> </ul>
<b>cargo</b>	FLOAT	Cargo units of the vehicle when leaving the stop.
<b>travel_time</b>	FLOAT	Travel time from previous <code>stop_seq</code> to current <code>stop_seq</code> . <ul style="list-style-type: none"> <li>0 When <code>stop_type = 1</code></li> </ul>
<b>arrival_time</b>	FLOAT	Previous <code>departure_time</code> plus current <code>travel_time</code> .
<b>wait_time</b>	FLOAT	Time spent waiting for current <code>location</code> to open.
<b>service_time</b>	FLOAT	Service time at current <code>location</code> .
<b>departure_time</b>	FLOAT	$\text{(arrival\_time + wait\_time + service\_time)}$ . <ul style="list-style-type: none"> <li>When <code>stop_type = 6</code> has the <code>total_time</code> used for the current vehicle.</li> </ul>

### Summary Row



#### Warning

TODO: Review the summary

Column	Type	Description
<b>seq</b>	INTEGER	Continues the Sequential value
<b>vehicle_seq</b>	INTEGER	-2 to indicate is a summary row
<b>vehicle_id</b>	BIGINT	<i>Total Capacity Violations</i> in the solution.
<b>stop_seq</b>	INTEGER	<i>Total Time Window Violations</i> in the solution.
<b>stop_type</b>	INTEGER	-1
<b>order_id</b>	BIGINT	-1
<b>cargo</b>	FLOAT	-1
<b>travel_time</b>	FLOAT	<i>total_travel_time</i> The sum of all the <code>travel_time</code>
<b>arrival_time</b>	FLOAT	-1
<b>wait_time</b>	FLOAT	<i>total_waiting_time</i> The sum of all the <code>wait_time</code>
<b>service_time</b>	FLOAT	<i>total_service_time</i> The sum of all the <code>service_time</code>
<b>departure_time</b>	FLOAT	<i>total_solution_time</i> = $\text{(total\_travel\_time + total\_wait\_time + total\_service\_time)}$ .

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Handling Parameters

To define a problem, several considerations have to be done, to get consistent results. This section gives an insight of how parameters are to be considered.

- Capacity and Demand Units Handling
- Locations
- Time Handling
- Factor Handling

#### Capacity and Demand Units Handling

The *capacity* of a vehicle, can be measured in:

- Volume units like  $(m^3)$ .
- Area units like  $(m^2)$  (when no stacking is allowed).

- Weight units like  $(kg)$ .
- Number of boxes that fit in the vehicle.
- Number of seats in the vehicle

The *demand* request of the pickup-deliver orders must use the same units as the units used in the vehicle's *capacity*.

To handle problems like: 10 (equal dimension) boxes of apples and 5 kg of feathers that are to be transported (not packed in boxes).

If the vehicle's *capacity* is measured by *boxes*, a conversion of *kg of feathers* to *equivalent number of boxes* is needed. If the vehicle's *capacity* is measured by *kg*, a conversion of *box of apples* to *equivalent number of kg* is needed.

Showing how the 2 possible conversions can be done

Let:  $f\_boxes$ : number of boxes that would be used for 1 kg of feathers.  $a\_weight$ : weight of 1 box of apples.

Capacity Units	apples	feathers
boxes	10	$(5 * f\_boxes)$
kg	$(10 * a\_weight)$	5

#### Locations

- When using the Euclidean signatures:
  - The vehicles have  $((x, y))$  pairs for start and ending locations.
  - The orders Have  $((x, y))$  pairs for pickup and delivery locations.
- When using a matrix:
  - The vehicles have identifiers for the start and ending locations.
  - The orders have identifiers for the pickup and delivery locations.
  - All the identifiers are indices to the given matrix.

#### Time Handling

The times are relative to 0

Suppose that a vehicle's driver starts the shift at 9:00 am and ends the shift at 4:30 pm and the service time duration is 10 minutes with 30 seconds.

All time units have to be converted

Meaning of 0	time units	9:00 am	4:30 pm	10 min 30 secs
0:00 am	hours	9	16.5	$(10.5 / 60 = 0.175)$
9:00 am	hours	0	7.5	$(10.5 / 60 = 0.175)$
0:00 am	minutes	$(9*60 = 54)$	$(16.5*60 = 990)$	10.5
9:00 am	minutes	0	$(7.5*60 = 540)$	10.5

#### Factor Handling



#### Warning

TODO

#### See Also

- [https://en.wikipedia.org/wiki/Vehicle\\_routing\\_problem](https://en.wikipedia.org/wiki/Vehicle_routing_problem)
- The queries use the **Sample Data** network.

#### Indices and tables

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

#### Not classified

- **pgr\_bellmanFord - Experimental**
- **pgr\_binaryBreadthFirstSearch - Experimental**
- **pgr\_breadthFirstSearch - Experimental**
- **pgr\_dagShortestPath - Experimental**
- **pgr\_edwardMoore - Experimental**
- **pgr\_stoerWagner - Experimental**

- **pgr\_topologicalSort - Experimental**
- **pgr\_transitiveClosure - Experimental**
- **pgr\_turnRestrictedPath - Experimental**

pgr\_bellmanFord - Experimental

`pgr_bellmanFord` — Returns the shortest path(s) using Bellman-Ford algorithm. In particular, the Bellman-Ford algorithm implemented by Boost.Graph.



Boost Graph Inside



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

#### Availability

- Version 3.0.0
  - New **experimental** function

#### Support

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

#### Description

Bellman-Ford's algorithm, is named after Richard Bellman and Lester Ford, who first published it in 1958 and 1956, respectively. It is a graph search algorithm that computes shortest paths from a starting vertex (`start_vid`) to an ending vertex (`end_vid`) in a graph where some of the edge weights may be negative number. Though it is more versatile, it is slower than Dijkstra's algorithm/ This implementation can be used with a directed graph and an undirected graph.

The main characteristics are:

- Process is valid for edges with both positive and negative edge weights.
- Values are returned when there is a path.
  - When the start vertex and the end vertex are the same, there is no path. The `agg_cost` would be 0.
  - When the start vertex and the end vertex are different, and there exists a path between them without having a *negative cycle*. The `agg_cost` would be some finite value denoting the shortest distance between them.
  - When the start vertex and the end vertex are different, and there exists a path between them, but it contains a *negative cycle*. In such case, `agg_cost` for those vertices keep on decreasing furthermore, Hence `agg_cost` can't be defined for them.
  - When the start vertex and the end vertex are different, and there is no path. The `agg_cost` is  $-\infty$ .
- 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:  $O(|\text{start\_vids}| * (V * E))$

## Signatures

## Summary

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

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

## Using defaults

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

## Example:

From vertex  $\{(2)\}$  to vertex  $\{(3)\}$  on a **directed** graph

```
SELECT * FROM pgr_bellmanFord(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3
);
```

```
seq | path_seq | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 4 | 1 | 0
2 | 2 | 5 | 8 | 1 | 1
3 | 3 | 6 | 9 | 1 | 2
4 | 4 | 9 | 16 | 1 | 3
5 | 5 | 4 | 3 | 1 | 4
6 | 6 | 3 | -1 | 0 | 5
```

(6 rows)

## One to One

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

## Example:

From vertex  $\{(2)\}$  to vertex  $\{(3)\}$  on an **undirected** graph

```
SELECT * FROM pgr_bellmanFord(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 3,
  FALSE
);
```

```
seq | path_seq | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 2 | 1 | 0
2 | 2 | 3 | -1 | 0 | 1
```

(2 rows)

## One to many

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

## Example:

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

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

#### Many to One

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

#### Example:

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

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

#### Many to Many

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

#### Example:

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

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

#### Parameters

##### Description of the parameters of the signatures

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

Parameter	Type	Default	Description
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When true Graph is considered <i>Directed</i></li> <li>When false the graph is considered as <i>Undirected</i>.</li> </ul>

#### Inner Query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Results Columns

Returns set of (seq, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li><b>One to Many</b></li> <li><b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

#### See Also

- [https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford\\_algorithm](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)
- The queries use the **Sample Data** network.

#### Indices and tables

- Index**
- Search Page**

#### pgr\_binaryBreadthFirstSearch - Experimental

`pgr_binaryBreadthFirstSearch` — Returns the shortest path(s) in a binary graph. Any graph whose edge-weights belongs to the set {0,X}, where 'X' is any non-negative real integer, is termed as a 'binary graph'.



## Boost Graph Inside



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

### Availability

- To-be experimental on v3.0.0

### Description

It is well-known that the shortest paths between a single source and all other vertices can be found using Breadth First Search in  $\mathcal{O}(|E|)$  in an unweighted graph, i.e. the distance is the minimal number of edges that you need to traverse from the source to another vertex. We can interpret such a graph also as a weighted graph, where every edge has the weight 1. If not all edges in graph have the same weight, that we need a more general algorithm, like Dijkstra's Algorithm which runs in  $\mathcal{O}(|E|\log|V|)$  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  $\mathcal{O}(|E|)$ , 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:  $\mathcal{O}(|start\_vids| * |E|)$

### Signatures

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

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



**Example:**

From vertex  $\{2\}$  to vertex  $\{3\}$  on a **directed** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
  'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
  2, 3
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	4	0	0
2	2	5	8	1	0
3	3	6	9	1	1
4	4	9	16	0	2
5	5	4	3	0	2
6	6	3	-1	0	2

(6 rows)

**One to One**

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

**Example:**

From vertex  $\{2\}$  to vertex  $\{3\}$  on an **undirected** binary graph

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
  'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
  2, 3,
  FALSE
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	2	2	1	0
2	2	3	-1	0	1

(2 rows)

**One to many**

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

**Example:**

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

```
SELECT * FROM pgr_binaryBreadthFirstSearch(
  'SELECT id, source, target, road_work as cost FROM roadworks',
  2, ARRAY[3,5],
  FALSE
);
```

seq	path_seq	end_vid	node	edge	cost	agg_cost
1	1	3	2	4	0	0
2	2	3	5	8	1	0
3	3	3	6	5	1	1
4	4	3	3	-1	0	2
5	1	5	2	4	0	0
6	2	5	5	-1	0	0

(6 rows)

**Many to One**

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

**Example:**

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

```

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

```

### Many to Many

```

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

```

### Example:

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

```

SELECT * FROM pgr_binaryBreadthFirstSearch(
'SELECT id, source, target, road_work as cost, reverse_road_work as reverse_cost FROM roadworks',
ARRAY[2,11], ARRAY[3,5],
FALSE
);
seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 3 | 2 | 2 | 1 | 0
2 | 2 | 2 | 3 | 3 | -1 | 0 | 1
3 | 1 | 2 | 5 | 2 | 4 | 0 | 0
4 | 2 | 2 | 5 | 5 | -1 | 0 | 0
5 | 1 | 11 | 3 | 11 | 13 | 1 | 0
6 | 2 | 11 | 3 | 12 | 15 | 0 | 1
7 | 3 | 11 | 3 | 9 | 16 | 0 | 1
8 | 4 | 11 | 3 | 4 | 3 | 0 | 1
9 | 5 | 11 | 3 | 3 | -1 | 0 | 1
10 | 1 | 11 | 5 | 11 | 12 | 0 | 0
11 | 2 | 11 | 5 | 10 | 10 | 1 | 0
12 | 3 | 11 | 5 | 5 | -1 | 0 | 1
(12 rows)

```

### Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source, target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source, target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target, source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target, source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGERS:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

**Return Columns**

Returns set of (seq, path\_id, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_id</b>	INT	Path identifier. Has value <b>1</b> for the first of a path. Used when there are multiple paths for the same <code>start_vid</code> to <code>end_vid</code> combination.
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li>• <b>Many to One</b></li> <li>• <b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li>• <b>One to Many</b></li> <li>• <b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <code>start_vid</code> to <code>end_vid</code> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <code>node</code> to the next node in the path sequence. <code>-1</code> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

**Example Data**

This type of data is used on the examples of this page.

Edwards-Moore Algorithm is best applied when trying to answer queries such as the following: **“Find the path with the minimum number from Source to Destination”** Here: \* *Source* = Source Vertex, *Destination* = Any arbitrary destination vertex \* *X* is an event/property \* Each edge in the graph is either “*X*” or “*Not X*” .

Example: “Find the path with the minimum number of road works from Source to Destination”

Here, a road under work (aka **road works**) means that part of the road is occupied for construction work/maintenance.

Here: \* Edge (*u* , *v* ) has weight = 0 if no road work is ongoing on the road from *u* to *v*. \* Edge (*u*, *v*) has weight = 1 if road work is ongoing on the road from *u* to *v*.

Then, upon running the algorithm, we obtain the path with the minimum number of road works from the given source and destination.

Thus, the queries used in the previous section can be interpreted in this manner.

**Table Data**

The queries in the previous sections use the table ‘roadworks’. The data of the table:

```

DROP TABLE IF EXISTS roadworks CASCADE;
NOTICE: table "roadworks" does not exist, skipping
DROP TABLE
CREATE table roadworks (
  id BIGINT not null primary key,
  source BIGINT,
  target BIGINT,
  road_work FLOAT,
  reverse_road_work FLOAT
);
CREATE TABLE
INSERT INTO roadworks(
  id, source, target, road_work, reverse_road_work) VALUES
(1, 1, 2, 0, 0),
(2, 2, 3, -1, 1),
(3, 3, 4, -1, 0),
(4, 2, 5, 0, 0),
(5, 3, 6, 1, -1),
(6, 7, 8, 1, 1),
(7, 8, 5, 0, 0),
(8, 5, 6, 1, 1),
(9, 6, 9, 1, 1),
(10, 5, 10, 1, 1),
(11, 6, 11, 1, -1),
(12, 10, 11, 0, -1),
(13, 11, 12, 1, -1),
(14, 10, 13, 1, 1),
(15, 9, 12, 0, 0),
(16, 4, 9, 0, 0),
(17, 14, 15, 0, 0),
(18, 16, 17, 0, 0);
INSERT 0 18

```

#### See Also

- [https://cp-algorithms.com/graph/01\\_bfs.html](https://cp-algorithms.com/graph/01_bfs.html)
- [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm#Specialized\\_variants](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Specialized_variants)

#### Indices and tables

- [Index](#)
- [Search Page](#)

pgr\_breadthFirstSearch - Experimental

`pgr_breadthFirstSearch` — Returns the traversal order(s) using Breadth First Search algorithm.



Boost Graph Inside



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgrTap 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 community.

- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

## Availability

## Description

Provides the Breadth First Search traversal order from a root vertex to a particular depth.

## The main Characteristics are:

- The implementation will work on any type of graph.
- Provides the Breadth First Search traversal order from a source node to a target depth level
- Breadth First Search Running time:  $O(E + V)$

## Signatures

```
pgr_breadthFirstSearch(Edges SQL, Root vid [, max_depth] [, directed])
pgr_breadthFirstSearch(Edges SQL, Root vids [, max_depth] [, directed])
```

RETURNS SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

## Single Vertex

```
pgr_breadthFirstSearch(Edges SQL, Root vid [, max_depth] [, directed])
```

RETURNS SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

## Example:

The Breadth First Search traversal with root vertex  $\{2\}$

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  2
);
```

seq	depth	start_vid	node	edge	cost	agg_cost
1	0	2	2	-1	0	0
2	1	2	1	1	1	1
3	1	2	5	4	1	1
4	2	2	8	7	1	2
5	2	2	6	8	1	2
6	2	2	10	10	1	2
7	3	2	7	6	1	3
8	3	2	9	9	1	3
9	3	2	11	11	1	3
10	3	2	13	14	1	3
11	4	2	12	15	1	4
12	4	2	4	16	1	4
13	5	2	3	3	1	5

## Multiple Vertices

```
pgr_breadthFirstSearch(Edges SQL, Root vids [, max_depth] [, directed])
```

RETURNS SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

## Example:

The Breadth First Search traversal starting on vertices  $\{11, 12\}$  with  $(depth \leq 2)$

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  ARRAY[11,12], max_depth := 2
);
```

seq	depth	start_vid	node	edge	cost	agg_cost
1	0	11	11	-1	0	0
2	1	11	12	13	1	1
3	2	11	9	15	1	2
4	0	12	12	-1	0	0
5	1	12	9	15	1	1
6	2	12	6	9	1	2
7	2	12	4	16	1	2

## Parameters

Parameter	Type	Description
<b>Edges SQL</b>	TEXT	SQL query described in <b>Inner query</b> .
<b>Root vid</b>	BIGINT	Identifier of the root vertex of the tree. <ul style="list-style-type: none"><li>Used on <b>Single Vertex</b>.</li></ul>
<b>Root vids</b>	ARRAY[ANY-INTEGER]	Array of identifiers of the root vertices. <ul style="list-style-type: none"><li>Used on <b>Multiple Vertices</b>.</li><li>For optimization purposes, any duplicated value is ignored.</li></ul>

## Optional Parameters

Parameter	Type	Default	Description
<b>max_depth</b>	BIGINT	\ (9223372036854775807)	Upper limit for depth of node in the tree <ul style="list-style-type: none"><li>When value is <b>Negative</b> then <b>throws error</b></li></ul>
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"><li>When <b>true</b> Graph is considered <i>Directed</i></li><li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li></ul>

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICALS:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Result Columns

Returns SET OF (seq, depth, start\_vid, node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	BIGINT	Sequential value starting from \{1\}.
<b>depth</b>	BIGINT	Depth of the <b>node</b> . <ul style="list-style-type: none"><li>\{0\} when <b>node</b> = <b>start_vid</b>.</li></ul>
<b>start_vid</b>	BIGINT	Identifier of the root vertex. <ul style="list-style-type: none"><li>In <i>Multiple Vertices</i> results are in ascending order.</li></ul>
<b>node</b>	BIGINT	Identifier of <b>node</b> reached using <b>edge</b> .
<b>edge</b>	BIGINT	Identifier of the <b>edge</b> used to arrive to <b>node</b> . <ul style="list-style-type: none"><li>\{-1\} when <b>node</b> = <b>start_vid</b>.</li></ul>
<b>cost</b>	FLOAT	Cost to traverse <b>edge</b> .
<b>agg_cost</b>	FLOAT	Aggregate cost from <b>start_vid</b> to <b>node</b> .

## Additional Examples

### Undirected Graph

#### Example:

The Breadth First Search traverses starting on vertices \{\{11, 12\}\} with \{depth <= 2\} as well as considering the graph to be undirected.

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  ARRAY[11,12], max_depth := 2, directed := false
);
```

seq	depth	start_vid	node	edge	cost	agg_cost
1	0	11	11	-1	0	0
2	1	11	6	11	1	1
3	1	11	10	12	1	1
4	1	11	12	13	1	1
5	2	11	3	5	1	2
6	2	11	5	8	1	2
7	2	11	9	9	1	2
8	2	11	13	14	1	2
9	0	12	12	-1	0	0
10	1	12	11	13	1	1
11	1	12	9	15	1	1
12	2	12	6	11	1	2
13	2	12	10	12	1	2
14	2	12	4	16	1	2

(14 rows)

## Vertex Out Of Graph

### Example:

The output of the function when a vertex not present in the graph is passed as a parameter.

```
SELECT * FROM pgr_breadthFirstSearch(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
  -10
);
```

seq	depth	start_vid	node	edge	cost	agg_cost
-----	-------	-----------	------	------	------	----------

(0 rows)

## See Also

- The queries use the **Sample Data** network.
- **Boost: Breadth First Search algorithm documentation**
- **Wikipedia: Breadth First Search algorithm**

## Indices and tables

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

## pgr\_dagShortestPath - Experimental

`pgr_dagShortestPath` — Returns the shortest path(s) for weighted directed acyclic graphs(DAG). In particular, the DAG shortest paths algorithm implemented by Boost.Graph.



## Boost Graph Inside



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

### Availability

- Version 3.0.0
  - New **experimental** function

### Support

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

### Description

Shortest Path for Directed Acyclic Graph(DAG) is a graph search algorithm that solves the shortest path problem for weighted directed acyclic graph, producing a shortest path from a starting vertex (`start_vid`) to an ending vertex (`end_vid`).

This implementation can only be used with a **directed** graph with no cycles i.e. directed acyclic graph.

The algorithm relies on topological sorting the dag to impose a linear ordering on the vertices, and thus is more efficient for DAG's than either the Dijkstra or Bellman-Ford algorithm.

The main characteristics are:

- Process is valid for weighted directed acyclic graphs only. otherwise it will throw warnings.
- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The `agg_cost` the non included values ( $v, v$ ) is  $0$
  - When the starting vertex and ending vertex are the different and there is no path:
    - The `agg_cost` the non included values ( $u, v$ ) is  $\infty$
- For optimization purposes, any duplicated value in the `start_vids` or `end_vids` are ignored.
- The returned values are ordered:
  - `start_vid` ascending
  - `end_vid` ascending
- Running time:  $O(|start\_vids| * (V + E))$

### Signatures

### Summary

```
pgr_dagShortestPath(edges_sql, from_vid, to_vid)
pgr_dagShortestPath(edges_sql, from_vid, to_vids)
pgr_dagShortestPath(edges_sql, from_vids, to_vid)
pgr_dagShortestPath(edges_sql, from_vids, to_vids)
```

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

### One to One

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

### Example:

From vertex  $\{(1)\}$  to vertex  $\{(6)\}$



```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edge_table',
  1, 6
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	1	1	1	0
2	2	2	4	1	1
3	3	5	8	1	2
4	4	6	-1	0	3

(4 rows)

#### One to Many

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

#### Example:

From vertex  $\{(1)\}$  to vertices  $\{(5, 6)\}$

```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edge_table',
  1, ARRAY[5,6]
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	1	1	1	0
2	2	2	4	1	1
3	3	5	-1	0	2
4	1	1	1	1	0
5	2	2	4	1	1
6	3	5	8	1	2
7	4	6	-1	0	3

(7 rows)

#### Many to One

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

#### Example:

From vertices  $\{(1, 3)\}$  to vertex  $\{(6)\}$

```
SELECT * FROM pgr_dagShortestPath(
  'SELECT id, source, target, cost FROM edge_table',
  ARRAY[1,3], 6
);
```

seq	path_seq	node	edge	cost	agg_cost
1	1	1	1	1	0
2	2	2	4	1	1
3	3	5	8	1	2
4	4	6	-1	0	3
5	1	3	5	1	0
6	2	6	-1	0	1

(6 rows)

#### Many to Many

```
pgr_dagShortestPath(edges_sql, from_vids, to_vids)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
OR EMPTY SET
```

#### Example:

From vertices  $\{(1, 4)\}$  to vertices  $\{(12, 6)\}$

```

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

```

## Parameters

### Description of the parameters of the signatures

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		SQL query as described above.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.

### Inner Query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

### Results Columns

Returns set of (seq, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li><b>One to Many</b></li> <li><b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <i>start_vid</i> to <i>end_vid</i> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <i>node</i> to the next node in the path sequence. <b>1</b> for the last node of the path.

Column	Type	Description
<code>cost</code>	FLOAT	Cost to traverse from <code>node</code> using <code>edge</code> to the next node in the path sequence.
<code>agg_cost</code>	FLOAT	Aggregate cost from <code>start_v</code> to <code>node</code> .

#### See Also

- [https://en.wikipedia.org/wiki/Topological\\_sorting](https://en.wikipedia.org/wiki/Topological_sorting)
- The queries use the **Sample Data** network.

#### Indices and tables

- [Index](#)
- [Search Page](#)

`pgr_edwardMoore` - Experimental

`pgr_edwardMoore` — Returns the shortest path(s) using Edward-Moore algorithm. Edward Moore's Algorithm is an improvement of the Bellman-Ford Algorithm.



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

#### Availability

- Version 3.0.0
  - New **experimental** function

#### Description

Edward Moore's Algorithm is an improvement of the Bellman-Ford Algorithm. It can compute the shortest paths from a single source vertex to all other vertices in a weighted directed graph. The main difference between Edward Moore's Algorithm and Bellman Ford's Algorithm lies in the run time.

The worst-case running time of the algorithm is  $O(|V| * |E|)$  similar to the time complexity of Bellman-Ford algorithm. However, experiments suggest that this algorithm has an average running time complexity of  $O(|E|)$  for random graphs. This is significantly faster in terms of computation speed.

Thus, the algorithm is at-best, significantly faster than Bellman-Ford algorithm and is at-worst, as good as Bellman-Ford algorithm

The main characteristics are:

- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The `agg_cost` the non included values  $(v, v)$  is 0
  - When the starting vertex and ending vertex are the different and there is no path:
    - The `agg_cost` the non included values  $(u, v)$  is  $(-\infty)$

- For optimization purposes, any duplicated value in the `start_vids` or `end_vids` are ignored.
- The returned values are ordered:
  - `start_vid` ascending
  - `end_vid` ascending
- Running time: - Worst case:  $O(|V| * |E|)$  - Average case:  $O(|E|)$

### Signatures

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

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

### Example:

From vertex `(2)` to vertex `(3)` on a **directed** graph

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

### One to One

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

### Example:

From vertex `(2)` to vertex `(3)` on an **undirected** graph

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

### One to many

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

### Example:

From vertex `(2)` to vertices `({3, 5})` on an **undirected** graph

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

### Many to One

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

### Example:

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

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

### Many to Many

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

### Example:

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

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

### Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		Inner SQL query as described below.
<b>start_vid</b>	BIGINT		Identifier of the starting vertex of the path.
<b>start_vids</b>	ARRAY[BIGINT]		Array of identifiers of starting vertices.
<b>end_vid</b>	BIGINT		Identifier of the ending vertex of the path.
<b>end_vids</b>	ARRAY[BIGINT]		Array of identifiers of ending vertices.

Parameter	Type	Default	Description
<b>directed</b>	BOOLEAN	true	<ul style="list-style-type: none"> <li>When <b>true</b> Graph is considered <i>Directed</i></li> <li>When <b>false</b> the graph is considered as <i>Undirected</i>.</li> </ul>

#### Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGERS		Identifier of the edge.
<b>source</b>	ANY-INTEGERS		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGERS		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

#### ANY-INTEGERS:

SMALLINT, INTEGER, BIGINT

#### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

#### Return Columns

Returns set of (seq, path\_id, path\_seq [, start\_vid] [, end\_vid], node, edge, cost, agg\_cost)

Column	Type	Description
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>path_id</b>	INT	Path identifier. Has value <b>1</b> for the first of a path. Used when there are multiple paths for the same <i>start_vid</i> to <i>end_vid</i> combination.
<b>path_seq</b>	INT	Relative position in the path. Has value <b>1</b> for the beginning of a path.
<b>start_vid</b>	BIGINT	Identifier of the starting vertex. Returned when multiple starting vertices are in the query. <ul style="list-style-type: none"> <li><b>Many to One</b></li> <li><b>Many to Many</b></li> </ul>
<b>end_vid</b>	BIGINT	Identifier of the ending vertex. Returned when multiple ending vertices are in the query. <ul style="list-style-type: none"> <li><b>One to Many</b></li> <li><b>Many to Many</b></li> </ul>
<b>node</b>	BIGINT	Identifier of the node in the path from <i>start_vid</i> to <i>end_vid</i> .
<b>edge</b>	BIGINT	Identifier of the edge used to go from <i>node</i> to the next node in the path sequence. <b>-1</b> for the last node of the path.
<b>cost</b>	FLOAT	Cost to traverse from <i>node</i> using <i>edge</i> to the next node in the path sequence.
<b>agg_cost</b>	FLOAT	Aggregate cost from <i>start_v</i> to <i>node</i> .

#### Example Application

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

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

#### Examples:

For queries marked as *directed* with *cost* and *reverse\_cost* columns

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

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

```
5 | 5 | 4 | 3 | 1 | 4
6 | 6 | 3 | -1 | 0 | 5
```

(6 rows)

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, 5
```

);

```
seq | path_seq | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 4 | 1 | 0
2 | 2 | 5 | -1 | 0 | 1
```

(2 rows)

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  2, ARRAY[3,5]
```

);

```
seq | path_seq | end_vid | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 3 | 2 | 4 | 1 | 0
2 | 2 | 3 | 5 | 8 | 1 | 1
3 | 3 | 3 | 6 | 9 | 1 | 2
4 | 4 | 3 | 9 | 16 | 1 | 3
5 | 5 | 3 | 4 | 3 | 1 | 4
6 | 6 | 3 | 3 | -1 | 0 | 5
7 | 1 | 5 | 2 | 4 | 1 | 0
8 | 2 | 5 | 5 | -1 | 0 | 1
```

(8 rows)

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  11, 3
```

);

```
seq | path_seq | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----
1 | 1 | 11 | 13 | 1 | 0
2 | 2 | 12 | 15 | 1 | 1
3 | 3 | 9 | 16 | 1 | 2
4 | 4 | 4 | 3 | 1 | 3
5 | 5 | 3 | -1 | 0 | 4
```

(5 rows)

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  11, 5
```

);

```
seq | path_seq | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----
1 | 1 | 11 | 13 | 1 | 0
2 | 2 | 12 | 15 | 1 | 1
3 | 3 | 9 | 9 | 1 | 2
4 | 4 | 6 | 8 | 1 | 3
5 | 5 | 5 | -1 | 0 | 4
```

(5 rows)

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[2,11], 5
```

);

```
seq | path_seq | start_vid | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 2 | 4 | 1 | 0
2 | 2 | 2 | 5 | -1 | 0 | 1
3 | 1 | 11 | 11 | 13 | 1 | 0
4 | 2 | 11 | 12 | 15 | 1 | 1
5 | 3 | 11 | 9 | 9 | 1 | 2
6 | 4 | 11 | 6 | 8 | 1 | 3
7 | 5 | 11 | 5 | -1 | 0 | 4
```

(7 rows)

```
SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost, reverse_cost FROM edge_table',
  ARRAY[2, 11], ARRAY[3,5]
```

);

```
seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
```

```
-----+-----+-----+-----+-----+-----+-----+-----
1 | 1 | 2 | 3 | 2 | 4 | 1 | 0
2 | 2 | 2 | 3 | 5 | 8 | 1 | 1
3 | 3 | 2 | 3 | 6 | 9 | 1 | 2
4 | 4 | 2 | 3 | 9 | 16 | 1 | 3
5 | 5 | 2 | 3 | 4 | 3 | 1 | 4
6 | 6 | 2 | 3 | 3 | -1 | 0 | 5
7 | 1 | 2 | 5 | 2 | 4 | 1 | 0
8 | 2 | 2 | 5 | 5 | -1 | 0 | 1
9 | 1 | 11 | 3 | 11 | 13 | 1 | 0
10 | 2 | 11 | 3 | 12 | 15 | 1 | 1
11 | 3 | 11 | 3 | 9 | 16 | 1 | 2
12 | 4 | 11 | 3 | 4 | 3 | 1 | 3
13 | 5 | 11 | 3 | 3 | -1 | 0 | 4
14 | 1 | 11 | 5 | 11 | 13 | 1 | 0
15 | 2 | 11 | 5 | 12 | 15 | 1 | 1
16 | 3 | 11 | 5 | 9 | 9 | 1 | 2
17 | 4 | 11 | 5 | 6 | 8 | 1 | 3
18 | 5 | 11 | 5 | 5 | -1 | 0 | 4
```

(18 rows)

**Examples:**

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

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



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

**Examples:**

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

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

```

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

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

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

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

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

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

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

```

**Examples:**

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

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

```

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

SELECT * FROM pgr_edwardMoore(
  'SELECT id, source, target, cost FROM edge_table',
  2, 5,

```

```

FALSE
);
seq | path_seq | node | edge | cost | agg_cost
-----+-----+-----+-----+-----+-----
 1 | 1 | 2 | 4 | 1 | 0
 2 | 2 | 5 | -1 | 0 | 1
(2 rows)

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

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

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

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

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

```

#### See Also

- [https://en.wikipedia.org/wiki/Shortest\\_Path\\_Faster\\_Algorithm](https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm)

#### Indices and tables

- [Index](#)
- [Search Page](#)

`pgr_stoerWagner` — Returns the weight of the min-cut of graph using stoerWagner algorithm. Function determines a min-cut and the min-cut weight of a connected, undirected graph implemented by Boost.Graph.



## Boost Graph Inside



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

## Availability

- Version 2.3.0
  - New **Experimental** function

## Support

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

## Description

In graph theory, the Stoer-Wagner algorithm is a recursive algorithm to solve the minimum cut problem in undirected weighted graphs with non-negative weights. The essential idea of this algorithm is to shrink the graph by merging the most intensive vertices, until the graph only contains two combined vertex sets. At each phase, the algorithm finds the minimum s-t cut for two vertices s and t chosen as its will. Then the algorithm shrinks the edge between s and t to search for non s-t cuts. The minimum cut found in all phases will be the minimum weighted cut of the graph.

A cut is a partition of the vertices of a graph into two disjoint subsets. A minimum cut is a cut for which the size or weight of the cut is not larger than the size of any other cut. For an unweighted graph, the minimum cut would simply be the cut with the least edges. For a weighted graph, the sum of all edges' weight on the cut determines whether it is a minimum cut.

## The main characteristics are:

- Process is done only on edges with positive costs.
- It's implementation is only on **undirected** graph.
- Sum of the weights of all edges between the two sets is mincut.
  - A **mincut** is a cut having the least weight.
- Values are returned when graph is connected.
  - When there is no edge in graph then EMPTY SET is return.
  - When the graph is unconnected then EMPTY SET is return.
- Sometimes a graph has multiple min-cuts, but all have the same weight. The this function determines exactly one of the min-cuts as well as its weight.

- Running time:  $\mathcal{O}(V \cdot E + V^2 \cdot \log V)$ .

### Signatures

```
pgr_stoerWagner(edges_sql)

RETURNS SET OF (seq, edge, cost, mincut)
OR EMPTY SET
```

### Example:

- TBD**

```
pgr_stoerWagner(TEXT edges_sql);
RETURNS SET OF (seq, edge, cost, mincut)
OR EMPTY SET
```

```
SELECT * FROM pgr_stoerWagner(
  'SELECT id, source, target, cost, reverse_cost
   FROM edge_table
   WHERE id < 17'
);
seq | edge | cost | mincut
-----+-----+-----+-----
  1 |  1 |  1 |    1
(1 row)
```

### Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		SQL query as described above.

### Inner query

#### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"> <li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li> </ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

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
<b>seq</b>	INT	Sequential value starting from <b>1</b> .
<b>edge</b>	BIGINT	Edges which divides the set of vertices into two.
<b>cost</b>	FLOAT	Cost to traverse of edge.
<b>mincut</b>	FLOAT	Min-cut weight of a undirected graph.

### Additional Example:

```

SELECT * FROM pgr_stoerWagner(
'SELECT id, source, target, cost, reverse_cost
  FROM edge_table
 WHERE id = 18'
);
seq | edge | cost | mincut
-----+-----
 1 | 18 | 1 | 1
(1 row)

```

Use `pgr_connectedComponents( )` function in query:

```

SELECT * FROM pgr_stoerWagner(
$$
SELECT id, source, target, cost, reverse_cost FROM edge_table
  where source = any (ARRAY(SELECT node FROM pgr_connectedComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table '
    WHERE component = 14)
  )
)
 OR
  target = any (ARRAY(SELECT node FROM pgr_connectedComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table '
    WHERE component = 14)
  )
)
$$
);
seq | edge | cost | mincut
-----+-----
 1 | 17 | 1 | 1
(1 row)

```

#### See Also

- [https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner\\_algorithm](https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner_algorithm)
- The queries use the **Sample Data** network.

#### Indices and tables

- [Index](#)
- [Search Page](#)

#### pgr\_topologicalSort - Experimental

`pgr_topologicalSort` — Returns the linear ordering of the vertices(s) for weighted directed acyclic graphs(DAG). In particular, the topological sort algorithm implemented by Boost.Graph.



#### Boost Graph Inside



#### Warning

Possible server crash

- These functions might create a server crash



#### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGGER 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 community.
- Might depend on a proposed function of pgRouting
- Might depend on a deprecated function of pgRouting

### Availability

- Version 3.0.0
  - New **experimental** function

### Support

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

### Description

The topological sort algorithm creates a linear ordering of the vertices such that if edge (u,v) appears in the graph, then v comes before u in the ordering.

This implementation can only be used with a **directed** graph with no cycles i.e. directed acyclic graph.

The main characteristics are:

- Process is valid for directed acyclic graphs only. otherwise it will throw warnings.
- For optimization purposes, if there are more than one answer, the function will return one of them.
- The returned values are ordered in topological order:
- Running time:  $\mathcal{O}(V + E)$

### Signatures

### Summary

```
pgr_topologicalSort(edges_sql)

RETURNS SET OF (seq, sorted_v)
OR EMPTY SET
```

### Example:

For a **directed** graph

```
SELECT * FROM pgr_topologicalsort(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table1'
);
seq | sorted_v
----+-----
 1 |         0
 2 |         1
 3 |         3
 4 |         2
(4 rows)
```

### Parameters

Parameter	Type	Default	Description
<b>edges_sql</b>	TEXT		SQL query as described above.

### Inner query

#### edges\_sql:

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

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> )

- When negative: edge (*source*, *target*) does not exist, therefore it's not part of the graph.

Column	Type	Default	Description
<code>reverse_cost</code>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"> <li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li> </ul>

Where:

**ANY-INTEGER:**

SMALLINT, INTEGER, BIGINT

**ANY-NUMERICAL:**

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

**Result Columns**

Returns set of (`seq`, `sorted_v`)

Column	Type	Description
<code>seq</code>	INT	Sequential value starting from <b>1</b> .
<code>sorted_v</code>	BIGINT	Linear ordering of the vertices(ordered in topological order)

**See Also**

- [https://en.wikipedia.org/wiki/Topological\\_sorting](https://en.wikipedia.org/wiki/Topological_sorting)
- The queries use the **Sample Data** network.

**Indices and tables**


- [Index](#)
- [Search Page](#)

`pgr_transitiveClosure` - Experimental


`pgr_transitiveClosure` — Returns the transitive closure graph of the input graph. In particular, the transitive closure algorithm implemented by Boost.Graph.



Boost Graph Inside

 **Warning**  
Possible server crash

- These functions might create a server crash

 **Warning**  
Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting



## Availability

- Version 3.0.0
  - New **experimental** function

## Support

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

## Description

The `transitive_closure()` function transforms the input graph `g` into the transitive closure graph `tc`.

This implementation can only be used with a **directed** graph with no cycles i.e. directed acyclic graph.

The main characteristics are:

- Process is valid for directed acyclic graphs only. otherwise it will throw warnings.
- The returned values are not ordered:
- Running time:  $\mathcal{O}(|V||E|)$

## Signatures

## Summary

The `pgr_transitiveClosure` function has the following signature:

```
pgr_transitiveClosure(Edges SQL)
RETURNS SETOF (id, vid, target_array)
```

## Example:

Complete Graph of 3 vertexs

```
SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table1'
);
seq | vid | target_array
-----+-----+-----
 1 |  0 | {1,3,2}
 2 |  1 | {3,2}
 3 |  3 | {2}
 4 |  2 | {}
(4 rows)
```

## Parameters

Column	Type	Description
<b>Edges SQL</b>	TEXT	SQL query as described in <b>Inner query</b>

## Inner query

Column	Type	Default	Description
<b>id</b>	ANY-INTEGER		Identifier of the edge.
<b>source</b>	ANY-INTEGER		Identifier of the first end point vertex of the edge.
<b>target</b>	ANY-INTEGER		Identifier of the second end point vertex of the edge.
<b>cost</b>	ANY-NUMERICAL		Weight of the edge ( <i>source</i> , <i>target</i> ) <ul style="list-style-type: none"><li>When negative: edge (<i>source</i>, <i>target</i>) does not exist, therefore it's not part of the graph.</li></ul>
<b>reverse_cost</b>	ANY-NUMERICAL	-1	Weight of the edge ( <i>target</i> , <i>source</i> ), <ul style="list-style-type: none"><li>When negative: edge (<i>target</i>, <i>source</i>) does not exist, therefore it's not part of the graph.</li></ul>

Where:

### ANY-INTEGER:

SMALLINT, INTEGER, BIGINT

### ANY-NUMERICAL:

SMALLINT, INTEGER, BIGINT, REAL, FLOAT

## Result Columns

RETURNS SETOF (seq, vid, target\_array)

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

Column	Type	Description
<b>seq</b>	INTEGER	Sequential value starting from <b>1</b> .
<b>vid</b>	BIGINT	Identifier of the vertex.
<b>target_array</b>	ARRAY[BIGINT]	Array of identifiers of the vertices that are reachable from vertex v.

#### Additional Examples

##### Example:

Some sub graphs of the sample data

```
SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=2'
);
seq | vid | target_array
-----+-----+-----
 1 | 2 | {}
 2 | 3 | {2}
(2 rows)

SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=3'
);
seq | vid | target_array
-----+-----+-----
 1 | 3 | {}
 2 | 4 | {3}
(2 rows)

SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=2 or id=3'
);
seq | vid | target_array
-----+-----+-----
 1 | 2 | {}
 2 | 3 | {2}
 3 | 4 | {3,2}
(3 rows)

SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where id=11'
);
seq | vid | target_array
-----+-----+-----
 1 | 6 | {11}
 2 | 11 | {}
(2 rows)

-- q3
SELECT * FROM pgr_transitiveclosure(
  'SELECT id,source,target,cost,reverse_cost FROM edge_table where cost=-1 or reverse_cost=-1'
);
seq | vid | target_array
-----+-----+-----
 1 | 2 | {}
 2 | 3 | {11,12,6,2}
 3 | 4 | {11,12,3,6,2}
 4 | 6 | {11,12}
 5 | 11 | {12}
 6 | 10 | {11,12}
 7 | 12 | {}
(7 rows)
```

#### See Also

- [https://en.wikipedia.org/wiki/Transitive\\_closure](https://en.wikipedia.org/wiki/Transitive_closure)
- The queries use the **Sample Data** network.

#### Indices and tables

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

pgr\_turnRestrictedPath - Experimental

pgr\_turnRestrictedPath



### Warning

Possible server crash

- These functions might create a server crash



### Warning

Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the community.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

### Availability

- Version 3.0.0
  - New **Experimental** function

### Support

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

### Description

- TBD

### Signatures

- TBD

### Parameters

- TBD

### Inner query

- TBD

### Result Columns

- TBD

### Additional Examples

**Example:**

### See Also

### Indices and tables

- [Index](#)
- [Search Page](#)

### See Also

### Indices and tables

- [Index](#)

## Release Notes

---

- [pgRouting 3.0.5 Release Notes](#)
- [pgRouting 3.0.4 Release Notes](#)
- [pgRouting 3.0.3 Release Notes](#)
- [pgRouting 3.0.2 Release Notes](#)
- [pgRouting 3.0.1 Release Notes](#)
- [pgRouting 3.0.0 Release Notes](#)
- [pgRouting 2.6.3 Release Notes](#)
- [pgRouting 2.6.2 Release Notes](#)
- [pgRouting 2.6.1 Release Notes](#)
- [pgRouting 2.6.0 Release Notes](#)
- [pgRouting 2.5.5 Release Notes](#)
- [pgRouting 2.5.4 Release Notes](#)
- [pgRouting 2.5.3 Release Notes](#)
- [pgRouting 2.5.2 Release Notes](#)
- [pgRouting 2.5.1 Release Notes](#)
- [pgRouting 2.5.0 Release Notes](#)
- [pgRouting 2.4.2 Release Notes](#)
- [pgRouting 2.4.1 Release Notes](#)
- [pgRouting 2.4.0 Release Notes](#)
- [pgRouting 2.3.2 Release Notes](#)
- [pgRouting 2.3.1 Release Notes](#)
- [pgRouting 2.3.0 Release Notes](#)
- [pgRouting 2.2.4 Release Notes](#)
- [pgRouting 2.2.3 Release Notes](#)
- [pgRouting 2.2.2 Release Notes](#)
- [pgRouting 2.2.1 Release Notes](#)
- [pgRouting 2.2.0 Release Notes](#)
- [pgRouting 2.1.0 Release Notes](#)
- [pgRouting 2.0.1 Release Notes](#)
- [pgRouting 2.0.0 Release Notes](#)
- [pgRouting 1.x Release Notes](#)

### [Release Notes](#)

To see the full list of changes check the list of [Git commits](#) on Github.

### Table of contents

- [pgRouting 3.0.5 Release Notes](#)
- [pgRouting 3.0.4 Release Notes](#)
- [pgRouting 3.0.3 Release Notes](#)
- [pgRouting 3.0.2 Release Notes](#)
- [pgRouting 3.0.1 Release Notes](#)
- [pgRouting 3.0.0 Release Notes](#)
- [pgRouting 2.6.3 Release Notes](#)
- [pgRouting 2.6.2 Release Notes](#)
- [pgRouting 2.6.1 Release Notes](#)
- [pgRouting 2.6.0 Release Notes](#)
- [pgRouting 2.5.5 Release Notes](#)
- [pgRouting 2.5.4 Release Notes](#)
- [pgRouting 2.5.3 Release Notes](#)
- [pgRouting 2.5.2 Release Notes](#)
- [pgRouting 2.5.1 Release Notes](#)
- [pgRouting 2.5.0 Release Notes](#)
- [pgRouting 2.4.2 Release Notes](#)
- [pgRouting 2.4.1 Release Notes](#)
- [pgRouting 2.4.0 Release Notes](#)
- [pgRouting 2.3.2 Release Notes](#)
- [pgRouting 2.3.1 Release Notes](#)
- [pgRouting 2.3.0 Release Notes](#)
- [pgRouting 2.2.4 Release Notes](#)
- [pgRouting 2.2.3 Release Notes](#)
- [pgRouting 2.2.2 Release Notes](#)

- [pgRouting 2.2.1 Release Notes](#)
- [pgRouting 2.2.0 Release Notes](#)
- [pgRouting 2.1.0 Release Notes](#)
- [pgRouting 2.0.1 Release Notes](#)
- [pgRouting 2.0.0 Release Notes](#)
- [pgRouting 1.x Release Notes](#)

#### [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](#)

To see all issues & pull requests closed by this release see the [Git closed milestone for 3.0.3](#) on Github.

#### 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**: Intermediate 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)

- o pgr\_maxCardinalityMatching
  - o pgr\_maxFlow
  - o pgr\_edgeDisjointPaths(one to one)
  - o pgr\_edgeDisjointPaths(one to many)
  - o pgr\_edgeDisjointPaths(many to one)
  - o pgr\_edgeDisjointPaths(many to many)
- o Components family
  - o pgr\_connectedComponents
  - o pgr\_strongComponents
  - o pgr\_biconnectedComponents
  - o pgr\_articulationPoints
  - o pgr\_bridges
- o Contraction:
  - o Removed unnecessary column seq
  - o Bug Fixes

### New Experimental functions

- o pgr\_maxFlowMinCost
- o pgr\_maxFlowMinCost\_Cost
- o pgr\_extractVertices
- o pgr\_turnRestrictedPath
- o pgr\_stoerWagner
- o pgr\_dagShortestpath
- o pgr\_topologicalSort
- o pgr\_transitiveClosure
- o VRP category
  - o pgr\_pickDeliverEuclidean
  - o pgr\_pickDeliver
- o Chinese Postman family
  - o pgr\_chinesePostman
  - o pgr\_chinesePostmanCost
- o Breadth First Search family
  - o pgr\_breadthFirstSearch
  - o pgr\_binaryBreadthFirstSearch
- o Bellman Ford family
  - o pgr\_bellmanFord
  - o pgr\_edwardMoore

### Moved to legacy

- o Experimental functions
  - o pgr\_labelGraph - Use the components family of functions instead.
  - o Max flow - functions were renamed on v2.5.0
    - o pgr\_maxFlowPushRelabel
    - o pgr\_maxFlowBoykovKolmogorov
    - o pgr\_maxFlowEdmondsKarp
    - o pgr\_maximumcardinalitymatching
  - o VRP
    - o pgr\_gsoc\_vrppdtw
- o TSP old signatures
- o pgr\_pointsAsPolygon
- o 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

- o **#1219** Implicit cast for via\_path integer to text
- o **#1193** Fixed pgr\_pointsAsPolygon breaking when comparing strings in WHERE clause
- o **#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\_bdDijkstra
  - 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 fallthrough on Astar and bdAstar.

## pgRouting 2.6.0 Release Notes

To see the issues closed by this release see the [Git closed milestone for 2.6.0](#) on Github.

## New fexperimental functions

- pgr\_lineGraphFull

## Bug fixes

- Fix pgr\_trsp(text,integer,double precision,integer,double precision,boolean,boolean[,text])
  - without restrictions
    - calls pgr\_dijkstra when both end points have a fraction IN (0,1)
    - calls pgr\_withPoints when at least one fraction NOT IN (0,1)
  - with restrictions
    - calls original trsp code

## Internal code

- Cleaned the internal code of trsp(text,integer,integer,boolean,boolean [, text])
  - Removed the use of pointers
  - Internal code can accept BIGINT
- Cleaned the internal code of withPoints

## pgRouting 2.5.5 Release Notes

To see the issues closed by this release see the [Git closed milestone for 2.5.5](#) on Github.



## Bug fixes

- Fixes driving distance when vertex is not part of the graph
- Fixes windows test
- Fixes build for python3 and perl5

## pgRouting 2.5.4 Release Notes

To see the issues closed by this release see the [Git closed milestone for 2.5.4](#) on Github.

- Fixes server crash on several functions.
  - pgr\_floydWarshall
  - pgr\_johnson
  - pgr\_astar
  - pgr\_bdAstar
  - pgr\_bdDijkstra
  - 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 fallthrough 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 postgresSQL

## 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 postgresSQL 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\_euclidianTSP

### 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\_euclidianTSP 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

- o pgr\_dijkstra(one to many)
- o pgr\_dijkstra(many to one)
- o pgr\_dijkstra(many to many)
- o pgr\_drivingDistance(multiple vertices)

### Refactored

- o pgr\_dijkstra(one to one)
- o pgr\_ksp
- o pgr\_drivingDistance(single vertex)

### Improvements

- o pgr\_alphaShape function now can generate better (multi)polygon with holes and alpha parameter.

### Proposed functionality

- o Proposed functions from Steve Woodbridge, (Classified as Convenience by the author.)
  - o pgr\_pointToEdgeNode - convert a point geometry to a vertex\_id based on closest edge.
  - o pgr\_flipEdges - flip the edges in an array of geometries so they connect end to end.
  - o pgr\_textToPoints - convert a string of x,y;x,y;... locations into point geometries.
  - o pgr\_pointsToVids - convert an array of point geometries into vertex ids.
  - o pgr\_pointsToDMatrix - Create a distance matrix from an array of points.
  - o pgr\_vidsToDMatrix - Create a distance matrix from an array of vertex\_id.
  - o pgr\_vidsToDMatrix - Create a distance matrix from an array of vertex\_id.
- o Added proposed functions from GSoc Projects:
  - o pgr\_vrppdtw
  - o pgr\_vrponedepot

### Deprecated functions

- o pgr\_getColumnName
- o pgr\_getTableName
- o pgr\_isColumnIndexed
- o pgr\_isColumnInTable
- o pgr\_quote\_ident
- o pgr\_versionless
- o pgr\_startPoint
- o pgr\_endPoint
- o pgr\_pointTold

### No longer supported

- o Removed the 1.x legacy functions

### Bug Fixes

- o Some bug fixes in other functions

### Refactoring Internal Code

- o A C and C++ library for developer was created
  - o encapsulates postgresSQL related functions
  - o encapsulates Boost.Graph graphs
    - o Directed Boost.Graph
    - o Undirected Boost.graph.
  - o allow any-integer in the id's
  - o allow any-numerical on the cost/reverse\_cost columns
- o 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

- o No track of the bug fixes were kept.

## pgRouting 2.0.0 Release Notes

To see the issues closed by this release see the [Git closed issues for 2.0.0](#) on Github.

With the release of pgRouting 2.0.0 the library has abandoned backwards compatibility to **pgRouting 1.x** releases. The main Goals for this release are:

- Major restructuring of pgRouting.
- Standardization of the function naming
- Preparation of the project for future development.

As a result of this effort:

- pgRouting has a simplified structure
- Significant new functionality has being added
- Documentation has being integrated
- Testing has being integrated
- And made it easier for multiple developers to make contributions.

### Important Changes

- Graph Analytics - tools for detecting and fixing connection some problems in a graph
- A collection of useful utility functions
- Two new All Pairs Short Path algorithms (pgr\_apspJohnson, pgr\_apspWarshall)
- Bi-directional Dijkstra and A-star search algorithms (pgr\_bdAstar, pgr\_bdDijkstra)
- One to many nodes search (pgr\_kDijkstra)
- K alternate paths shortest path (pgr\_ksp)
- New TSP solver that simplifies the code and the build process (pgr\_tsp), dropped "Gaul Library" dependency
- Turn Restricted shortest path (pgr\_trsp) that replaces Shooting Star
- Dropped support for Shooting Star
- Built a test infrastructure that is run before major code changes are checked in
- Tested and fixed most all of the outstanding bugs reported against 1.x that existing in the 2.0-dev code base.
- Improved build process for Windows
- Automated testing on Linux and Windows platforms trigger by every commit
- Modular library design
- Compatibility with PostgreSQL 9.1 or newer
- Compatibility with PostGIS 2.0 or newer
- Installs as PostgreSQL EXTENSION
- Return types re factored and unified
- Support for table SCHEMA in function parameters
- Support for `st_` PostGIS function prefix
- Added `pgr_` prefix to functions and types
- Better documentation: <https://docs.pgrouting.org>
- shooting\_star is discontinued

## pgRouting 1.x Release Notes

To see the issues closed by this release see the [Git closed issues for 1.x](#) on Github. The following release notes have been copied from the previous RELEASE\_NOTES file and are kept as a reference.

### Changes for release 1.05

- Bug fixes

### Changes for release 1.03

- Much faster topology creation
- Bug fixes

### Changes for release 1.02

- Shooting\* bug fixes
- 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](#)**