pgRouting Manual

Release v2.6.0

pgRouting Contributors

June 01, 2018

pgRouting extends the PostGIS[1]/PostgreSQL[2] geospatial database to provide geospatial routing and other network analysis functionality.

This is the manual for pgRouting v2.6.0.

The pgRouting Manual is licensed under a Creative Commons Attribution-Share Alike 3.0 License[3]. 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 http://pgrouting.org. For other licenses used in pgRouting see the *Licensing* page.

[1] http://postgis.net
[2] http://postgresql.org
[3] http://creativecommons.org/licenses/by-sa/3.0/

CHAPTER 1

General

1.1 Introduction

pgRouting is an extension of PostGIS[1] and PostgreSQL[2] geospatial database and adds routing and other network analysis functionality. A predecessor of pgRouting – pgDijkstra, written by Sylvain Pasche from Camptocamp[3], was later extended by Orkney[4] and renamed to pgRouting. The project is now supported and maintained by Georepublic[5], iMaptools[6] and a broad user community.

pgRouting is part of OSGeo Community Projects[7] from the OSGeo Foundation[8] and included on OSGeo Live[9].

1.1.1 Licensing

The following licenses can be found in pgRouting:

License	
GNU General Public License, version 2	Most features of pgRouting are available under GNU General Public License, version 2[10].
Boost Software License - Version 1.0	Some Boost extensions are available under Boost Software License - Version 1.0[11].
MIT-X License	Some code contributed by iMaptools.com is available under MIT-X license.
Creative Commons Attribution-Share Alike 3.0 License	The pgRouting Manual is licensed under a Creative Commons Attribution-Share Alike 3.0 License[12].

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

1.1.2 Contributors

This Release Contributors

[1] http://postgis.net
[2] http://postgresql.org
[3] http://camptocamp.com
[4] http://www.orkney.co.jp
[5] http://georepublic.info
[6] http://imaptools.com/
[7] http://wiki.osgeo.org/wiki/OSGeo_Community_Projects
[8] http://osgeo.org
[9] http://live.osgeo.org/
[10] http://www.gnu.org/licenses/gpl-2.0.html
[11] http://www.boost.org/LICENSE_1_0.txt
[12] http://creativecommons.org/licenses/by-sa/3.0/

Individuals (in alphabetical order)

Anthony Tasca, Virginia Vergara

And all the people that give us a little of their time making comments, finding issues, making pull requests etc.

Corporate Sponsors (in alphabetical order)

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

- Georepublic[13]
- Google Summer of Code[14]
- iMaptools[15]
- Paragon Corporation[16]
- Versaterm Inc.[17]

Contributors Past & Present:

Individuals (in alphabetical order)

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

Corporate Sponsors (in alphabetical order)

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

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

[13] https://georepublic.info/en/
[14] https://developers.google.com/open-source/gsoc/
[15] http://imaptools.com
[16] http://www.paragoncorporation.com/
[17] http://www.versaterm.com/

1.1.3 More Information

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

1.2 Installation

Table of Contents

Instructions for downloading and installing binaries for different Operative systems instructions and additional notes and corrections not included in this documentation can be found in Installation wiki[18]

To use pgRouting postGIS needs to be installed, please read the information about installation in this Install Guide[19]

1.2.1 Short Version

Extracting the tar ball

```
tar xvfz pgrouting-2.6.0.tar.gz
cd pgrouting-2.6.0
```

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

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

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

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

[18] https://github.com/pgRouting/pgrouting/wiki/Notes-on-Download%2C-Installation-and-building-pgRouting
[19] http://www.postgis.us/presentations/postgis_install_guide_22.html

1.2.2 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-2.6.0.tar.gz https://github.com/pgRouting/pgrouting/archive/v2.6.0.tar.gz
```

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

git

To download the repository

```
git clone git://github.com/pgRouting/pgrouting.git
cd pgrouting
git checkout v2.6.0
```

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

1.2.3 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 2.6.0 use the following command:

```
ALTER EXTENSION pgrouting UPDATE TO "2.6.0";
```

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

1.2.4 Dependencies

Compilation Dependencies

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

- C and C++0x compilers * g++ version >= 4.8
- Postgresql version >= 9.3
- PostGIS version >= 2.2
- The Boost Graph Library (BGL). Version >= 1.53
- CMake >= 3.2
- CGAL >= 4.2

optional dependencies

For user's documentation

- Sphinx >= 1.1
- Latex

For developer's documentation

- Doxygen >= 1.7

For testing

- pgtap
- pg_prove

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 \
    libcgal-dev
```

Optional dependencies

For documentation and testing

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

1.2.5 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.

1.2.6 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.

1.2.7 Testing

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

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

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

1.2.8 See Also

Indices and tables

- genindex
- search

1.3 Support

pgRouting community support is available through the pgRouting website[20], documentation[21], tutorials, mailing lists and others. If you're looking for *commercial support*, find below a list of companies providing pgRouting development and consulting services.

1.3.1 Reporting Problems

Bugs are reported and managed in an issue tracker[22]. 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[23] 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.

[20] http://pgrouting.org/support.html
[21] http://docs.pgrouting.org
[22] https://github.com/pgrouting/pgrouting/issues
[23] https://github.com/pgRouting/pgrouting/issues/new

5. For the versions where you can replicate the problem, note the operating system and version of pgRouting, PostGIS and PostgreSQL.
6. It is recommended to use the following wrapper on the problem to pin point the step that is causing the problem.

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

1.3.2 Mailing List and GIS StackExchange

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

- User mailing list: http://lists.osgeo.org/mailman/listinfo/pgrouting-users
- Developer mailing list: http://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[24] and tag the question with `pgrouting`. Find all questions tagged with `pgrouting` under http://gis.stackexchange.com/questions/tagged/pgrouting or subscribe to the pgRouting questions feed[25].

1.3.3 Commercial Support

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

Company	Offices in	Website
Georepublic	Germany, Japan	https://georepublic.info
iMaptools	United States	http://imaptools.com
Paragon Corporation	United States	http://www.paragoncorporation.com
Camptocamp	Switzerland, France	http://www.camptocamp.com

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

1.4 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,
```

[24] http://gis.stackexchange.com/
[25] http://gis.stackexchange.com/feeds/tag?tagnames=pgrouting&sort=newest

```
    y1 FLOAT,
    x2 FLOAT,
    y2 FLOAT,
    the_geom geometry
);
```

Insert data

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

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

Topology

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

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

Points of interest

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

```
CREATE TABLE pointsOfInterest(
    pid BIGSERIAL,
    x FLOAT,
    y FLOAT,
    edge_id BIGINT,
```

```
    side CHAR,
    fraction FLOAT,
    the_geom geometry,
    newPoint geometry
);

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

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

Restrictions

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

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

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

Categories

- Used with the *Flow - Family of functions* functions.

```
/*
CREATE TABLE categories (
    category_id INTEGER,
    category text,
    capacity BIGINT
);

INSERT INTO categories VALUES
(1, 'Category 1', 130),
(2, 'Category 2', 100),
(3, 'Category 3',  80),
(4, 'Category 4',  50);
*/
```

Vertex table

- Used in some deprecated signatures or deprecated functions.

```
-- TODO check if this table is still used
CREATE TABLE vertex_table (
    id SERIAL,
    x FLOAT,
    y FLOAT
);
INSERT INTO vertex_table VALUES
(1,2,0), (2,2,1), (3,3,1), (4,4,1), (5,0,2), (6,1,2), (7,2,2),
(8,3,2), (9,4,2), (10,2,3), (11,3,3), (12,4,3), (13,2,4);
```

1.4.1 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.

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.

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

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

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),
```

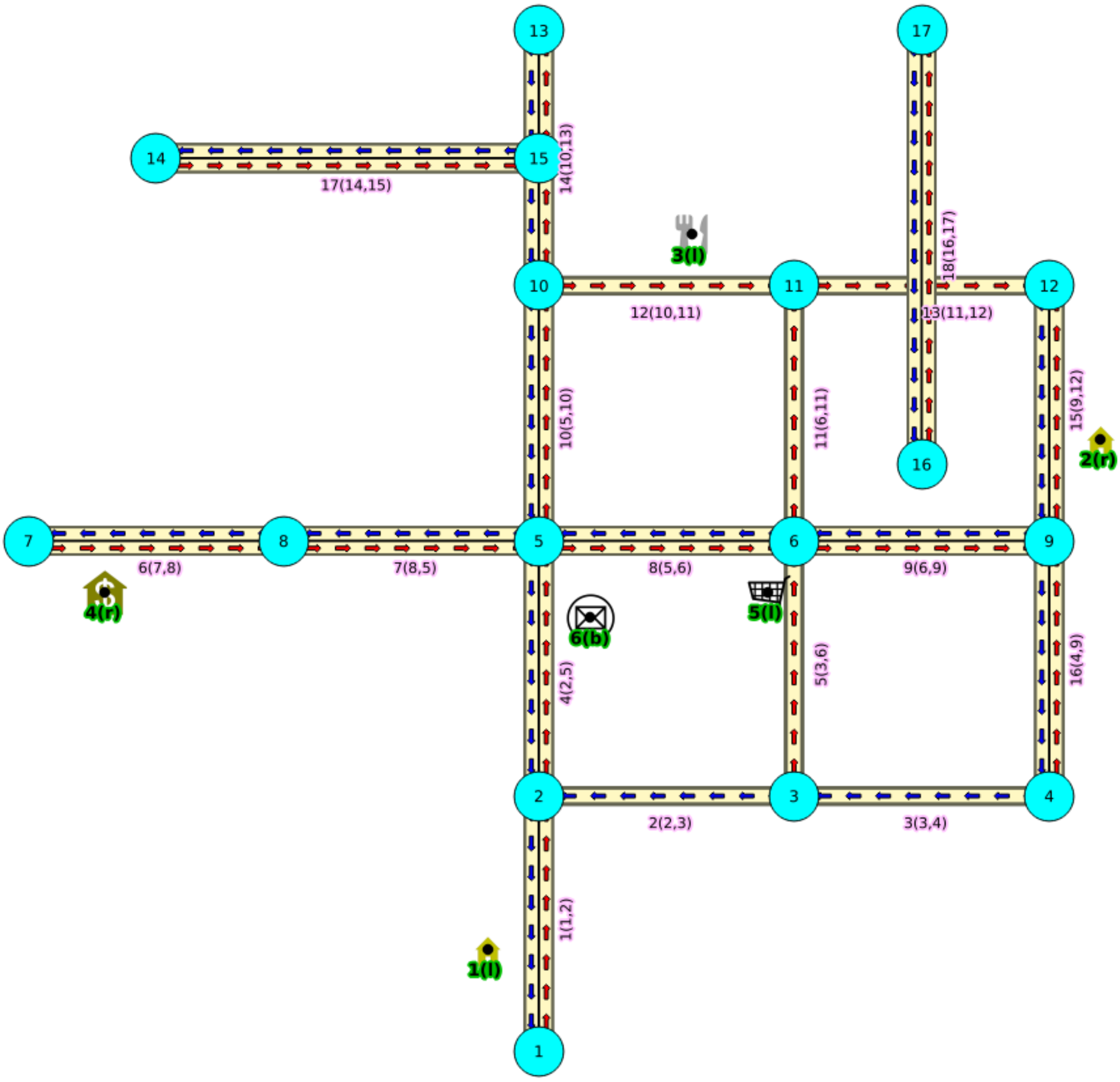


Fig. 1.1: **Graph 1: Directed, with cost and reverse cost**

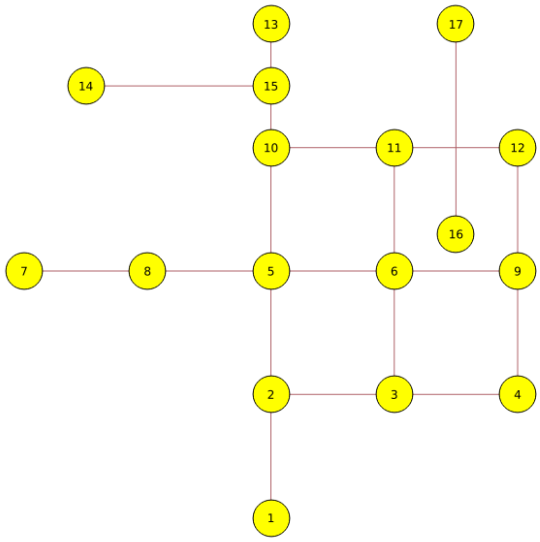


Fig. 1.2: **Graph 2: Undirected, with cost and reverse cost**

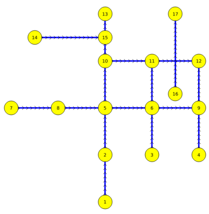


Fig. 1.3: **Graph 3: Directed, with cost**

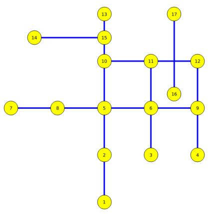


Fig. 1.4: **Graph 4: Undirected, with cost**

```
(  3,    42,   66,    10,    65,   146,   90,    0,   75),
(  4,    42,   68,   -10,   727,   782,   90,    9,    0),
(  5,    42,   65,    10,    15,    67,   90,    0,    7),
(  6,    40,   69,    20,   621,   702,   90,    0,    2),
(  7,    40,   66,   -10,   170,   225,   90,    5,    0),
(  8,    38,   68,    20,   255,   324,   90,    0,   10),
(  9,    38,   70,    10,   534,   605,   90,    0,    4),
( 10,    35,   66,   -20,   357,   410,   90,    8,    0),
( 11,    35,   69,    10,   448,   505,   90,    0,    1),
( 12,    25,   85,   -20,   652,   721,   90,   18,    0),
( 13,    22,   75,    30,    30,    92,   90,    0,   17),
( 14,    22,   85,   -40,   567,   620,   90,   16,    0),
( 15,    20,   80,   -10,   384,   429,   90,   19,    0),
( 16,    20,   85,    40,   475,   528,   90,    0,   14),
( 17,    18,   75,   -30,    99,   148,   90,   13,    0),
( 18,    15,   75,    20,   179,   254,   90,    0,   12),
( 19,    15,   80,    10,   278,   345,   90,    0,   15),
( 20,    30,   50,    10,    10,    73,   90,    0,   24),
( 21,    30,   52,   -10,   914,   965,   90,   30,    0),
( 22,    28,   52,   -20,   812,   883,   90,   28,    0),
( 23,    28,   55,    10,   732,   777,    0,    0,  103),
( 24,    25,   50,   -10,    65,   144,   90,   20,    0),
( 25,    25,   52,    40,   169,   224,   90,    0,   27),
( 26,    25,   55,   -10,   622,   701,   90,   29,    0),
( 27,    23,   52,   -40,   261,   316,   90,   25,    0),
( 28,    23,   55,    20,   546,   593,   90,    0,   22),
( 29,    20,   50,    10,   358,   405,   90,    0,   26),
( 30,    20,   55,    10,   449,   504,   90,    0,   21),
( 31,    10,   35,   -30,   200,   237,   90,   32,    0),
( 32,    10,   40,    30,    31,   100,   90,    0,   31),
( 33,     8,   40,    40,    87,   158,   90,    0,   37),
( 34,     8,   45,   -30,   751,   816,   90,   38,    0),
( 35,     5,   35,    10,   283,   344,   90,    0,   39),
( 36,     5,   45,    10,   665,   716,    0,    0,  105),
( 37,     2,   40,   -40,   383,   434,   90,   33,    0),
( 38,     0,   40,    30,   479,   522,   90,    0,   34),
( 39,     0,   45,   -10,   567,   624,   90,   35,    0),
( 40,    35,   30,   -20,   264,   321,   90,   42,    0),
( 41,    35,   32,   -10,   166,   235,   90,   43,    0),
( 42,    33,   32,    20,    68,   149,   90,    0,   40),
( 43,    33,   35,    10,    16,    80,   90,    0,   41),
( 44,    32,   30,    10,   359,   412,   90,    0,   46),
( 45,    30,   30,    10,   541,   600,   90,    0,   48),
( 46,    30,   32,   -10,   448,   509,   90,   44,    0),
( 47,    30,   35,   -10,  1054,  1127,   90,   49,    0),
( 48,    28,   30,   -10,   632,   693,   90,   45,    0),
( 49,    28,   35,    10,  1001,  1066,   90,    0,   47),
( 50,    26,   32,    10,   815,   880,   90,    0,   52),
( 51,    25,   30,    10,   725,   786,    0,    0,  101),
( 52,    25,   35,   -10,   912,   969,   90,   50,    0),
( 53,    44,    5,    20,   286,   347,   90,    0,   58),
( 54,    42,   10,    40,   186,   257,   90,    0,   60),
( 55,    42,   15,   -40,    95,   158,   90,   57,    0),
( 56,    40,    5,    30,   385,   436,   90,    0,   59),
( 57,    40,   15,    40,    35,    87,   90,    0,   55),
( 58,    38,    5,   -20,   471,   534,   90,   53,    0),
( 59,    38,   15,   -30,   651,   740,   90,   56,    0),
( 60,    35,    5,   -40,   562,   629,   90,   54,    0),
( 61,    50,   30,   -10,   531,   610,   90,   67,    0),
( 62,    50,   35,    20,   262,   317,   90,    0,   68),
( 63,    50,   40,    50,   171,   218,   90,    0,   74),
( 64,    48,   30,    10,   632,   693,    0,    0,  102),
( 65,    48,   40,    10,    76,   129,   90,    0,   72),
```

```
( 66,    47,   35,    10,   826,   875,   90,    0,   69),
( 67,    47,   40,    10,    12,    77,   90,    0,   61),
( 68,    45,   30,   -20,   734,   777,   90,   62,    0),
( 69,    45,   35,   -10,   916,   969,   90,   66,    0),
( 70,    95,   30,   -30,   387,   456,   90,   81,    0),
( 71,    95,   35,    20,   293,   360,   90,    0,   77),
( 72,    53,   30,   -10,   450,   505,   90,   65,    0),
( 73,    92,   30,   -10,   478,   551,   90,   76,    0),
( 74,    53,   35,   -50,   353,   412,   90,   63,    0),
( 75,    45,   65,   -10,   997,  1068,   90,    3,    0),
( 76,    90,   35,    10,   203,   260,   90,    0,   73),
( 77,    88,   30,   -20,   574,   643,   90,   71,    0),
( 78,    88,   35,    20,   109,   170,    0,    0,  104),
( 79,    87,   30,    10,   668,   731,   90,    0,   80),
( 80,    85,   25,   -10,   769,   820,   90,   79,    0),
( 81,    85,   35,    30,    47,   124,   90,    0,   70),
( 82,    75,   55,    20,   369,   420,   90,    0,   85),
( 83,    72,   55,   -20,   265,   338,   90,   87,    0),
( 84,    70,   58,    20,   458,   523,   90,    0,   89),
( 85,    68,   60,   -20,   555,   612,   90,   82,    0),
( 86,    66,   55,    10,   173,   238,   90,    0,   91),
( 87,    65,   55,    20,    85,   144,   90,    0,   83),
( 88,    65,   60,   -10,   645,   708,   90,   90,    0),
( 89,    63,   58,   -20,   737,   802,   90,   84,    0),
( 90,    60,   55,    10,    20,    84,   90,    0,   88),
( 91,    60,   60,   -10,   836,   889,   90,   86,    0),
( 92,    67,   85,    20,   368,   441,   90,    0,   93),
( 93,    65,   85,   -20,   475,   518,   90,   92,    0),
( 94,    65,   82,   -10,   285,   336,   90,   96,    0),
( 95,    62,   80,   -20,   196,   239,   90,   98,    0),
( 96,    60,   80,    10,    95,   156,   90,    0,   94),
( 97,    60,   85,    30,   561,   622,    0,    0,  106),
( 98,    58,   75,    20,    30,    84,   90,    0,   95),
( 99,    55,   80,   -20,   743,   820,   90,  100,    0),
( 100,   55,   85,    20,   647,   726,   90,    0,   99),
( 101,   25,   30,   -10,   725,   786,   90,   51,    0),
( 102,   48,   30,   -10,   632,   693,   90,   64,    0),
( 103,   28,   55,   -10,   732,   777,   90,   23,    0),
( 104,   88,   35,   -20,   109,   170,   90,   78,    0),
( 105,    5,   45,   -10,   665,   716,   90,   36,    0),
( 106,   60,   85,   -30,   561,   622,   90,   97,    0);
```

CHAPTER 2

Pgrouting Concepts

2.1 pgRouting Concepts

Contents

2.1.1 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

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

For Postgresql 9.2 and later versions

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

Load Data

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

osm2pgrouting

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

shp2pgsql

- this is the postgresql shapefile loader

ogr2ogr

- this is a vector data conversion utility

osm2pgsql

- this is a tool for loading OSM data into postgresql

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

Build a Routing Topology

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

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

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

- *pgr_createTopology*

Check the Routing Topology

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

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

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

- *pgr_analyzeGraph*
- *pgr_analyzeOneway*
- *pgr_nodeNetwork*

Compute a Path

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

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

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

- *pgr_dijkstra*

2.1.2 Inner Queries

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the edges_sql query (id is not necessary)

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the edges_sql query for Max-flow like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the Points SQL query

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

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

Where:

ANY-INTEGER smallint, int, bigint

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

2.1.3 Return columns & values

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

Description of the return values for a path

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

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

Description of the return values for a Cost function

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

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

Description of the Return Values

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

2.1.4 Advanced Topics

Routing Topology

Overview

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

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

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

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

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

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

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

```
UPDATE edge_table SET x1 = st_x(st_startpoint(the_geom)),
                      y1 = st_y(st_startpoint(the_geom)),
                      x2 = st_x(st_endpoint(the_geom)),
                      y2 = st_y(st_endpoint(the_geom)),
  cost_len  = st_length_spheroid(the_geom, 'SPHEROID["WGS84",6378137,298.25728]'),
```

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

-- UPDATE the cost information based on oneway streets

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

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

VACUUM ANALYZE VERBOSE edge_table;
```

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

Graph Analytics

Overview

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

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

Analyze a Graph

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

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

In the vertices table "mytab_vertices_pgr":

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

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

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

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

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

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

Analyze One Way Streets

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

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

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

Example

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

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

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

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

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

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

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

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

2.1.5 Performance Tips

For the Routing functions

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

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

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

Integrating the inner query to the pgRouting function:

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

For the topology functions:

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

Analize the new topology based on the actual topology:

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

Or create a new topology if the change is permanent:

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

2.1.6 How to contribute

Wiki

- Edit an existing pgRouting Wiki[1] page.
- Or create a new Wiki page
  - Create a page on the pgRouting Wiki[2]
  - Give the title an appropriate name
- Example[3]

Adding Functionaity to pgRouting

Consult the developer's documentation[4]

Indices and tables

- genindex
- search

Reference

pgr_version - to get pgRouting's version information.

[1] https://github.com/pgRouting/pgrouting/wiki
[2] https://github.com/pgRouting/pgrouting/wiki
[3] https://github.com/pgRouting/pgrouting/wiki/How-to:-Handle-parallel-edges-(KSP)
[4] http://docs.pgrouting.org/doxy/2.4/index.html

2.2 pgr_version

2.2.1 Name

`pgr_version` — Query for pgRouting version information.

2.2.2 Synopsis

Returns a table with pgRouting version information.

```
table() pgr_version();
```

2.2.3 Description

Returns a table with:

Column	Type	Description
version	`varchar`	pgRouting version
tag	`varchar`	Git tag of pgRouting build
hash	`varchar`	Git hash of pgRouting build
branch	`varchar`	Git branch of pgRouting build
boost	`varchar`	Boost version

History

- New in version 2.0.0

2.2.4 Examples

- Query for the version string

```
SELECT version FROM pgr_version();
 version
---------
 2.6.0
(1 row)
```

2.2.5 See Also

Indices and tables

- genindex
- search

CHAPTER 3

Data Types

- *pgr_costResult[]* - A set of records to describe a path result with cost attribute.
- *pgr_costResult3[]* - A set of records to describe a path result with cost attribute.
- *pgr_geomResult* - A set of records to describe a path result with geometry attribute.

3.1 pgRouting Data Types

The following are commonly used data types for some of the pgRouting functions.

- *pgr_costResult[]* - A set of records to describe a path result with cost attribute.
- *pgr_costResult3[]* - A set of records to describe a path result with cost attribute.
- *pgr_geomResult* - A set of records to describe a path result with geometry attribute.

3.1.1 pgr_costResult[]

Name

`pgr_costResult[]` — A set of records to describe a path result with cost attribute.

Description

```
CREATE TYPE pgr_costResult AS
(
    seq integer,
    id1 integer,
    id2 integer,
    cost float8
);
```

seq sequential ID indicating the path order

id1 generic name, to be specified by the function, typically the node id

id2 generic name, to be specified by the function, typically the edge id

cost cost attribute

3.1.2 pgr_costResult3[] - Multiple Path Results with Cost

Name

`pgr_costResult3[]` — A set of records to describe a path result with cost attribute.

Description

```
CREATE TYPE pgr_costResult3 AS
(
    seq integer,
    id1 integer,
    id2 integer,
    id3 integer,
    cost float8
);
```

seq sequential ID indicating the path order

id1 generic name, to be specified by the function, typically the path id

id2 generic name, to be specified by the function, typically the node id

id3 generic name, to be specified by the function, typically the edge id

cost cost attribute

History

- New in version 2.0.0
- Replaces `path_result`

See Also

- *Introduction*

Indices and tables

- genindex
- search

3.1.3 pgr_geomResult[]

Name

`pgr_geomResult[]` — A set of records to describe a path result with geometry attribute.

Description

```
CREATE TYPE pgr_geomResult AS
(
    seq integer,
    id1 integer,
    id2 integer,
```

```
    geom geometry
);
```

seq sequential ID indicating the path order

id1 generic name, to be specified by the function

id2 generic name, to be specified by the function

geom geometry attribute

History

- New in version 2.0.0
- Replaces `geoms`

See Also

- *Introduction*

Indices and tables

- genindex
- search

3.1.4 See Also

Indices and tables

- genindex
- search

Topology 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.

4.1 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.

4.1.1 pgr_createTopology

Name

`pgr_createTopology` — Builds a network topology based on the geometry information.

Synopsis

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.

```
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)
```

Description

Parameters

The topology creation function accepts the following parameters:

edge_table `text` Network table name. (may contain the schema name AS well)

tolerance `float8` Snapping tolerance of disconnected edges. (in projection unit)

the_geom `text` Geometry column name of the network table. Default value is `the_geom`.

id `text` Primary key column name of the network table. Default value is `id`.

source `text` Source column name of the network table. Default value is `source`.

target `text` Target column name of the network table. Default value is `target`.

rows_where `text` Condition to SELECT a subset or rows. Default value is `true` to indicate all rows that where `source` or `target` have a null value, otherwise the condition is used.

clean `boolean` Clean any previous topology. Default value is `false`.

Warning: The `edge_table` will be affected

- The `source` column values will change.
- The `target` column values will change.
  - An index will be created, if it doesn't exists, to speed up the process to the following columns:
    - `id`
    - `the_geom`
    - `source`
    - `target`

The function returns:

- `OK` after the network topology has been built.
  - Creates a vertices table: <edge_table>_vertices_pgr.
  - Fills `id` and `the_geom` columns of the vertices table.
  - Fills the source and target columns of the edge table referencing the `id` of the vertices table.
- `FAIL` when the network topology was not built due to an error:
  - A required column of the Network table is not found or is not of the appropriate type.
  - The condition is not well formed.
  - The names of source , target or id are the same.
  - The SRID of the geometry could not be determined.

The Vertices Table

The vertices table is a requirement of the *pgr_analyzeGraph* and the *pgr_analyzeOneway* functions.

The structure of the vertices table is:

id `bigint` Identifier of the vertex.

cnt `integer` Number of vertices in the edge_table that reference this vertex. See *pgr_analyzeGraph*.

chk `integer` Indicator that the vertex might have a problem. See *pgr_analyzeGraph*.

ein `integer` Number of vertices in the edge_table that reference this vertex AS incoming. See *pgr_analyzeOneway*.

eout `integer` Number of vertices in the edge_table that reference this vertex AS outgoing. See *pgr_analyzeOneway*.

the_geom `geometry` Point geometry of the vertex.

History

- Renamed in version 2.0.0

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
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
NOTICE:  Performing checks, please wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
 pgr_createtopology
--------------------
 OK
(1 row)
```

Warning:
An error would occur when the arguments are not given in the appropriate order:
In this example, the column `id` of the table `ege_table` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the id column.

```
SELECT  pgr_createTopology('edge_table', 0.001,
    'id', 'the_geom');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table', 0.001, 'id', 'the_geom', 'source', 'target', rows_wher
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 FRO
SELECT 18
```

Using positional notation:

The arguments need to be given in the order described in the parameters.

Note that this example uses clean flag. So it recreates the whole vertices table.

```
SELECT  pgr_createTopology('mytable', 0.001, 'mygeom', 'gid', 'src', 'tgt', clean := TRUE);
 pgr_createtopology
--------------------
 OK
(1 row)
```

Warning:
An error would occur when the arguments are not given in the appropiriate order:
In this example, the column `gid` of the table `mytable` is passed to the function AS the geometry column, and the geometry column `mygeom` is passed to the function AS the id column.

```
SELECT  pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt', rows_where := 'true
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
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('mytable', 0.001, source:='src', id:='gid', target:='tgt', the_geom:='m
 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:='m
 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:='m
    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:='m
    rows_where:='mygeom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100)');
 pgr_createtopology
```

```
--------------------
 OK
(1 row)
```

Examples 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
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
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

- *Routing Topology* 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

- genindex
- search

4.1.2 pgr_createVerticesTable

Name

`pgr_createVerticesTable` — Reconstructs the vertices table based on the source and target information.

Synopsis

The function returns:

- `OK` after the vertices table has been reconstructed.
- `FAIL` when the vertices table was not reconstructed due to an error.

```
    pgr_createVerticesTable(edge_table, the_geom, source, target, rows_where)
RETURNS VARCHAR
```

Description

Parameters

The reconstruction of the vertices table function accepts the following parameters:

edge_table `text` Network table name. (may contain the schema name as well)

the_geom `text` Geometry column name of the network table. Default value is `the_geom`.

source `text` Source column name of the network table. Default value is `source`.

target `text` Target column name of the network table. Default value is `target`.

rows_where `text` Condition to SELECT a subset or rows. Default value is `true` to indicate all rows.

Warning: The `edge_table` will be affected

- An index will be created, if it doesn't exists, to speed up the process to the following columns:
  - `the_geom`
  - `source`
  - `target`

The function returns:

- `OK` after the vertices table has been reconstructed.
  - Creates a vertices table: <edge_table>_vertices_pgr.
  - Fills `id` and `the_geom` columns of the vertices table based on the source and target columns of the edge table.
- `FAIL` when the vertices table was not reconstructed due to an error.
  - A required column of the Network table is not found or is not of the appropriate type.
  - The condition is not well formed.
  - The names of source, target are the same.
  - The SRID of the geometry could not be determined.

The Vertices Table

The vertices table is a requierment of the *pgr_analyzeGraph* and the *pgr_analyzeOneway* functions.

The structure of the vertices table is:

id `bigint` Identifier of the vertex.

cnt `integer` Number of vertices in the edge_table that reference this vertex. See *pgr_analyzeGraph*.

chk `integer` Indicator that the vertex might have a problem. See *pgr_analyzeGraph*.

ein `integer` Number of vertices in the edge_table that reference this vertex as incoming. See *pgr_analyzeOneway*.

eout `integer` Number of vertices in the edge_table that reference this vertex as outgoing. See *pgr_analyzeOneway*.

the_geom `geometry` Point geometry of the vertex.

History

- Renamed in version 2.0.0

Usage when the edge table's columns MATCH the default values:

The simplest way to use pgr_createVerticesTable is:

```
SELECT  pgr_createVerticesTable('edge_table');
```

When the arguments are given in the order described in the parameters:

```
SELECT  pgr_createVerticesTable('edge_table','the_geom','source','target');
```

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:  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 fou
NOTICE:  Unexpected error raise_exception
pgr_createverticestable
-------------------------
  FAIL
(1 row)
```

When using the named notation

The order of the parameters do not matter:

```
SELECT  pgr_createVerticesTable('edge_table',the_geom:='the_geom',source:='source',target:='targe
```

```
SELECT  pgr_createVerticesTable('edge_table',source:='source',target:='target',the_geom:='the_geo
```

Parameters defined with a default value can be omitted, as long as the value matches the default:

```
SELECT  pgr_createVerticesTable('edge_table',source:='source');
```

Selecting rows using rows_where parameter

Selecting rows based on the id.

```
SELECT  pgr_createVerticesTable('edge_table',rows_where:='id < 10');
```

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,
```

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;
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT  pgr_createVerticesTable('edge_table',rows_where:='the_geom && (select st_buffer(othergeom,
```

Usage when the edge table's columns DO NOT MATCH the default values:

For the following table

```
DROP TABLE IF EXISTS mytable;
CREATE TABLE mytable AS (SELECT id AS gid, the_geom AS mygeom,source AS src ,target AS tgt FROM ed
```

Using positional notation:

The arguments need to be given in the order described in the parameters:

```
SELECT  pgr_createVerticesTable('mytable','mygeom','src','tgt');
```

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: Table mytable not found
HINT:    ----> Check your table name
NOTICE:  Unexpected error raise_exception
pgr_createverticestable
-------------------------
 FAIL
(1 row)
```

When using the named notation

The order of the parameters do not matter:

```
SELECT  pgr_createVerticesTable('mytable',the_geom:='mygeom',source:='src',target:='tgt');
```

```
SELECT  pgr_createVerticesTable('mytable',source:='src',target:='tgt',the_geom:='mygeom');
```

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 gid.

```
SELECT  pgr_createVerticesTable('mytable','mygeom','src','tgt',rows_where:='gid < 10');
```

```
SELECT  pgr_createVerticesTable('mytable',source:='src',target:='tgt',the_geom:='mygeom',rows_whe
```

Selecting the rows where the geometry is near the geometry of row with `gid` =5 .

```
SELECT  pgr_createVerticesTable('mytable','mygeom','src','tgt',
                                rows_where:='the_geom && (SELECT st_buffer(mygeom,0.5) FROM mytable WH
```

```
SELECT  pgr_createVerticesTable('mytable',source:='src',target:='tgt',the_geom:='mygeom',
                                rows_where:='mygeom && (SELECT st_buffer(mygeom,0.5) FROM mytable WHER
```

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;
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT  pgr_createVerticesTable('mytable','mygeom','src','tgt',
                                 rows_where:='the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTa
```

```
SELECT  pgr_createVerticesTable('mytable',source:='src',target:='tgt',the_geom:='mygeom',
                                 rows_where:='the_geom && (SELECT st_buffer(othergeom,0.5) FROM otherTa
```

Examples

```
    SELECT pgr_createVerticesTable('edge_table');
    NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, pelase 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)
```

The example uses the Sample Data network.

See Also

- *Routing Topology* for an overview of a topology for routing algorithms.
- *pgr_createTopology* 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

- genindex
- search

4.1.3 pgr_analyzeGraph

Name

`pgr_analyzeGraph` — Analyzes the network topology.

Synopsis

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')
```

Description

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.

History

- New in version 2.0.0

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

When the arguments are given in the order described in the parameters:

```
SELECT  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target');
```

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
```

```
SELECT  pgr_analyzeGraph('edge_table',0.001,source:='source',id:='id',target:='target',the_geom:=
```

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');
```

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');
```

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
```

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;
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(other_geom,
```

Usage when the edge table's columns DO NOT MATCH the default values:

For the following table

```
DROP TABLE IF EXISTS mytable;
CREATE TABLE mytable AS  (SELECT id AS gid,  source AS src ,target AS tgt , the_geom AS mygeom FROM
SELECT pgr_createTopology('mytable',0.001,'mygeom','gid','src','tgt');
```

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');
```

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',0.001,'gid','mygeom','src','tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('mytable',0.001,'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'
```

```
SELECT  pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom'
```

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');
```

```
SELECT  pgr_analyzeGraph('mytable',0.001,source:='src',id:='gid',target:='tgt',the_geom:='mygeom'
```

Selecting the rows WHERE the geometry is near the geometry of row with `id` =5 .

```
SELECT  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',
                                rows_where:='mygeom && (SELECT st_buffer(mygeom,1) FROM mytable WHERE 
```

```
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 
```

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;
CREATE TABLE otherTable AS  (SELECT 'myhouse'::text AS place, st_point(2.5,2.5) AS other_geom) ;
SELECT  pgr_analyzeGraph('mytable',0.001,'mygeom','gid','src','tgt',
         rows_where:='mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE place='||quo
```

```
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='||quot
```

Examples

```
    SELECT  pgr_createTopology('edge_table',0.001);
    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, pelase wait...
    NOTICE:  Analyzing for dead ends. Please wait...
    NOTICE:  Analyzing for gaps. Please wait...
    NOTICE:  Analyzing for isolated edges. Please wait...
    NOTICE:  Analyzing for ring geometries. Please wait...
    NOTICE:  Analyzing for intersections. Please wait...
    NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
    NOTICE:                    Isolated segments: 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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 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)

-- Simulate removal of edges
SELECT pgr_createTopology('edge_table', 0.001,rows_where:='id <17');
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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 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');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.001,'the_geom','id','source','target','id <17')
NOTICE:  Performing checks, pelase 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_analyzeGraph
    ---------------------
     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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 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

- *Routing Topology* 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

- genindex
- search

4.1.4 pgr_analyzeOneway

Name

`pgr_analyzeOneway` — Analyzes oneway Sstreets and identifies flipped segments.

Synopsis

This function analyzes oneway streets in a graph and identifies any flipped segments.

```
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);
```

Description

The analyses of one way segments is pretty simple but can be a powerful tools to identifying some the potential problems created by setting the direction of a segment the wrong way. A node is a *source* if it has edges the exit from that node and no edges enter that node. Conversely, a node is a *sink* if all edges enter the node but none exit that node. For a *source* type node it is logically impossible to exist because no vehicle can exit the node if no vehicle and enter the node. Likewise, if you had a *sink* node you would have an infinite number of vehicle piling up on this node because you can enter it but not leave it.

So why do we care if the are not feasible? Well if the direction of an edge was reversed by mistake we could generate exactly these conditions. Think about a divided highway and on the north bound lane one segment got entered wrong or maybe a sequence of multiple segments got entered wrong or maybe this happened on a round-about. The result would be potentially a *source* and/or a *sink* node.

So by counting the number of edges entering and exiting each node we can identify both *source* and *sink* nodes so that you can look at those areas of your network to make repairs and/or report the problem back to your data vendor.

Prerequisites

The edge table to be analyzed must contain a source column and a target column filled with id's of the vertices of the segments and the corresponding vertices table <edge_table>_vertices_pgr that stores the vertices information.

- Use *pgr_createVerticesTable* to create the vertices table.
- Use *pgr_createTopology* to create the topology and the vertices table.

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.

History

- New in version 2.0.0

Examples

```
SELECT pgr_analyzeOneway('edge_table',
ARRAY['', 'B', 'TF'],
ARRAY['', 'B', 'FT'],
ARRAY['', 'B', 'FT'],
ARRAY['', 'B', 'TF'],
oneway:='dir');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table','{"",B,TF}','{"",B,FT}','{"",B,FT}','{"",B,TF}','dir','sou
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

- *Routing Topology* 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

- genindex
- search

4.1.5 pgr_nodeNetwork

Name

`pgr_nodeNetwork` - Nodes an network edge table.

Author Nicolas Ribot

Copyright Nicolas Ribot, The source code is released under the MIT-X license.

Synopsis

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
```

Description

A common problem associated with bringing GIS data into pgRouting is the fact that the data is often not "noded" correctly. This will create invalid topologies, which will result in routes that are incorrect.

What we mean by "noded" is that at every intersection in the road network all the edges will be broken into separate road segments. There are cases like an over-pass and under-pass intersection where you can not traverse from the over-pass to the under-pass, but this function does not have the ability to detect and accommodate those situations.

This function reads the `edge_table` table, that has a primary key column `id` and geometry column named `the_geom` and intersect all the segments in it against all the other segments and then creates a table `edge_table_noded`. It uses the `tolerance` for deciding that multiple nodes within the tolerance are considered the same node.

Parameters

edge_table `text` Network table name. (may contain the schema name as well)

tolerance `float8` tolerance for coincident points (in projection unit)dd

id `text` Primary key column name of the network table. Default value is `id`.

the_geom text Geometry column name of the network table. Default value is the_geom.

table_ending text Suffix for the new table's. Default value is noded.

The output table will have for edge_table_noded

id bigint Unique identifier for the table

old_id bigint Identifier of the edge in original table

sub_id integer Segment number of the original edge

source integer Empty source column to be used with *pgr_createTopology* function

target integer Empty target column to be used with *pgr_createTopology* function

the geom geometry Geometry column of the noded network

History

- New in version 2.0.0

Example

Let's create the topology for the data in Sample Data

```
SELECT pgr_createTopology('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase 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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 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:  pgr_nodeNetwork('edge_table',0.001,'the_geom','id','noded')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Processing, pelase 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

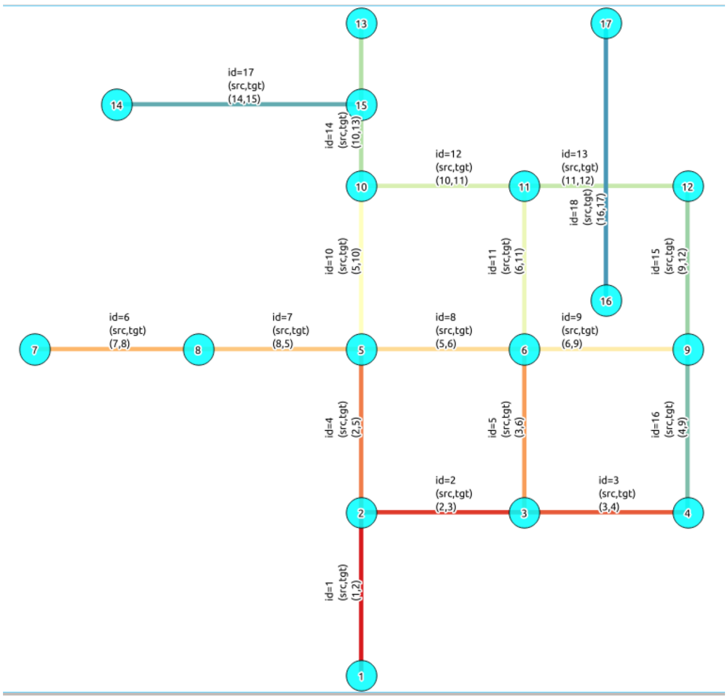
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED 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_pg
NOTICE:  ----------------------------------------------
pgr_createtopology
--------------------
 OK
(1 row)
```

Now let's analyze the new topology

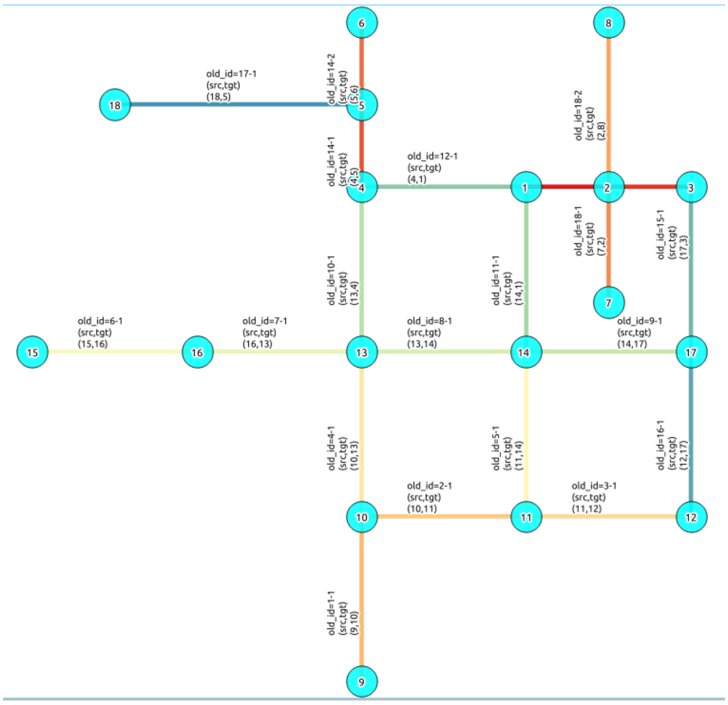
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 6
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
 OK
(1 row)
```

Images

Before Image



After Image



Comparing the results

Comparing with the Analysis in the original edge_table, we see that.

	Before	After
Table name	edge_table	edge_table_noded
Fields	All original fields	Has only basic fields to do a topology analysis
Dead ends	• Edges with 1 dead end: 1,6,24 • Edges with 2 dead ends 17,18 Edge 17's right node is a dead end because there is no other edge sharing that same node. (cnt=1)	Edges with 1 dead end: 1-1 ,6-1,14-2, 18-1 17-1 18-2
Isolated segments	two isolated segments: 17 and 18 both they have 2 dead ends	**No Isolated segments** • Edge 17 now shares a node with edges 14-1 and 14-2 • Edges 18-1 and 18-2 share a node with edges 13-1 and 13-2
Gaps	There is a gap between edge 17 and 14 because edge 14 is near to the right node of edge 17	Edge 14 was segmented Now edges: 14-1 14-2 17 share the same node The tolerance value was taken in account
Intersections	Edges 13 and 18 were intersecting	Edges were segmented, So, now in the interection's point there is a node and the following edges share it: 13-1 13-2 18-1 18-2

Now, we are going to include the segments 13-1, 13-2 14-1, 14-2 ,18-1 and 18-2 into our edge-table, copying the data for dir,cost,and reverse cost with tho following steps:

- Add a column old_id into edge_table, this column is going to keep track the id of the original edge
- Insert only the segmented edges, that is, the ones whose max(sub_id) >1

```
alter table edge_table drop column if exists old_id;
alter table edge_table add column old_id integer;
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 = segment
                where edges.id in (select old_id from segmented where i>1) );
```

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','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED FOR  24 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

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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 6
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
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, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
```

```
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                            Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
OK
(1 row)
```

Or we can analyze everything because, maybe edge 18 is an overpass, edge 14 is an under pass and there is also a street level juction, and the same happens with edges 17 and 13.

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

NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 3
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 5
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
OK
(1 row)
```

See Also

Routing Topology 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

- genindex
- search

4.1.6 See Also

Indices and tables

- genindex
- search

CHAPTER 5

Routing functions

5.1 Routing Functions

All Pairs - Family of Functions

- *pgr_floydWarshall* - Floyd-Warshall's Algorithm
- *pgr_johnson*- Johnson's Algorithm

pgr_aStar - Shortest Path A*

pgr_bdAstar - Bi-directional A* Shortest Path

pgr_bdDijkstra - Bi-directional Dijkstra Shortest Path

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 - proposed* - Use pgr_dijkstra to create a costs matrix.
- *pgr_drivingDistance* - Use pgr_dijkstra to calculate catchament information.
- *pgr_KSP* - Use Yen algorithm with pgr_dijkstra to get the K shortest paths.
- *pgr_dijkstraVia - Proposed* - Get a route of a seuence of vertices.

pgr_KSP - K-Shortest Path

pgr_trsp - Turn Restriction Shortest Path (TRSP)

Traveling Sales Person - Family of functions

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

Driving Distance - Category

- *pgr_drivingDistance* - Driving Distance based on pgr_dijkstra
- *pgr_withPointsDD - Proposed* - Driving Distance based on pgr_withPoints
- Post pocessing
  - *pgr_alphaShape* - Alpha shape computation
  - *pgr_pointsAsPolygon* - Polygon around a set of points

5.1.1 All Pairs - Family of Functions

The following functions work on all vertices pair combinations

pgr_floydWarshall

Synopsis

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

[1]

Fig. 5.1: Boost Graph Inside

Availability: 2.0.0

- Renamed on 2.2.0, previous name pgr_apspWarshall

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,

Characteristics

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.**

Signature Summary

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

Signatures

Minimal Signature

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

Example 1 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 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)
```

Description of the Signatures

Description of the edges_sql query (id is not necessary)

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures Receives `(edges_sql, directed)`

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

Description of the return values 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`.

History

- Re-design of pgr_apspWarshall in Version 2.2.0

See Also

- *pgr_johnson*
- Boost floyd-Warshall[2] algorithm
- Queries uses the Sample Data network.

[2] http://www.boost.org/libs/graph/doc/floyd_warshall_shortest.html

Indices and tables

- genindex
- search

pgr_johnson

Synopsis

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

[3]

Fig. 5.2: Boost Graph Inside

Availability: 2.0.0

- Renamed on 2.2.0, previous name pgr_apspJohnson

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

Characteristics

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.

Signature Summary

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

Signatures

Minimal Signature

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

Example 1 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 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)
```

Description of the Signatures

Description of the edges_sql query (id is not necessary)

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures Receives `(edges_sql, directed)`

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

Description of the return values 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`.

History

- Re-design of pgr_apspJohnson in Version 2.2.0

See Also

- *pgr_floydWarshall*
- Boost Johnson[4] algorithm implementation.
- Queries uses the Sample Data network.

[4] http://www.boost.org/libs/graph/doc/johnson_all_pairs_shortest.html

Indices and tables

- genindex
- search

Performance

The following tests:

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

Data

The following data was used

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

Data processing was done with osm2pgrouting-alpha

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

Results

Test One

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

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

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

The results of this tests are presented as:

SIZE is the number of edges given as input.

EDGES is the total number of records in the query.

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

OUT ROWS is the number of records returned by the queries.

Floyd-Warshall is the average execution time in seconds of pgr_floydWarshall.

Johnson is the average execution time in seconds of pgr_johnson.

SIZE	EDGES	DENSITY	OUT ROWS	Floyd-Warshall	Johnson
500	500	0.18E-7	1346	0.14	0.13
1000	1000	0.36E-7	2655	0.23	0.18
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
```

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
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[5] algorithm

Indices and tables

- genindex
- search

5.1.2 pgr_bdAstar

Name

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

[6]

Fig. 5.3: Boost Graph Inside

Availability:

- pgr_bdAstar(one to one) 2.0.0, Signature change on 2.5.0
- pgr_bdAstar(other signatures) 2.5.0

[5] http://www.boost.org/libs/graph/doc/floyd_warshall_shortest.html

Signature Summary

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

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

```
pgr_bdAstar(edges_sql, start_vid, end_vids [, directed, heuristic, factor, epsilon])
pgr_bdAstar(edges_sql, start_vids, end_vid [, directed, heuristic, factor, epsilon])
pgr_bdAstar(edges_sql, start_vids, end_vids [, directed, heuristic, factor, epsilon])

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

Using these signatures, will load once the graph and perform several one to one *pgr_bdAstar*

- The result is the union of the results of the one to one *pgr_bdAStar*.
- The extra `start_vid` and/or `end_vid` in the result is used to distinguish to which path it belongs.

Avaliability

- pgr_bdAstar(one to one) 2.0, signature change on 2.5
- pgr_bdAstar(other signatures) 2.5

Signatures

Minimal Signature

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

This usage finds the shortest path from the `start_vid` to the `end_vid`

- on a **directed** graph
- with **heuristic**'s value 5
- with **factor**'s value 1
- with **epsilon**'s value 1

Example Using the defaults

```
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)
```

pgr_bdAstar One to One

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

This usage finds the shortest path from the `start_vid` to the `end_vid` allowing the user to choose

- **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Note: In the One to One signature, because of the deprecated signature existence, it is compulsory to indicate if the graph is **directed** or **undirected**.

Example Directed 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)
```

pgr_bdAstar One to many

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

This usage finds the shortest path from the `start_vid` to each `end_vid` in `end_vids` allowing the user to choose

- if the graph is **directed** or **undirected**

- and/or **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Directed using Heuristic 3 and a factor of 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)
```

pgr_bdAstar Many to One

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

This usage finds the shortest path from each `start_vid` in `start_vids` to the `end_vid` allowing the user to choose

- if the graph is **directed** or **undirected**
- and/or **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Undirected graph with 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 |    4 |    1 |        2
   6 |        4 |         7 |    2 |    2 |    1 |        3
   7 |        5 |         7 |    3 |   -1 |    0 |        4
(7 rows)
```

pgr_bdAstar Many to Many

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

This usage finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids` allowing the use

- if the graph is **directed** or **undirected**
- and/or **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Directed graph with a factor of 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 |   10 |    1 |        2
  21 |        4 |         7 |      11 |   10 |   12 |    1 |        3
  22 |        5 |         7 |      11 |   11 |   -1 |    0 |        4
(22 rows)
```

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type	Description
edges_sql	`TEXT`	Edges SQL query as described above.
start_vid	`ANY-INTEGER`	Starting vertex identifier.
start_vids	`ARRAY[ANY-INTEGER]`	Starting vertices identifierers.
end_vid	`ANY-INTEGER`	Ending vertex identifier.
end_vids	`ARRAY[ANY-INTEGER]`	Ending vertices identifiers.
directed	`BOOLEAN`	• Optional. – When `false` the graph is considered as Undirected. – Default is `true` which considers the graph as Directed.
heuristic	`INTEGER`	(optional). Heuristic number. Current valid values 0~5. Default `5` • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	`FLOAT`	(optional). For units manipulation. $factor > 0$. Default `1`. see *Factor*
epsilon	`FLOAT`	(optional). For less restricted results. $epsilon >= 1$. Default `1`.

Description of the return values for a path

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

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

See Also

- *Bidirectional A* - Family of functions*

- Sample Data network.
- http://www.boost.org/libs/graph/doc/astar_search.html
- http://en.wikipedia.org/wiki/A*_search_algorithm

Indices and tables

- genindex
- search

5.1.3 pgr_bdDijkstra

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

[7]

Fig. 5.4: Boost Graph Inside

Availability:

- pgr_bdDijkstra(one to one) 2.0.0, Signature changed 2.4.0
- pgr_bdDijkstra(other signatures) 2.5.0

Signature Summary

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

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

```
pgr_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
```

Signatures

Minimal signature

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

The minimal signature is for a **directed** graph from one `start_vid` to one `end_vid`:

Example

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

pgr_bdDijkstra 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
```

This signature finds the shortest path from one `start_vid` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
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)
```

pgr_bdDijkstra 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
```

This signature finds the shortest path from one `start_vid` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform a one to one *pgr_dijkstra* where the starting vertex is fixed, and stop when all `end_vids` are reached.

- The result is equivalent to the union of the results of the one to one *pgr_dijkstra*.
- The extra `end_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

pgr_bdDijkstra 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
```

This signature finds the shortest path from each `start_vid` in `start_vids` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to one *pgr_dijkstra* where the ending vertex is fixed.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

pgr_bdDijkstra 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
```

This signature finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to Many *pgr_dijkstra* for all `start_vids`.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

The extra `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

Example

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

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for a path

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

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

See Also

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

Indices and tables

- genindex
- search

5.1.4 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 - proposed* - Use pgr_dijkstra to create a costs matrix.
- *pgr_drivingDistance* - Use pgr_dijkstra to calculate catchament information.
- *pgr_KSP* - Use Yen algorithm with pgr_dijkstra to get the K shortest paths.
- *pgr_dijkstraVia - Proposed* - Get a route of a seuence of vertices.

pgr_dijkstra

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

Availability

- pgr_dijkstra(one to one) 2.0.0, signature change 2.1.0
- pgr_dijkstra(other signatures) 2.1.0

[8]

Fig. 5.5: Boost Graph Inside

Synopsis

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.

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 ∞
- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: $O(|start\_vids| * (V \log V + E))$

Signature Summary

```
pgr_dijkstra(edges_sql, start_vid,  end_vid)
pgr_dijkstra(edges_sql, start_vid,  end_vid,  directed:=true)
pgr_dijkstra(edges_sql, start_vid,  end_vids, directed:=true)
pgr_dijkstra(edges_sql, start_vids, end_vid,  directed:=true)
pgr_dijkstra(edges_sql, start_vids, end_vids, directed:=true)

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

Signatures

Minimal signature

```
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
```

The minimal signature is for a **directed** graph from one `start_vid` to one `end_vid`.

Example

```
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)
```

pgr_dijkstra 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
```

This signature finds the shortest path from one `start_vid` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
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)
```

pgr_dijkstra 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
```

This signature finds the shortest path from one `start_vid` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform a one to one *pgr_dijkstra* where the starting vertex is fixed, and stop when all `end_vids` are reached.

- The result is equivalent to the union of the results of the one to one *pgr_dijkstra*.
- The extra `end_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

pgr_dijkstra 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
```

This signature finds the shortest path from each `start_vid` in `start_vids` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to one *pgr_dijkstra* where the ending vertex is fixed.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

pgr_dijkstra 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
```

This signature finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to Many *pgr_dijkstra* for all `start_vids`.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

The extra `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Default
sql	`TEXT`	
start_vid	`BIGINT`	
start_vids	`ARRAY[BIGINT]`	
end_vid	`BIGINT`	
end_vids	`ARRAY[BIGINT]`	
directed	`BOOLEAN`	`true`

Description of the return values for a path Returns set of `(seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)`

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

Additional Examples

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

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

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

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

```
   6 |        6 |    3 |   -1 |    0 |        5
(6 rows)

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

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

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

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

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

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

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

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

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

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

SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
```

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

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

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

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

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

```
   8 |         1 |          11 |         5 |    11 |    11 |     1 |          0
   9 |         2 |          11 |         5 |     6 |     8 |     1 |          1
  10 |         3 |          11 |         5 |     5 |    -1 |     0 |          2
(10 rows)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Indices and tables

- genindex
- search

pgr_dijkstraCost

Synopsis

`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.

[9]

Fig. 5.6: Boost Graph Inside

Availability

- pgr_dijkstraCost(all signatures) 2.2.0

The `pgr_dijkstraCost` algorithm, is a good choice to calculate the sum of the costs of the shortest path for a subset of pairs of nodes of the graph. We make use of the Boost's implementation of dijkstra which runs in $O(V \log V + E)$ time.

Characteristics

The main Characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of *(start_vid, end_vid, agg_cost)*.

- When the starting vertex and ending vertex are the same, there is no path.
  - The *agg_cost* int the non included values *(v, v)* is *0*
- When the starting vertex and ending vertex are the different and there is no path.
  - The *agg_cost* in the non included values *(u, v)* is ∞

- 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: $O(|start\_vids| * (V \log V + E))$

Signature Summary

```
pgr_dijkstraCost(edges_sql, start_vid, end_vid);
pgr_dijkstraCost(edges_sql, start_vid, end_vid, directed);
pgr_dijkstraCost(edges_sql, start_vids, end_vid, directed);
pgr_dijkstraCost(edges_sql, start_vid, end_vids, directed);
pgr_dijkstraCost(edges_sql, start_vids, end_vids, directed);

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

Signatures

Minimal signature The minimal signature is for a **directed** graph from one `start_vid` to one `end_vid`:

```
pgr_dijkstraCost(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid)
    RETURNS SET OF (start_vid, end_vid, agg_cost) or EMPTY SET
```

Example

```
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)
```

pgr_dijkstraCost One to One

This signature performs a Dijkstra from one `start_vid` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

```
pgr_dijkstraCost(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid,
                     BOOLEAN directed:=true);
    RETURNS SET OF (start_vid, end_vid, agg_cost) or EMPTY SET
```

Example

```
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)
```

pgr_dijkstraCost One to Many

```
pgr_dijkstraCost(TEXT edges_sql, BIGINT start_vid, array[ANY_INTEGER] end_vids,
        BOOLEAN directed:=true);
    RETURNS SET OF (start_vid, end_vid, agg_cost) or EMPTY SET
```

This signature performs a Dijkstra from one `start_vid` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
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)
```

pgr_dijkstraCost Many to One

```
pgr_dijkstraCost(TEXT edges_sql, array[ANY_INTEGER] start_vids, BIGINT end_vid,
                     BOOLEAN directed:=true);
    RETURNS SET OF (start_vid, end_vid, agg_cost) or EMPTY SET
```

This signature performs a Dijkstra from each `start_vid` in `start_vids` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
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)
```

pgr_dijkstraCost Many to Many

```
pgr_dijkstraCost(TEXT edges_sql, array[ANY_INTEGER] start_vids, array[ANY_INTEGER] end_vids,
        BOOLEAN directed:=true);
    RETURNS SET OF (start_vid, end_vid, agg_cost) or EMPTY SET
```

This signature performs a Dijkstra from each `start_vid` in `start_vids` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Default
sql	`TEXT`	
start_vid	`BIGINT`	
start_vids	`ARRAY[BIGINT]`	
end_vid	`BIGINT`	
end_vids	`ARRAY[BIGINT]`	
directed	`BOOLEAN`	`true`

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

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

- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Sample Data network.

Indices and tables

- genindex
- search

pgr_dijkstraCostMatrix - proposed

Name

`pgr_dijkstraCostMatrix` - Calculates the a cost matrix using pgr_dijktras.

Warning: Proposed functions for next mayor release.

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

[10]

Fig. 5.7: Boost Graph Inside

Availability: 2.3.0

Synopsis

Using Dijkstra algorithm, calculate and return a cost matrix.

Signature Summary

```
pgr_dijkstraCostMatrix(edges_sql, start_vids)
pgr_dijkstraCostMatrix(edges_sql, start_vids, directed)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Signatures

Minimal Signature

The minimal signature:

- Is for a **directed** graph.

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

Example Cost matrix for vertices 1, 2, 3, and 4.

```
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:=true)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example Cost matrix for an undirected graph for vertices 1, 2, 3, and 4.

This example returns a symmetric cost matrix.

```
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

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

- genindex
- search

pgr_drivingDistance

Name

`pgr_drivingDistance` - Returns the driving distance from a start node.

[11]

Fig. 5.8: Boost Graph Inside

Availability

- pgr_drivingDistance(single vertex) 2.0.0, signature change 2.1.0
- pgr_drivingDistance(multiple vertices) 2.1.0

Synopsis

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.

Signature Summary

```
pgr_drivingDistance(edges_sql, start_vid, distance)
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)
```

Signatures

Minimal Use

```
pgr_drivingDistance(edges_sql, start_vid, distance)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Driving Distance From A Single Starting Vertex

```
pgr_drivingDistance(edges_sql, start_vid, distance, directed)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Driving Distance From Multiple Starting Vertices

```
pgr_drivingDistance(edges_sql, start_vids, distance, directed, equicost)
RETURNS SET OF (seq, start_vid, node, edge, cost, agg_cost)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Description
edges_sql	TEXT	SQL query as described above.
start_vid	BIGINT	Identifier of the starting vertex.
start_vids	ARRAY[ANY-INTEGER]	Array of identifiers of the starting vertices.
distance	FLOAT	Upper limit for the inclusion of the node in the result
directed	BOOLEAN	(optional). When `false` the graph is considered as U which considers the graph as Directed.
equicost	BOOLEAN	(optional). When `true` the node will only appear in Default is `false` which resembles several calls usin signatures. Tie brakes are arbitrary.

Description of the return values Returns set of `(seq [, start_v], node, edge, cost, agg_cost)`

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
start_vid	INTEGER	Identifier of the starting vertex.
node	BIGINT	Identifier of the node in the path within the limits from `start_vid`.
edge	BIGINT	Identifier of the edge used to arrive to `node`. `0` when the `node` is the `start_vid`.
cost	FLOAT	Cost to traverse `edge`.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `node`.

Additional Examples

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_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)
```

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_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)
```

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_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)
```

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_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
- *pgr_pointsAsPolygon* - Polygon around set of points
- Sample Data network.

Indices and tables

- genindex
- search

pgr_KSP

Name

`pgr_KSP` — Returns the "K" shortest paths.

[12]

Fig. 5.9: Boost Graph Inside

Availability: 2.0.0

- Signature change 2.1.0

Synopsis

The K shortest path routing algorithm based on Yen's algorithm. "K" is the number of shortest paths desired.

Signature Summary

```
pgr_KSP(edges_sql, start_vid, end_vid, K);
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
```

Signatures

Minimal Signature

```
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
```

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
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
start_vid	`BIGINT`	Identifier of the starting vertex.
end_vid	`BIGINT`	Identifier of the ending vertex.
k	`INTEGER`	The desiered number of paths.
directed	`BOOLEAN`	(optional). When `false` the graph is considered as Un which considers the graph as Directed.
heap_paths	`BOOLEAN`	(optional). When `true` returns all the paths stored in th `false` which only returns `k` paths.

Roughly, if the shortest path has `N` edges, the heap will contain about than `N * k` paths for small value of `k` and `k > 1`.

Description of the return values Returns set of `(seq, path_seq, path_id, node, edge, cost, agg_cost)`

Column	Type	Description
seq	`INTEGER`	Sequential value starting from **1**.
path_seq	`INTEGER`	Relative position in the path of `node` and `edge`. Has value **1** for the beginning of a path.
path_id	`BIGINT`	Path identifier. The ordering of the paths For two paths i, j if i < j then agg_cost(i) <= agg_cost(j).
node	`BIGINT`	Identifier of the node in the path.
edge	`BIGINT`	Identifier of the edge used to go from `node` to the next node in the path sequence. `-1` for the last node of the route.
cost	`FLOAT`	Cost to traverse from `node` using `edge` to the next node in the path sequence.
agg_cost	`FLOAT`	Aggregate cost from `start_vid` to `node`.

> **Warning:** During the transition to 3.0, because pgr_ksp version 2.0 doesn't have defined a directed flag nor a heap_path flag, when pgr_ksp is used with only one flag version 2.0 signature will be used.

Additional Examples

Examples 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)
```

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_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 |    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, 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)
```

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_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)
```

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_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

- http://en.wikipedia.org/wiki/K_shortest_path_routing
- Sample Data network.

Indices and tables

- genindex
- search

pgr_dijkstraVia - Proposed

Name

`pgr_dijkstraVia` — Using dijkstra algorithm, it finds the route that goes through a list of vertices.

[13]

Fig. 5.10: Boost Graph Inside

Availability: 2.2.0

Synopsis

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.

Note: This is a proposed function

Signatrue Summary

```
pgr_dijkstraVia(edges_sql, via_vertices)
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
```

Signatures

Minimal Signature

```
pgr_dijkstraVia(edges_sql, via_vertices)
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
    node, edge, cost, agg_cost, route_agg_cost) or EMPTY SET
```

Example Find the route that visits the vertices 1 3 9 in that order

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

Complete Signature

```
pgr_dijkstraVia(edges_sql, via_vertices, directed, strict, U_turn_on_edge)
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
    node, edge, cost, agg_cost, route_agg_cost) or EMPTY SET
```

Example Find the route that visits the vertices 1 3 9 in that order on an undirected graph, avoiding U-turns when possible

```
SELECT * FROM pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 3, 9], false, strict:=true, U_turn_on_edge:=false
);
 seq | path_id | path_seq | start_vid | end_vid | node | edge | cost | agg_cost | route_agg_cost
-----+---------+----------+-----------+---------+------+------+------+----------+----------------
   1 |       1 |        1 |         1 |       3 |    1 |    1 |    1 |        0 |              0
   2 |       1 |        2 |         1 |       3 |    2 |    2 |    1 |        1 |              1
   3 |       1 |        3 |         1 |       3 |    3 |   -1 |    0 |        2 |              2
   4 |       2 |        1 |         3 |       9 |    3 |    5 |    1 |        0 |              2
   5 |       2 |        2 |         3 |       9 |    6 |    9 |    1 |        1 |              3
   6 |       2 |        3 |         3 |       9 |    9 |   -2 |    0 |        2 |              4
(6 rows)
```

Description of the Signature

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type	Default
edges_sql	`TEXT`	
via_vertices	`ARRAY[ANY-INTEGER]`	
directed	`BOOLEAN`	`true`
strict	`BOOLEAN`	`false`
U_turn_on_edge	`BOOLEAN`	`true`

Description of the parameters of the signatures

Parameter	Type	Description
edges_sql	`TEXT`	SQL query as described above.
via_vertices	`ARRAY[ANY-INTEGER]`	Array of vertices identifiers
directed	`BOOLEAN`	(optional) Default is true (is directed). When set to as Undirected
strict	`BOOLEAN`	(optional) ignores if a subsection of the route is m found Default is true (is directed). When set to fal Undirected
U_turn_on_edge	`BOOLEAN`	(optional) Default is true (is directed). When set to as Undirected

Description of the return values Returns set of `(start_vid, end_vid, agg_cost)`

Column	Type	Description
seq	BIGINT	Sequential value starting from 1.
path_pid	BIGINT	Identifier of the path.
path_seq	BIGINT	Sequential value starting from 1 for the path.
start_vid	BIGINT	Identifier of the starting vertex of the path.
end_vid	BIGINT	Identifier of the ending vertex of the path.
node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path. -2 for the last node of the route.
cost	FLOAT	Cost to traverse from `node` using `edge` to the next node in the route sequence.
agg_cost	FLOAT	Total cost from `start_vid` to `end_vid` of the path.
route_agg_cost	FLOAT	Total cost from `start_vid` of `path_pid = 1` to `end_vid` of the current `path_pid`.

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 |    1 |        0 |              0
   2 |       1 |        2 |         1 |       5 |    2 |    4 |    1 |        1 |              1
   3 |       1 |        3 |         1 |       5 |    5 |   -1 |    0 |        2 |              2
   4 |       2 |        1 |         5 |       3 |    5 |    8 |    1 |        0 |              2
   5 |       2 |        2 |         5 |       3 |    6 |    9 |    1 |        1 |              3
   6 |       2 |        3 |         5 |       3 |    9 |   16 |    1 |        2 |              4
   7 |       2 |        4 |         5 |       3 |    4 |    3 |    1 |        3 |              5
   8 |       2 |        5 |         5 |       3 |    3 |   -1 |    0 |        4 |              6
   9 |       3 |        1 |         3 |       9 |    3 |    5 |    1 |        0 |              6
  10 |       3 |        2 |         3 |       9 |    6 |    9 |    1 |        1 |              7
  11 |       3 |        3 |         3 |       9 |    9 |   -1 |    0 |        2 |              8
  12 |       4 |        1 |         9 |       4 |    9 |   16 |    1 |        0 |              8
  13 |       4 |        2 |         9 |       4 |    4 |   -2 |    0 |        1 |              9
(13 rows)
```

Example 2 What's the aggregate cost of the third path?

```
SELECT agg_cost FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE path_id = 3 AND edge <0;
 agg_cost
----------
        2
(1 row)
```

Example 3 What's the route's aggregate cost of the route at the end of the third path?

```
SELECT route_agg_cost FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE path_id = 3 AND edge < 0;
 route_agg_cost
```

```
----------------
               8
(1 row)
```

Example 4 How are the nodes visited in the route?

```
SELECT row_number() over () as node_seq, node
FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE edge <> -1 ORDER BY seq;
 node_seq | node
----------+------
        1 |    1
        2 |    2
        3 |    5
        4 |    6
        5 |    9
        6 |    4
        7 |    3
        8 |    6
        9 |    9
       10 |    4
(10 rows)
```

Example 5 What are the aggregate costs of the route when the visited vertices are reached?

```
SELECT path_id, route_agg_cost FROM  pgr_dijkstraVia(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
    ARRAY[1, 5, 3, 9, 4]
)
WHERE edge < 0;
 path_id | route_agg_cost
---------+----------------
       1 |              2
       2 |              6
       3 |              8
       4 |              9
(4 rows)
```

Example 6 show the route's seq and aggregate cost and a status of "passes in front" or "visits" 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

- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Sample Data network.

Indices and tables

- genindex
- search

The problem definition (Advanced documentation)

Given the following query:

pgr_dijkstra($sql, start_{vid}, end_{vid}, directed$)

where $sql = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $source = \bigcup source_i$,
- $target = \bigcup target_i$,

The graphs are defined as follows:

Directed graph

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

- the set of vertices V
  - $V = source \cup target \cup start_{vid} \cup end_{vid}$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0\} & \text{if } reverse\_cost \neq \end{cases}$

Undirected graph

The weighted undirected graph, $G_u(V, E)$, is definied by:

- the set of vertices V
  - $V = source \cup target \cup start_v vid \cup end_{vid}$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \\ \cup\{(source_i, target_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \text{if } reverse\_cost \neq \end{cases}$

The problem

Given:

- $start_{vid} \in V$ a starting vertex
- $end_{vid} \in V$ an ending vertex
- $G(V,E) = \begin{cases} G_d(V,E) & \text{if6 } directed = true \\ G_u(V,E) & \text{if5 } directed = false \end{cases}$

Then:

- $\boldsymbol{\pi} = \{(path\_seq_i, node_i, edge_i, cost_i, agg\_cost_i)\}$

where:

- $path\_seq_i = i$
- $path\_seq_{|\pi|} = |\pi|$
- $node_i \in V$
- $node_1 = start_{vid}$
- $node_{|\pi|} = end_{vid}$
- $\forall i \neq |\pi|, \quad (node_i, node_{i+1}, cost_i) \in E$
- $edge_i = \begin{cases} id_{(node_i, node_{i+1}, cost_i)} & \text{when } i \neq |\pi| \\ -1 & \text{when } i = |\pi| \end{cases}$
- $cost_i = cost_{(node_i, node_{i+1})}$
- $agg\_cost_i = \begin{cases} 0 & \text{when } i = 1 \\ \displaystyle\sum_{k=1}^{i} cost_{(node_{k-1}, node_k)} & \text{when } i \neq 1 \end{cases}$

In other words: The algorithm returns a the shortest path between $start_{vid}$ and end_{vid} , if it exists, in terms of a sequence of

- $path\_seq$ indicates the relative position in the path of the $node$ or $edge$.
- $cost$ is the cost of the edge to be used to go to the next node.
- agg_cost is the cost from the $start_{vid}$ up to the node.

If there is no path, the resulting set is empty.

See Also

Indices and tables

- genindex
- search

5.1.5 pgr_trsp - Turn Restriction Shortest Path (TRSP)

Name

`pgr_trsp` — Returns the shortest path with support for turn restrictions.

Synopsis

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 *pgr_costResult* (seq, id1, id2, cost) rows, that make up a path.

```
pgr_costResult[] pgr_trsp(sql text, source integer, target integer,
            directed boolean, has_rcost boolean [,restrict_sql text]);
```

```
pgr_costResult[] pgr_trsp(sql text, source_edge integer, source_pos float8,
                target_edge integer, target_pos float8,
            directed boolean, has_rcost boolean [,restrict_sql text]);
```

```
pgr_costResult3[] pgr_trspViaVertices(sql text, vids integer[],
                directed boolean, has_rcost boolean
                [, turn_restrict_sql text]);
```

```
pgr_costResult3[] pgr_trspViaEdges(sql text, eids integer[], pcts float8[],
               directed boolean, has_rcost boolean
               [, turn_restrict_sql text]);
```

Description

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 *pgr_costResult[]*:

seq row sequence

id1 node ID

id2 edge ID (`-1` for the last row)

cost cost to traverse from `id1` using `id2`

History

- New in version 2.0.0

Support for Vias

Warning: The Support for Vias functions are prototypes. Not all corner cases are being considered.

We also have support for vias where you can say generate a from A to B to C, etc. We support both methods above only you pass an array of vertices or and array of edges and percentage position along the edge in two arrays.

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

```
SELECT id, source, target, cost, [,reverse_cost] FROM edge_table
```

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

cost `float8` value, of the edge traversal cost. A negative cost will prevent the edge from being inserted in the graph.

reverse_cost (optional) the cost for the reverse traversal of the edge. This is only used when the `directed` and `has_rcost` parameters are `true` (see the above remark about negative costs).

vids `int4[]` An ordered array of **NODE id** the path will go through from start to end.

directed `true` if the graph is directed

has_rcost if `true`, the `reverse_cost` column of the SQL generated set of rows will be used for the cost of the traversal of the edge in the opposite direction.

restrict_sql (optional) a SQL query, which should return a set of rows with the following columns:

```
SELECT to_cost, target_id, via_path FROM restrictions
```

to_cost `float8` turn restriction cost

target_id `int4` target id

via_path `text` commar separated list of edges in the reverse order of `rule`

Another variant of TRSP allows to specify **EDGE id** together with a fraction to interpolate the position:

eids `int4` An ordered array of **EDGE id** that the path has to traverse

pcts `float8` An array of fractional positions along the respective edges in `eids`, where 0.0 is the start of the edge and 1.0 is the end of the eadge.

Returns set of *pgr_costResult[]*:

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`

History

- Via Support prototypes new in version 2.1.0

Examples

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)
```

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.

See Also

- *pgr_costResult[]*

Indices and tables

- genindex
- search

5.1.6 Traveling Sales Person - Family of functions

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

pgr_TSP

Name

- `pgr_TSP` - Returns a route that visits all the nodes exactly once.

Availability: 2.0.0

- Signature changed 2.3.0

Synopsis

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, what is the shortest possible route that visits each city exactly once and returns to the origin city?

This implementation uses simulated annealing to return the approximate solution when the input is given in the form of matrix cell contents. The matrix information must be symmetrical.

Signature Summary

```
pgr_TSP(matrix_cell_sql)
pgr_TSP(matrix_cell_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)
```

Signatures

Basic Use

```
pgr_TSP(matrix_cell_sql)
RETURNS SETOF (seq, node, cost, agg_cost)
```

Example

Because the documentation examples are auto generated and tested for non changing results, and the default is to have random execution, the example is wrapping the actual call.

```
WITH
query AS (
    SELECT * FROM pgr_TSP(
        $$
        SELECT * FROM pgr_dijkstraCostMatrix(
            'SELECT id, source, target, cost, reverse_cost FROM edge_table',
            (SELECT array_agg(id) FROM edge_table_vertices_pgr WHERE id < 14),
            directed := false
        )
        $$
    )
)
SELECT agg_cost < 20 AS under_20 FROM query WHERE seq = 14;
 under_20
----------
 t
(1 row)
```

Complete Signature

```
pgr_TSP(matrix_cell_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:

```
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)
```

Description of the Signatures

Description of the Matrix Cell SQL query

Column	Type	Description
start_vid	`BIGINT`	Identifier of the starting vertex.
end_vid	`BIGINT`	Identifier of the ending vertex.
agg_cost	`FLOAT`	Cost for going from start_vid to end_vid

Can be Used with:

- *pgr_dijkstraCostMatrix - proposed*
- *pgr_withPointsCostMatrix - proposed*
- *pgr_floydWarshall*
- *pgr_johnson*

To generate a symmetric matrix

- 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.

```
Description Of the Control parameters
.....................................................................

The control parameters are optional, and have a default value.

=============================== =========== ============ ==============================================
Parameter                       Type        Default      Description
=============================== =========== ============ ==============================================
**start_vid**                   ``BIGINT``   `0`         The greedy part of the implementation w
**end_vid**                     ``BIGINT``   `0`         Last visiting vertex before returning t
**max_processing_time**         ``FLOAT``   `+infinity`  Stop the annealing processing when the
**tries_per_temperature**       ``INTEGER`` `500`        Maximum number of times a neighbor(s) i
**max_changes_per_temperature** ``INTEGER`` `60`         Maximum number of times the solution is
**max_consecutive_non_changes** ``INTEGER`` `100`        Maximum number of consecutive times the
**initial_temperature**         ``FLOAT``   `100`        Starting temperature.
**final_temperature**           ``FLOAT``   `0.1`        Ending temperature.
**cooling_factor**              ``FLOAT``   `0.9`        Value between between 0 and 1 (not incl
```

```
**randomize**                    ``BOOLEAN``  `true`        Choose the random seed

                                                            - true: Use current time as seed
                                                            - false: Use `1` as seed. Using this va

================================ ===========  ============  ======================================
```

```
Description of the return columns
...............................................................................

Returns set of ``(seq, node, cost, agg_cost)``

============= =========== =================================================
Column           Type              Description
============= =========== =================================================
**seq**       ``INTEGER`` Row sequence.
**node**      ``BIGINT``  Identifier of the node/coordinate/point.
**cost**      ``FLOAT``   Cost to traverse from the current ``node`` ito the next ``node`` in the
                            - ``0`` for the last row in the path sequence.

**agg_cost**  ``FLOAT``   Aggregate cost from the ``node`` at ``seq = 1`` to the current node.
                            - ``0`` for the first row in the path sequence.

============= =========== =================================================
```

Examples

Example Using with points of interest.

To generate a symmetric matrix:

- the **side** information of pointsOfInterset is ignored by not including it in the query
- and **directed := false**

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_withPointsCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
        'SELECT pid, edge_id, fraction from pointsOfInterest',
        array[-1, 3, 5, 6, -6], directed := false);
    $$,
    start_id := 5,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    5 |    1 |        0
   2 |    6 |    1 |        1
   3 |    3 |  1.6 |        2
   4 |   -1 |  1.3 |      3.6
   5 |   -6 |  0.3 |      4.9
   6 |    5 |    0 |      5.2
(6 rows)
```

The queries use the Sample Data network.

See Also

- *Traveling Sales Person - Family of functions*
- http://en.wikipedia.org/wiki/Traveling_salesman_problem
- http://en.wikipedia.org/wiki/Simulated_annealing

Indices and tables

- genindex
- search

pgr_eucledianTSP

Name

`pgr_eucledianTSP` - Returns a route that visits all the coordinates pairs exactly once.

Availability: 2.3.0

Synopsis

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, what is the shortest possible route that visits each city exactly once and returns to the origin city?

This implementation uses simulated annealing to return the approximate solution when the input is given in the form of coordinates.

Signature Summary

```
pgr_eucledianTSP(coordinates_sql)
pgr_eucledianTSP(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)
```

Signatures

Minimal Signature

```
pgr_eucledianTSP(coordinates_sql)
RETURNS SETOF (seq, node, cost, agg_cost)
```

Example

Because the documentation examples are auto generated and tested for non changing results, and the default is to have random execution, the example is wrapping the actual call.

```
WITH
query AS (
    SELECT * FROM pgr_eucledianTSP(
        $$
        SELECT id, st_X(the_geom) AS x, st_Y(the_geom)AS y  FROM edge_table_vertices_pgr
        $$
    )
)
SELECT agg_cost < 20 AS under_20 FROM query WHERE seq = 18;
 under_20
----------
 t
(1 row)
```

Complete Signature

```
pgr_eucledianTSP(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:

```
SELECT* from pgr_eucledianTSP(
    $$
    SELECT id, st_X(the_geom) AS x, st_Y(the_geom) AS y FROM edge_table_vertices_pgr
    $$,
    tries_per_temperature := 3,
    cooling_factor := 0.5,
    randomize := false
);
 seq | node |       cost        |     agg_cost
-----+------+-------------------+------------------
   1 |    1 |  1.4142135623731 |                0
   2 |    3 |                 1 |  1.4142135623731
   3 |    4 |                 1 | 2.41421356237309
   4 |    9 | 0.58309518948453 | 3.41421356237309
   5 |   16 | 0.58309518948453 | 3.99730875185762
   6 |    6 |                 1 | 4.58040394134215
   7 |    5 |                 1 | 5.58040394134215
   8 |    8 |                 1 | 6.58040394134215
   9 |    7 | 1.58113883008419 | 7.58040394134215
  10 |   14 |    1.499999999999 | 9.16154277142634
  11 |   15 |               0.5 | 10.6615427714253
  12 |   13 |               1.5 | 11.1615427714253
  13 |   17 | 1.11803398874989 | 12.6615427714253
  14 |   12 |                 1 | 13.7795767601752
  15 |   11 |                 1 | 14.7795767601752
  16 |   10 |                 2 | 15.7795767601752
  17 |    2 |                 1 | 17.7795767601752
  18 |    1 |                 0 | 18.7795767601752
(18 rows)
```

Description of the Signatures

Description of the coordinates SQL query

Column	Type	Description
id	`BIGINT`	Identifier of the coordinate. (optional)
x	`FLOAT`	X value of the coordinate.
y	`FLOAT`	Y value of the coordinate.

When the value of **id** is not given then the coordinates will receive an **id** starting from 1, in the order given.

```
Description Of the Control parameters
.....................................................................

The control parameters are optional, and have a default value.

=============================== =========== ============ ===========================================
Parameter                       Type        Default      Description
=============================== =========== ============ ===========================================
**start_vid**                   ``BIGINT``   `0`         The greedy part of the implementation w
**end_vid**                     ``BIGINT``   `0`         Last visiting vertex before returning t
**max_processing_time**         ``FLOAT``   `+infinity`  Stop the annealing processing when the
**tries_per_temperature**       ``INTEGER`` `500`        Maximum number of times a neighbor(s) i
**max_changes_per_temperature** ``INTEGER`` `60`         Maximum number of times the solution is
**max_consecutive_non_changes** ``INTEGER`` `100`        Maximum number of consecutive times the
**initial_temperature**         ``FLOAT``   `100`        Starting temperature.
**final_temperature**           ``FLOAT``   `0.1`        Ending temperature.
**cooling_factor**              ``FLOAT``   `0.9`        Value between between 0 and 1 (not inc
**randomize**                   ``BOOLEAN`` `true`       Choose the random seed

                                                         - true: Use current time as seed
                                                         - false: Use `1` as seed. Using this va

=============================== =========== ============ ===========================================
```

```
Description of the return columns
...............................................................................

Returns set of ``(seq, node, cost, agg_cost)``

============= =========== =================================================
Column           Type              Description
============= =========== =================================================
**seq**       ``INTEGER`` Row sequence.
**node**      ``BIGINT``  Identifier of the node/coordinate/point.
**cost**      ``FLOAT``   Cost to traverse from the current ``node`` ito the next ``node`` in the
                            - ``0`` for the last row in the path sequence.

**agg_cost**  ``FLOAT``   Aggregate cost from the ``node`` at ``seq = 1`` to the current node.
                            - ``0`` for the first row in the path sequence.

============= =========== =================================================
```

Examples

Example Skipping the Simulated Annealing & showing some process information

```
SET client_min_messages TO DEBUG1;
SET
SELECT* from pgr_eucledianTSP(
    $$
    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:  pgr_eucledianTSP 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 |  1.4142135623731 |                0
   2 |    3 |                1 |  1.4142135623731
   3 |    4 |                1 | 2.41421356237309
   4 |    9 | 0.58309518948453 | 3.41421356237309
   5 |   16 | 0.58309518948453 | 3.99730875185762
   6 |    6 |                1 | 4.58040394134215
   7 |    5 |                1 | 5.58040394134215
   8 |    8 |                1 | 6.58040394134215
   9 |    7 | 1.58113883008419 | 7.58040394134215
  10 |   14 |   1.499999999999 | 9.16154277142634
  11 |   15 |              0.5 | 10.6615427714253
  12 |   13 |              1.5 | 11.1615427714253
  13 |   17 | 1.11803398874989 | 12.6615427714253
  14 |   12 |                1 | 13.7795767601752
  15 |   11 |                1 | 14.7795767601752
  16 |   10 |                2 | 15.7795767601752
  17 |    2 |                1 | 17.7795767601752
  18 |    1 |                0 | 18.7795767601752
(18 rows)
```

The queries use the Sample Data network.

History

- New in version 2.3.0

See Also

- *Traveling Sales Person - Family of functions*
- http://en.wikipedia.org/wiki/Traveling_salesman_problem
- http://en.wikipedia.org/wiki/Simulated_annealing

Indices and tables

- genindex
- search

General Information

Origin

The traveling sales person problem was studied in the 18th century by mathematicians Sir William Rowam Hamilton and **Thomas Penyngton Kirkman**.

A discussion about the work of Hamilton & Kirkman can be found in the book **Graph Theory (Biggs et al. 1976)**.

- ISBN-13: 978-0198539162
- ISBN-10: 0198539169

It is believed that the general form of the TSP have been first studied by Kalr Menger in Vienna and Harvard. The problem was later promoted by Hassler, Whitney & Merrill at Princeton. A detailed description about the connection between Menger & Whitney, and the development of the TSP can be found in On the history of combinatorial optimization (till 1960)[14]

Problem Definition

Given a collection of cities and travel cost between each pair, find the cheapest way for visiting all of the cities and returning to the starting point.

Characteristics

- The travel costs are symmetric:
  - traveling costs from city A to city B are just as much as traveling from B to A.
- This problem is an NP-hard optimization problem.
- To calculate the number of different tours through n cities:
  - Given a starting city,
  - There are $n - 1$ choices for the second city,
  - And $n - 2$ choices for the third city, etc.
  - Multiplying these together we get $(n - 1)! = (n - 1)(n - 2)..1$.
  - Now since our travel costs do not depend on the direction we take around the tour:
    * this number by 2
    * $(n - 1)!/2$.

TSP & Simulated Annealing

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. *[C001]*

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 **snew** is calculated. The higher the temperature the higher the probability of accepting the new solution as a possible bester solution.

[14] http://www.cwi.nl/ lex/files/histco.ps

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 {
        snew = neighbour(solution);
        If P(E(solution), E(snew), T) >= random(0, 1)
            solution = snew;
    }

    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.

To cool down faster to the next temperature:

- max_changes_per_temperature: limits the number of changes in the solution per temperature
- max_consecutive_non_changes: limits the number of consecutive non changes per temperature

This is done by doing some book keeping on the times **solution = snew;** is executed.

- max_changes_per_temperature: Increases by one when **solution** changes
- max_consecutive_non_changes: Reset to 0 when **solution** changes, and increased each **try**

Additionally to stop the algorithm at a higher temperature than the desired one:

- max_processing_time: limits the time the simulated annealing is performed.
- book keeping is done to see if there was a change in **solution** on the last temperature

Note that, if no change was found in the first **max_consecutive_non_changes** tries, then the simulated annealing will stop.

```
Solution = initial_solution;

temperature = initial_temperature;
while (temperature > final_temperature) {

    do tries_per_temperature times {
        snew = neighbour(solution);
        If P(E(solution), E(snew), T) >= random(0, 1)
            solution = snew;

        when max_changes_per_temperature is reached
            or max_consecutive_non_changes is reached
            BREAK;
    }

    temperature = temperature * cooling factor;
    when no changes were done in the current temperature
        or max_processing_time has being reached
        BREAK;
}

Output: the best solution
```

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.

- Your computational time is crucial, then put your time limit to **max_processing_time**.
- Make the **tries_per_temperture** depending on the number of cities, for example:
  - Useful to estimate the time it takes to do one cycle: use *1*
    - this will help to set a reasonable **max_processing_time**
  - $n * (n - 1)$
  - $500 * n$
- For a faster decreasing the temperature set **cooling_factor** to a smaller number, and set to a higher number for a slower decrease.
- When for the same given data the same results are needed, set **randomize** to *false*.
  - When estimating how long it takes to do one cycle: use *false*

A recommendation is to play with the values and see what fits to the particular data.

Description Of the Control parameters

The control parameters are optional, and have a default value.

Parameter	Type	Default	Description
start_vid	`BIGINT`	*0*	The greedy part of the implementation will use this identifier.
end_vid	`BIGINT`	*0*	Last visiting vertex before returning to start_vid.
max_processing_time	`FLOAT`	*+infinity*	Stop the annealing processing when the value is reached.
tries_per_temperature	`INTEGER`	*500*	Maximum number of times a neighbor(s) is searched in each temperature.
max_changes_per_temperature	`INTEGER`	*60*	Maximum number of times the solution is changed in each temperature.
max_consecutive_non_changes	`INTEGER`	*100*	Maximum number of consecutive times the solution is not changed in each temperature.
initial_temperature	`FLOAT`	*100*	Starting temperature.
final_temperature	`FLOAT`	*0.1*	Ending temperature.
cooling_factor	`FLOAT`	*0.9*	Value between between 0 and 1 (not including) used to calculate the next temperature.
randomize	`BOOLEAN`	*true*	Choose the random seed • true: Use current time as seed • false: Use *1* as seed. Using this value will get the same results with the same data in each execution.

Description of the return columns

Returns set of `(seq, node, cost, agg_cost)`

Column	Type	Description
seq	`INTEGER`	Row sequence.
node	`BIGINT`	Identifier of the node/coordinate/point.
cost	`FLOAT`	**Cost to traverse from the current `node` ito the ne** • `0` for the last row in the path sequence.
agg_cost	`FLOAT`	**Aggregate cost from the `node` at `seq = 1` to th** • `0` for the first row in the path sequence.

See Also

References

- http://en.wikipedia.org/wiki/Traveling_salesman_problem
- http://en.wikipedia.org/wiki/Simulated_annealing

Indices and tables

- genindex
- search

5.1.7 Driving Distance - Category

- *pgr_drivingDistance* - Driving Distance based on pgr_dijkstra
- *pgr_withPointsDD - Proposed* - Driving Distance based on pgr_withPoints
- Post pocessing
  - *pgr_alphaShape* - Alpha shape computation
  - *pgr_pointsAsPolygon* - Polygon around a set of points

pgr_alphaShape

Name

`pgr_alphaShape` — Core function for alpha shape computation.

Synopsis

Returns a table with (x, y) rows that describe the vertices of an alpha shape.

```
table() pgr_alphaShape(text sql [, float8 alpha]);
```

Description

sql `text` a SQL query, which should return a set of rows with the following columns:

```
SELECT id, x, y FROM vertex_table
```

id `int4` identifier of the vertex

x `float8` x-coordinate

y `float8` y-coordinate

alpha (optional) `float8` alpha value. If specified alpha value equals 0 (default), then optimal alpha value is used. For more information, see CGAL - 2D Alpha Shapes[15].

Returns a vertex record for each row:

x x-coordinate

y y-coordinate

[15] http://doc.cgal.org/latest/Alpha_shapes_2/group__PkgAlphaShape2.html

If a result includes multiple outer/inner rings, return those with separator row (x=NULL and y=NULL).

History

- Renamed in version 2.0.0
- Added alpha argument with default 0 (use optimal value) in version 2.1.0
- Supported to return multiple outer/inner ring coordinates with separator row (x=NULL and y=NULL) in version 2.1.0

Examples

PgRouting's alpha shape implementation has no way to control the order of the output points, so the actual output might different for the same input data. The first query, has the output ordered, he second query shows an example usage:

Example: the (ordered) results

```
SELECT * FROM pgr_alphaShape(
    'SELECT id::integer, ST_X(the_geom)::float AS x, ST_Y(the_geom)::float AS y
    FROM edge_table_vertices_pgr') ORDER BY x, y;
  x  |  y
-----+-----
   0 |   2
 0.5 | 3.5
   2 |   0
   2 |   4
 3.5 |   4
   4 |   1
   4 |   2
   4 |   3
(8 rows)
```

Example: calculating the area

Steps:

- **Calculates the alpha shape**
  - the `ORDER BY` clause is not used.
- constructs a polygon
- and computes the area

```
SELECT round(ST_Area(ST_MakePolygon(ST_AddPoint(foo.openline, ST_StartPoint(foo.openline))))::nume
FROM (
    SELECT ST_MakeLine(points ORDER BY id) AS openline
    FROM (
        SELECT ST_MakePoint(x, y) AS points, row_number() over() AS id
        FROM pgr_alphaShape(
            'SELECT id::integer, ST_X(the_geom)::float AS x, ST_Y(the_geom)::float AS y
            FROM edge_table_vertices_pgr')
        ) AS a
    ) AS foo;
 st_area
---------
   11.75
```

```
(1 row)
```

The queries use the Sample Data network.

See Also

- *pgr_drivingDistance* - Driving Distance
- *pgr_pointsAsPolygon* - Polygon around set of points

Indices and tables

- genindex
- search

pgr_pointsAsPolygon

Name

`pgr_pointsAsPolygon` — Draws an alpha shape around given set of points.

Synopsis

Returns the alpha shape as (multi)polygon geometry.

```
geometry pgr_pointsAsPolygon(text sql [, float8 alpha]);
```

Description

sql `text` a SQL query, which should return a set of rows with the following columns:

```
        SELECT id, x, y FROM vertex_result;
```

> **id** `int4` identifier of the vertex
>
> **x** `float8` x-coordinate
>
> **y** `float8` y-coordinate

alpha (optional) `float8` alpha value. If specified alpha value equals 0 (default), then optimal alpha value is used. For more information, see CGAL - 2D Alpha Shapes[16].

Returns a (multi)polygon geometry (with holes).

History

- Renamed in version 2.0.0
- Added alpha argument with default 0 (use optimal value) in version 2.1.0
- Supported to return a (multi)polygon geometry (with holes) in version 2.1.0

[16] http://doc.cgal.org/latest/Alpha_shapes_2/group__PkgAlphaShape2.html

Examples

In the following query there is no way to control which point in the polygon is the first in the list, so you may get similar but different results than the following which are also correct.

```
SELECT ST_AsText(pgr_pointsAsPolygon('SELECT id::integer, ST_X(the_geom)::float AS x, ST_Y(the_ge
        FROM edge_table_vertices_pgr'));
                      st_astext
------------------------------------------------------
 POLYGON((2 4,3.5 4,4 3,4 2,4 1,2 0,0 2,0.5 3.5,2 4))
(1 row)
```

The query use the Sample Data network.

See Also

- *pgr_drivingDistance* - Driving Distance
- *pgr_alphaShape* - Alpha shape computation

Indices and tables

- genindex
- search

See Also

Indices and tables

- genindex
- search

5.1.8 See Also

Indices and tables

- genindex
- search

All Pairs - Family of Functions

- *pgr_floydWarshall* - Floyd-Warshall's Algorithm
- *pgr_johnson*- Johnson's Algorithm

pgr_aStar - Shortest Path A*

pgr_bdAstar - Bi-directional A* Shortest Path

pgr_bdDijkstra - Bi-directional Dijkstra Shortest Path

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 - proposed* - Use pgr_dijkstra to create a costs matrix.

- *pgr_drivingDistance* - Use pgr_dijkstra to calculate catchament information.
- *pgr_KSP* - Use Yen algorithm with pgr_dijkstra to get the K shortest paths.
- *pgr_dijkstraVia - Proposed* - Get a route of a seuence of vertices.

pgr_KSP - K-Shortest Path

pgr_trsp - Turn Restriction Shortest Path (TRSP)

Traveling Sales Person - Family of functions

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

Driving Distance - Category

- *pgr_drivingDistance* - Driving Distance based on pgr_dijkstra
- *pgr_withPointsDD - Proposed* - Driving Distance based on pgr_withPoints
- Post pocessing
  - *pgr_alphaShape* - Alpha shape computation
  - *pgr_pointsAsPolygon* - Polygon around a set of points

Available Functions but not official pgRouting functions

- *Stable Proposed Functions*
- *Experimental Functions*

6.1 Stable Proposed Functions

Warning: Proposed functions for next mayor release.

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

As part of the *Dijkstra - Family of functions*

- *pgr_dijkstraCostMatrix - proposed* Use pgr_dijkstra to calculate a cost matrix.
- *pgr_dijkstraVia - Proposed* - Use pgr_dijkstra to make a route via vertices.

Families

aStar - Family of functions

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

Bidirectional A - Family of functions*

- *pgr_bdAstar* - Bidirectional A* algorithm for obtaining paths.
- *pgr_bdAstarCost - Proposed* - Bidirectional A* algorithm to calculate the cost of the paths.
- *pgr_bdAstarCostMatrix - proposed* - 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 - Proposed* - Bidirectional Dijkstra to calculate the cost of the shortest paths

- *pgr_bdDijkstraCostMatrix - proposed* - Bidirectional Dijkstra algorithm to create a matrix of costs of the shortest paths.

Flow - Family of functions

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

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_aStarCost – proposed*
- *pgr_bdAstarCost - Proposed*
- *pgr_bdDijkstraCost - Proposed*
- *pgr_dijkstraCost*
- *pgr_withPointsCost - Proposed*

Cost Matrix - Category

- *pgr_aStarCostMatrix - proposed*
- *pgr_bdAstarCostMatrix - proposed*
- *pgr_bdDijkstraCostMatrix - proposed*
- *pgr_dijkstraCostMatrix - proposed*
- *pgr_withPointsCostMatrix - proposed*

KSP Category

- *pgr_KSP* - Driving Distance based on pgr_dijkstra
- *pgr_withPointsKSP - Proposed* - Driving Distance based on pgr_dijkstra

6.1.1 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 – proposed* - Get the aggregate cost of the shortest paths.

- *pgr_aStarCostMatrix - proposed* - Get the cost matrix of the shortest paths.

pgr_aStar

Name

`pgr_aStar` — Returns the shortest path using A* algorithm.

[1]

Fig. 6.1: Boost Graph Inside

Availability:

- pgr_astar(one to one) 2.0.0, Signature changed 2.3.0
- pgr_astar(other signatures) 2.4.0

Characteristics

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

Signature Summary

```
pgr_aStar(edges_sql, start_vid, end_vid)
pgr_aStar(edges_sql, start_vid, end_vid, directed, heuristic, factor, epsilon)
```

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

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

Signatures

Minimal Signature

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

Example Using the defaults

```
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 |   10 |    1 |        1
   3 |        3 |   10 |   12 |    1 |        2
   4 |        4 |   11 |   13 |    1 |        3
   5 |        5 |   12 |   -1 |    0 |        4
(5 rows)
```

One to One

```
pgr_aStar(edges_sql, start_vid, end_vid, directed, heuristic, factor, epsilon)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Example Undirected using Heuristic 2

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

One to many

```
pgr_aStar(edges_sql, start_vid, end_vids, directed, heuristic, factor, epsilon) -- Proposed
RETURNS SET OF (seq, path_seq, end_vid, node, edge, cost, agg_cost) or EMPTY SET
```

This signature finds the shortest path from one `start_vid` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform a one to one *pgr_astar* where the starting vertex is fixed, and stop when all `end_vids` are reached.

- The result is equivalent to the union of the results of the one to one *pgr_astar*.
- The extra `end_vid` in the result is used to distinguish to which path it belongs.

Example

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

Many to One

```
pgr_aStar(edges_sql, starts_vid, end_vid, directed, heuristic, factor, epsilon) -- Proposed
RETURNS SET OF (seq, path_seq, start_vid, node, edge, cost, agg_cost) or EMPTY SET
```

This signature finds the shortest path from each `start_vid` in `start_vids` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to one *pgr_aStar* where the ending vertex is fixed.

- The result is the union of the results of the one to one *pgr_aStar*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

Example

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

```
   6 |        1 |         7 |     7 |    6 |    1 |        0
   7 |        2 |         7 |     8 |    7 |    1 |        1
   8 |        3 |         7 |     5 |   10 |    1 |        2
   9 |        4 |         7 |    10 |   12 |    1 |        3
  10 |        5 |         7 |    11 |   13 |    1 |        4
  11 |        6 |         7 |    12 |   -1 |    0 |        5
(11 rows)
```

Many to Many

```
pgr_aStar(edges_sql, starts_vid, end_vids, directed, heuristic, factor, epsilon) -- Proposed
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost) or EMPTY SET
```

This signature finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to Many *pgr_dijkstra* for all `start_vids`.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

The extra `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

Example

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

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	`TEXT`
start_vid	`ANY-INTEGER`
end_vid	`ANY-INTEGER`
directed	`BOOLEAN`
heuristic	`INTEGER`
factor	`FLOAT`
epsilon	`FLOAT`

Description of the return values for a path Returns set of `(seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)`

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

See Also

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

Indices and tables

- genindex
- search

pgr_aStarCost – proposed

Name

`pgr_aStarCost` — Returns the aggregate cost shortest path using *aStar - Family of functions* algorithm.

[2]

Fig. 6.2: Boost Graph Inside

Availability: 2.4.0

Signature Summary

Warning: Proposed functions for next mayor release.

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

```
pgr_aStarCost(edges_sql, start_vid, end_vid) -- Proposed
pgr_aStarCost(edges_sql, start_vid, end_vid, directed, heuristic, factor, epsilon) -- Proposed
pgr_aStarCost(edges_sql, start_vid, end_vids, directed, heuristic, factor, epsilon) -- Proposed
pgr_aStarCost(edges_sql, starts_vid, end_vid, directed, heuristic, factor, epsilon) -- Proposed
pgr_aStarCost(edges_sql, starts_vid, end_vids, directed, heuristic, factor, epsilon) -- Proposed

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

Signatures

Minimal Signature

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

Example Using the defaults

```
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, start_vid, end_vid, directed, heuristic, factor, epsilon)
RETURNS SET OF (start_vid, end_vid, agg_cost) OR EMPTY SET
```

Example Setting a Heuristic

```
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, start_vid, end_vids, directed, heuristic, factor, epsilon) -- Proposed
RETURNS SET OF (start_vid, end_vid, agg_cost) OR EMPTY SET
```

This signature finds a path from one `start_vid` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform a one to one *pgr_astar* where the starting vertex is fixed, and stop when all `end_vids` are reached.

- The result is equivalent to the union of the results of the one to one *pgr_astar*.
- The extra `end_vid` column in the result is used to distinguish to which path it belongs.

Example

```
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, starts_vid, end_vid, directed, heuristic, factor, epsilon) -- Proposed
RETURNS SET OF (start_vid, end_vid, agg_cost) OR EMPTY SET
```

This signature finds the shortest path from each `start_vid` in `start_vids` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to one *pgr_aStar* where the ending vertex is fixed.

- The result is the union of the results of the one to one *pgr_aStar*.

- The extra `start_vid` column in the result is used to distinguish to which path it belongs.

  Example

```
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, starts_vid, end_vids, directed, heuristic, factor, epsilon) -- Proposed
RETURNS SET OF (start_vid, end_vid, agg_cost) OR EMPTY SET
```

This signature finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to Many *pgr_dijkstra* for all `start_vids`.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

The extra `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	`TEXT`
start_vid	`ANY-INTEGER`
end_vid	`ANY-INTEGER`
directed	`BOOLEAN`
heuristic	`INTEGER`
factor	`FLOAT`
epsilon	`FLOAT`

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

See Also

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

Indices and tables

- genindex
- search

pgr_aStarCostMatrix - proposed

Name

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

Warning: Proposed functions for next mayor release.

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

[3]

Fig. 6.3: Boost Graph Inside

Availability: 2.4.0

Synopsis

Using aStar algorithm, calculate and return a cost matrix.

Signature Summary

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

Signatures

Minimal Signature

The minimal signature:

- Is for a **directed** graph.

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

Example Cost matrix for vertices 1, 2, 3, and 4.

```
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
-----------+---------+----------
         2 |       1 |        1
         3 |       1 |        2
         4 |       1 |        3
```

```
         1 |         2 |          1
         3 |         2 |          1
         4 |         2 |          2
         1 |         3 |          6
         2 |         3 |          5
         4 |         3 |          1
         1 |         4 |          5
         2 |         4 |          4
         3 |         4 |          3
(12 rows)
```

Complete Signature

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

Example Cost matrix for an undirected graph for vertices 1, 2, 3, and 4.

This example returns a symmetric cost matrix.

```
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
-----------+---------+----------
         2 |         1 |          1
         3 |         1 |          2
         4 |         1 |          3
         1 |         2 |          1
         3 |         2 |          1
         4 |         2 |          2
         1 |         3 |          2
         2 |         3 |          1
         4 |         3 |          1
         1 |         4 |          3
         2 |         4 |          2
         3 |         4 |          1
(12 rows)
```

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	TEXT
vids	ARRAY[ANY-INTEGER]
directed	BOOLEAN
heuristic	INTEGER
factor	FLOAT
epsilon	FLOAT

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

Examples

Example Use with 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 Matrix - Category*
- *Traveling Sales Person - Family of functions*
- The queries use the Sample Data network.

Indices and tables

- genindex
- search

The problem definition (Advanced documentation)

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

Heuristic

Currently the heuristic functions available are:

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

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

Factor

Analysis 1

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

Analysis 2

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

latitude	conversion	Factor
45	1 longitude degree is 78846.81 m	78846
0	1 longitude degree is 111319.46 m	111319

Analysis 3

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

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

See Also

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

Indices and tables

- genindex
- search

6.1.2 Bidirectional A* - Family of functions

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

pgr_bdAstarCost - Proposed

Name

`pgr_bdAstarCost` — Returns the shortest path using A* algorithm.

[4]

Fig. 6.4: Boost Graph Inside

Availability: 2.5.0

Warning: Experimental functions
- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Signature Summary

```
pgr_bdAstarCost(edges_sql, start_vid, end_vid)
pgr_bdAstarCost(edges_sql, start_vid, end_vid [, directed , heuristic, factor, epsilon])
pgr_bdAstarCost(edges_sql, start_vid, end_vids [, directed, heuristic, factor, epsilon])
pgr_bdAstarCost(edges_sql, start_vids, end_vid [, directed, heuristic, factor, epsilon])
pgr_bdAstarCost(edges_sql, start_vids, end_vids [, directed, heuristic, factor, epsilon])

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

Using these signatures, will load once the graph and perform several one to one *pgr_bdAstarCost*

- The result is the union of the results of the one to one *pgr_bdAstarCost*.
- The extra `start_vid` and/or `end_vid` in the result is used to distinguish to which path it belongs.

Signatures

Minimal Signature

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

This usage finds the shortest path from the `start_vid` to the `end_vid`

- on a **directed** graph
- with **heuristic**'s value 5
- with **factor**'s value 1
- with **epsilon**'s value 1

Example Using the defaults

```
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)
```

pgr_bdAstarCost One to One

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

This usage finds the shortest path from the `start_vid` to each `end_vid` in `end_vids` allowing the user to choose

- if the graph is **directed** or **undirected**
- **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Note: In the One to One signature, because of the deprecated signature existence, it is compulsory to indicate if the graph is **directed** or **undirected**.

Example Directed 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)
```

pgr_bdAstarCost One to many

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

This usage finds the shortest path from the `start_vid` to each `end_vid` in `end_vids` allowing the user to choose

- if the graph is **directed** or **undirected**
- and/or **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Directed using Heuristic 3 and a factor of 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)
```

pgr_bdAstarCost Many to One

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

This usage finds the shortest path from each `start_vid` in `start_vids` to the `end_vid` allowing the user to choose

- if the graph is **directed** or **undirected**
- and/or **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Undirected graph with 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)
```

pgr_bdAstarCost Many to Many

```
pgr_bdAstarCost(edges_sql, start_vids, end_vids [, directed, heuristic, factor, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

This usage finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids` allowing the use

- if the graph is **directed** or **undirected**
- and/or **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Directed graph with a factor of 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)
```

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	`TEXT`
start_vid	`ANY-INTEGER`
start_vids	`ARRAY[ANY-INTEGER]`
end_vid	`ANY-INTEGER`
end_vids	`ARRAY[ANY-INTEGER]`
directed	`BOOLEAN`
heuristic	`INTEGER`
factor	`FLOAT`
epsilon	`FLOAT`

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

See Also

- *Bidirectional A* - Family of functions*
- Sample Data network.
- Migration Guide[5]
- http://www.boost.org/libs/graph/doc/astar_search.html
- http://en.wikipedia.org/wiki/A*_search_algorithm

Indices and tables

- genindex
- search

[5] https://github.com/cvvergara/pgrouting/wiki/Migration-Guide#pgr_bdastar

pgr_bdAstarCostMatrix - proposed

Name

`pgr_bdAstarCostMatrix` - Calculates the a cost matrix using *pgr_bdAstar*.

[6]

Fig. 6.5: Boost Graph Inside

Availability: 2.5.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

Using Dijkstra algorithm, calculate and return a cost matrix.

Signature Summary

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

Signatures

Minimal Signature

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

This usage calculates the cost from the each `start_vid` in `start_vids` to each `start_vid` in `start_vids`

- on a **directed** graph
- with **heuristic**'s value 5
- with **factor**'s value 1
- with **epsilon**'s value 1

Example Cost matrix for vertices 1, 2, 3, and 4.

```
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, start_vids, [, directed , heuristic, factor, epsilon])
RETURNS SET OF (start_vid, end_vid, agg_cost)
OR EMPTY SET
```

This usage calculates the cost from the each `start_vid` in `start_vids` to each `start_vid` in `start_vids` allowing th

- if the graph is **directed** or **undirected**
- **heuristic**,
- and/or **factor**
- and/or **epsilon**.

Example Cost matrix for an undirected graph for vertices 1, 2, 3, and 4.

This example returns a symmetric cost matrix.

```
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)
```

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	TEXT
start_vid	ANY-INTEGER
start_vids	ARRAY[ANY-INTEGER]
end_vid	ANY-INTEGER
end_vids	ARRAY[ANY-INTEGER]
directed	BOOLEAN
heuristic	INTEGER
factor	FLOAT
epsilon	FLOAT

Description of the return values for a Cost function Returns set of (start_vid, end_vid, agg_cost)

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

Examples

Example Use with 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

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

Indices and tables

- genindex
- search

Synopsis

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.

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 ∞
- Running time (worse case scenario): $O((E + V) * \log V)$
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_astar

Description of the Signatures

Description of the edges_sql query for astar like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Parameter	Type	Description
edges_sql	`TEXT`	Edges SQL query as described above.
start_vid	`ANY-INTEGER`	Starting vertex identifier.
start_vids	`ARRAY[ANY-INTEGER]`	Starting vertices identifierers.
end_vid	`ANY-INTEGER`	Ending vertex identifier.
end_vids	`ARRAY[ANY-INTEGER]`	Ending vertices identifiers.
directed	`BOOLEAN`	• Optional. – When `false` the graph is considered as Undirected. – Default is `true` which considers the graph as Directed.
heuristic	`INTEGER`	(optional). Heuristic number. Current valid values 0~5. Default `5` • 0: h(v) = 0 (Use this value to compare with pgr_dijkstra) • 1: h(v) abs(max(dx, dy)) • 2: h(v) abs(min(dx, dy)) • 3: h(v) = dx * dx + dy * dy • 4: h(v) = sqrt(dx * dx + dy * dy) • 5: h(v) = abs(dx) + abs(dy)
factor	`FLOAT`	(optional). For units manipulation. $factor > 0$. Default `1`. see *Factor*
epsilon	`FLOAT`	(optional). For less restricted results. $epsilon >= 1$. Default `1`.

See Also

Indices and tables

- genindex
- search

6.1.3 Bidirectional Dijkstra - Family of functions

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

pgr_bdDijkstraCost - Proposed

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

[7]

Fig. 6.6: Boost Graph Inside

Availability: 2.5.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Signature Summary

```
pgr_dijkstraCost(edges_sql, start_vid,  end_vid)
pgr_bdDijkstraCost(edges_sql, start_vid, end_vid, directed)
pgr_bdDijkstraCost(edges_sql, start_vid, end_vids, directed)
pgr_bdDijkstraCost(edges_sql, start_vids, end_vid, directed)
pgr_bdDijkstraCost(edges_sql, start_vids, end_vids, directed)

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

Signatures

Minimal signature

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

The minimal signature is for a **directed** graph from one `start_vid` to one `end_vid`:

Example

```
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)
```

pgr_bdDijkstraCost One to One

```
pgr_bdDijkstraCost(edges_sql, start_vid, end_vid, directed)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

This signature finds the shortest path from one `start_vid` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
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)
```

pgr_bdDijkstraCost 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
```

This signature finds the shortest path from one `start_vid` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform a one to one *pgr_dijkstra* where the starting vertex is fixed, and stop when all `end_vids` are reached.

- The result is equivalent to the union of the results of the one to one *pgr_dijkstra*.
- The extra `end_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

pgr_bdDijkstraCost 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
```

This signature finds the shortest path from each `start_vid` in `start_vids` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to one *pgr_dijkstra* where the ending vertex is fixed.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

Example

```
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)
```

pgr_bdDijkstraCost 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
```

This signature finds the shortest path from each `start_vid` in `start_vids` to each `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Using this signature, will load once the graph and perform several one to Many *pgr_dijkstra* for all `start_vids`.

- The result is the union of the results of the one to one *pgr_dijkstra*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

The extra `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

Example

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

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

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Default
sql	`TEXT`	
start_vid	`BIGINT`	
start_vids	`ARRAY[BIGINT]`	
end_vid	`BIGINT`	
end_vids	`ARRAY[BIGINT]`	
directed	`BOOLEAN`	`true`

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

See Also

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

Indices and tables

- genindex
- search

pgr_bdDijkstraCostMatrix - proposed

Name

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

[8]

Fig. 6.7: Boost Graph Inside

Availability: 2.5.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

Using Dijkstra algorithm, calculate and return a cost matrix.

Signature Summary

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

Signatures

Minimal Signature

The minimal signature:

- Is for a **directed** graph.

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

Example Cost matrix for vertices 1, 2, 3, and 4.

```
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:=true)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example Cost matrix for an undirected graph for vertices 1, 2, 3, and 4.

This example returns a symmetric cost matrix.

```
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

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

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

Indices and tables

- genindex
- search

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 ∞
- Running time (worse case scenario): $O((V \log V + E))$
- For large graphs where there is a path bewtween the starting vertex and ending vertex:
  - It is expected to terminate faster than pgr_dijkstra

See Also

Indices and tables

- genindex

- search

6.1.4 withPoints - Family of functions

When points are also given as input:

- *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

Name

`pgr_withPoints` - Returns the shortest path in a graph with additional temporary vertices.

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

[9]

Fig. 6.8: Boost Graph Inside

Availability: 2.2.0

Synopsis

Modify the graph to include points defined by points_sql. Using Dijkstra algorithm, find the shortest path(s)

Characteristics:

The main Characteristics are:

- Process is done only on edges with positive costs.
- Vertices of the graph are:
  - **positive** when it belongs to the edges_sql
  - **negative** when it belongs to the points_sql

- Values are returned when there is a path.
  - When the starting vertex and ending vertex are the same, there is no path. - The agg_cost the non included values (v, v) is 0
  - When the starting vertex and ending vertex are the different and there is no path: - The agg_cost the non included values (u, v) is ∞
- For optimization purposes, any duplicated value in the start_vids or end_vids are ignored.
- The returned values are ordered: - start_vid ascending - end_vid ascending
- Running time: $O(|start\_vids| \times (V \log V + E))$

Signature Summary

```
pgr_withPoints(edges_sql, points_sql, start_vid, end_vid)
pgr_withPoints(edges_sql, points_sql, start_vid, end_vid, directed, driving_side, details)
pgr_withPoints(edges_sql, points_sql, start_vid, end_vids, directed, driving_side, details)
pgr_withPoints(edges_sql, points_sql, start_vids, end_vid, directed, driving_side, details)
pgr_withPoints(edges_sql, points_sql, start_vids, end_vids, directed, driving_side, details)
RETURNS SET OF (seq, path_seq, [start_vid,] [end_vid,] node, edge, cost, agg_cost)
```

Signatures

Minimal Use

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.
- No **details** are given about distance of other points of points_sql query.

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

Example From point 1 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',
    -1, -3);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |   -1 |    1 |  0.6 |        0
   2 |        2 |    2 |    4 |    1 |      0.6
   3 |        3 |    5 |   10 |    1 |      1.6
   4 |        4 |   10 |   12 |  0.6 |      2.6
   5 |        5 |   -3 |   -1 |    0 |      3.2
(5 rows)
```

One to One

```
pgr_withPoints(edges_sql, points_sql, start_vid, end_vid,
    directed:=true, driving_side:='b', details:=false)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Example From point 1 to vertex 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',
    -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, start_vid, end_vids,
    directed:=true, driving_side:='b', details:=false)
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, start_vids, end_vid,
    directed:=true, driving_side:='b', details:=false)
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, start_vids, end_vids,
    directed:=true, driving_side:='b', details:=false)
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the Points SQL query

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

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

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	TEXT
points_sql	TEXT
start_vid	ANY-INTEGER
end_vid	ANY-INTEGER
start_vids	ARRAY[ANY-INTEGER]
end_vids	ARRAY[ANY-INTEGER]
directed	BOOLEAN
driving_side	CHAR
details	BOOLEAN

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

Column	Type	Description
seq	`INTEGER`	Row sequence.
path_seq	`INTEGER`	Path sequence that indicates the relative position on the path.
start_vid	`BIGINT`	Identifier of the starting vertex. When negative: is a point's pid.
end_vid	`BIGINT`	Identifier of the ending vertex. When negative: is a point's pid.
node	`BIGINT`	**Identifier of the node:** • A positive value indicates the node is a vertex of edges_sql. • A negative value indicates the node is a point of points_sql.
edge	`BIGINT`	**Identifier of the edge used to go from `node` to th** • `-1` for the last row in the path sequence.
cost	`FLOAT`	**Cost to traverse from `node` using `edge` to the ne** • `0` for the last row in the path sequence.
agg_cost	`FLOAT`	**Aggregate cost from `start_pid` to `node`.** • `0` for the first row in the path sequence.

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_a
        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_a
        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 Many to many example with a twist: on undirected graph and showing 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.

History

- Proposed in version 2.2

See Also

- *withPoints - Family of functions*

Indices and tables

- genindex
- search

pgr_withPointsCost - Proposed

Name

`pgr_withPointsCost` - Calculates the shortest path and returns only the aggregate cost of the shortest path(s) found, for the combination of points given.

Warning: Proposed functions for next mayor release.

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

[10]

Fig. 6.9: Boost Graph Inside

Availability: 2.2.0

Synopsis

Modify the graph to include points defined by points_sql. Using Dijkstra algorithm, return only the aggregate cost of the shortest path(s) found.

Characteristics:

The main Characteristics are:

- It does not return a path.
- Returns the sum of the costs of the shortest path for pair combination of vertices in the modified graph.
- Vertices of the graph are:
  - **positive** when it belongs to the edges_sql
  - **negative** when it belongs to the points_sql
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of *(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* in the non included values *(v, v)* is *0*
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values *(u, v)* is ∞
- If the values returned are stored in a table, the unique index would be the pair: *(start_vid, end_vid)*.
- For undirected graphs, the results are symmetric.
  - The *agg_cost* of *(u, v)* is the same as for *(v, u)*.
- For optimization purposes, any duplicated value in the *start_vids* or *end_vids* is ignored.
- The returned values are ordered:
  - *start_vid* ascending

- *end_vid* ascending

- Running time: $O(|start\_vids| * (V \log V + E))$

Signature Summary

```
pgr_withPointsCost(edges_sql, points_sql, start_vid, end_vid, directed, driving_side)
pgr_withPointsCost(edges_sql, points_sql, start_vid, end_vids, directed, driving_side)
pgr_withPointsCost(edges_sql, points_sql, start_vids, end_vid, directed, driving_side)
pgr_withPointsCost(edges_sql, points_sql, start_vids, end_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.

Signatures

Minimal Use

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_withPointsCost(edges_sql, points_sql, start_vid, end_vid)
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example

```
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, start_vid, end_vid,
    directed:=true, driving_side:='b')
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example

```
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, start_vid, end_vids,
    directed:=true, driving_side:='b')
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example

```
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, start_vids, end_vid,
    directed:=true, driving_side:='b')
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example

```
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, start_vids, end_vids,
    directed:=true, driving_side:='b')
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the Points SQL query

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

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

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	TEXT
points_sql	TEXT
start_vid	ANY-INTEGER
end_vid	ANY-INTEGER
start_vids	ARRAY[ANY-INTEGER]
end_vids	ARRAY[ANY-INTEGER]
directed	BOOLEAN
driving_side	CHAR

Description of the return values Returns set of `(start_vid, end_vid, agg_cost)`

Column	Type	Description
start_vid	BIGINT	Identifier of the starting vertex. When negative: is a point's pid.
end_vid	BIGINT	Identifier of the ending point. When negative: is a point's pid.
agg_cost	FLOAT	Aggregate cost from `start_vid` to `end_vid`.

Examples

Example 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 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 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.

History

- Proposed in version 2.2

See Also

- *withPoints - Family of functions*

Indices and tables

- genindex
- search

pgr_withPointsCostMatrix - proposed

Name

`pgr_withPointsCostMatrix` - Calculates the shortest path and returns only the aggregate cost of the shortest path(s) found, for the combination of points given.

Warning: Proposed functions for next mayor release.

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

[11]

Fig. 6.10: Boost Graph Inside

Availability: 2.2.0

Signature Summary

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vids)
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.

Signatures

Minimal Signature

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

```
SELECT * FROM pgr_withPointsCostMatrix(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction from pointsOfInterest',
    array[-1, 3, 6, -6]);
 start_vid | end_vid | agg_cost
-----------+---------+----------
        -6 |      -1 |      1.3
        -6 |       3 |      4.3
        -6 |       6 |      1.3
        -1 |      -6 |      1.3
        -1 |       3 |      5.6
        -1 |       6 |      2.6
         3 |      -6 |      1.7
         3 |      -1 |      1.6
```

```
          3 |        6 |         1
          6 |       -6 |       1.3
          6 |       -1 |       2.6
          6 |        3 |         3
(12 rows)
```

Complete Signature

```
pgr_withPointsCostMatrix(edges_sql, points_sql, start_vids,
    directed:=true, driving_side:='b')
RETURNS SET OF (start_vid, end_vid, agg_cost)
```

Example returning a symmetrical cost matrix

- Using the default **side** value on the **points_sql** query
- Using an undirected graph
- 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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the Points SQL query

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

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

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	`TEXT`
points_sql	`TEXT`
start_vids	`ARRAY[ANY-INTEGER]`
directed	`BOOLEAN`
driving_side	`CHAR`

Description of the return values for a Cost function Returns set of `(start_vid, end_vid, agg_cost)`

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

Examples

Example Use with tsp

```
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

- *withPoints - Family of functions*
- *Cost Matrix - Category*

- *Traveling Sales Person - Family of functions*
- *sampledata* network.

Indices and tables

- genindex
- search

pgr_withPointsKSP - Proposed

Name

`pgr_withPointsKSP` - Find the K shortest paths using Yen's algorithm.

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

[12]

Fig. 6.11: Boost Graph Inside

Availability: 2.2.0

Synopsis

Modifies the graph to include the points defined in the `points_sql` and using Yen algorithm, finds the K shortest paths.

Signature Summary

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K)
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K, directed, heap_paths, driving_sid
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

Signatures

Minimal Usage

The minimal usage:

- 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.
- No **details** are given about distance of other points of the query.
- No **heap paths** are returned.

```
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

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -2, 2);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.6 |        0
   2 |       1 |        2 |    2 |    4 |    1 |      0.6
   3 |       1 |        3 |    5 |    8 |    1 |      1.6
   4 |       1 |        4 |    6 |    9 |    1 |      2.6
   5 |       1 |        5 |    9 |   15 |  0.4 |      3.6
   6 |       1 |        6 |   -2 |   -1 |    0 |        4
   7 |       2 |        1 |   -1 |    1 |  0.6 |        0
   8 |       2 |        2 |    2 |    4 |    1 |      0.6
   9 |       2 |        3 |    5 |    8 |    1 |      1.6
  10 |       2 |        4 |    6 |   11 |    1 |      2.6
  11 |       2 |        5 |   11 |   13 |    1 |      3.6
  12 |       2 |        6 |   12 |   15 |  0.6 |      4.6
  13 |       2 |        7 |   -2 |   -1 |    0 |      5.2
(13 rows)
```

Complete Signature Finds the K shortest paths depending on the optional parameters setup.

```
pgr_withPointsKSP(edges_sql, points_sql, start_pid, end_pid, K,
    directed:=true, heap_paths:=false, driving_side:='b', details:=false)
RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost)
```

Example With details.

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, 6, 2, details := true);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.6 |        0
   2 |       1 |        2 |    2 |    4 |  0.7 |      0.6
   3 |       1 |        3 |   -6 |    4 |  0.3 |      1.3
   4 |       1 |        4 |    5 |    8 |    1 |      1.6
   5 |       1 |        5 |    6 |   -1 |    0 |      2.6
   6 |       2 |        1 |   -1 |    1 |  0.6 |        0
   7 |       2 |        2 |    2 |    4 |  0.7 |      0.6
   8 |       2 |        3 |   -6 |    4 |  0.3 |      1.3
   9 |       2 |        4 |    5 |   10 |    1 |      1.6
  10 |       2 |        5 |   10 |   12 |  0.6 |      2.6
  11 |       2 |        6 |   -3 |   12 |  0.4 |      3.2
  12 |       2 |        7 |   11 |   13 |    1 |      3.6
  13 |       2 |        8 |   12 |   15 |  0.6 |      4.6
  14 |       2 |        9 |   -2 |   15 |  0.4 |      5.2
  15 |       2 |       10 |    9 |    9 |    1 |      5.6
  16 |       2 |       11 |    6 |   -1 |    0 |      6.6
```

```
(16 rows)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the Points SQL query

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

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

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	`TEXT`
points_sql	`TEXT`
start_pid	`ANY-INTEGER`
end_pid	`ANY-INTEGER`
K	`INTEGER`
directed	`BOOLEAN`
heap_paths	`BOOLEAN`
driving_side	`CHAR`
details	`BOOLEAN`

Description of the return values Returns set of (seq, path_id, path_seq, node, edge, cost, agg_cost)

Column	Type	Description
seq	INTEGER	Row sequence.
path_seq	INTEGER	Relative position in the path of node and edge. Has value 1 for the beginning of a path.
path_id	INTEGER	Path identifier. The ordering of the paths: For two paths i, j if i < j then agg_cost(i) <= agg_cost(j).
node	BIGINT	Identifier of the node in the path. Negative values are the identifiers of a point.
edge	BIGINT	**Identifier of the edge used to go from node to th** • -1 for the last row in the path sequence.
cost	FLOAT	**Cost to traverse from node using edge to the ne** • 0 for the last row in the path sequence.
agg_cost	FLOAT	**Aggregate cost from start_pid to node.** • 0 for the first row in the path sequence.

Examples

Example Left side driving topology 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 with heap paths and details.

```
SELECT * FROM pgr_withPointsKSP(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table ORDER BY id',
    'SELECT pid, edge_id, fraction, side from pointsOfInterest',
    -1, -2, 2,
    heap_paths := true, driving_side := 'r', details := true);
 seq | path_id | path_seq | node | edge | cost | agg_cost
-----+---------+----------+------+------+------+----------
   1 |       1 |        1 |   -1 |    1 |  0.4 |        0
   2 |       1 |        2 |    1 |    1 |    1 |      0.4
   3 |       1 |        3 |    2 |    4 |  0.7 |      1.4
   4 |       1 |        4 |   -6 |    4 |  0.3 |      2.1
   5 |       1 |        5 |    5 |    8 |    1 |      2.4
   6 |       1 |        6 |    6 |    9 |    1 |      3.4
   7 |       1 |        7 |    9 |   15 |  0.4 |      4.4
   8 |       1 |        8 |   -2 |   -1 |    0 |      4.8
   9 |       2 |        1 |   -1 |    1 |  0.4 |        0
  10 |       2 |        2 |    1 |    1 |    1 |      0.4
  11 |       2 |        3 |    2 |    4 |  0.7 |      1.4
  12 |       2 |        4 |   -6 |    4 |  0.3 |      2.1
  13 |       2 |        5 |    5 |    8 |    1 |      2.4
  14 |       2 |        6 |    6 |   11 |    1 |      3.4
  15 |       2 |        7 |   11 |   13 |    1 |      4.4
  16 |       2 |        8 |   12 |   15 |    1 |      5.4
  17 |       2 |        9 |    9 |   15 |  0.4 |      6.4
  18 |       2 |       10 |   -2 |   -1 |    0 |      6.8
  19 |       3 |        1 |   -1 |    1 |  0.4 |        0
  20 |       3 |        2 |    1 |    1 |    1 |      0.4
  21 |       3 |        3 |    2 |    4 |  0.7 |      1.4
  22 |       3 |        4 |   -6 |    4 |  0.3 |      2.1
  23 |       3 |        5 |    5 |   10 |    1 |      2.4
  24 |       3 |        6 |   10 |   12 |  0.6 |      3.4
  25 |       3 |        7 |   -3 |   12 |  0.4 |        4
  26 |       3 |        8 |   11 |   13 |    1 |      4.4
  27 |       3 |        9 |   12 |   15 |    1 |      5.4
  28 |       3 |       10 |    9 |   15 |  0.4 |      6.4
  29 |       3 |       11 |   -2 |   -1 |    0 |      6.8
(29 rows)
```

The queries use the Sample Data network.

History

- Proposed in version 2.2

See Also

- *withPoints - Family of functions*

Indices and tables

- genindex
- search

pgr_withPointsDD - Proposed

Name

`pgr_withPointsDD` - Returns the driving distance from a starting point.

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

[13]

Fig. 6.12: Boost Graph Inside

Availability: 2.2.0

Synopsis

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.

Signature Summary

```
pgr_withPointsDD(edges_sql, points_sql, start_vid, distance)
pgr_withPointsDD(edges_sql, points_sql, start_vid, distance, directed, driving_side, details)
pgr_withPointsDD(edges_sql, points_sql, start_vids, distance, directed, driving_side, details, eq
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Signatures

Minimal Use

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.
- No **details** are given about distance of other points of the query.

```
pgr_withPointsDD(edges_sql, points_sql, start_vid, distance)
    directed:=true, driving_side:='b', details:=false)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example

```
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)
```

Driving distance from a single point Finds the driving distance depending on the optional parameters setup.

```
pgr_withPointsDD(edges_sql, points_sql, start_vids, distance,
    directed:=true, driving_side:='b', details:=false)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Example Right side driving topology

```
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)
```

Driving distance from many starting points Finds the driving distance depending on the optional parameters setup.

```
pgr_withPointsDD(edges_sql, points_sql, start_vids, distance,
    directed:=true, driving_side:='b', details:=false, equicost:=false)
RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the Points SQL query

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

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

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

Parameter	Type
edges_sql	TEXT
points_sql	TEXT
start_vid	ANY-INTEGER
distance	ANY-NUMERICAL
directed	BOOLEAN
driving_side	CHAR
details	BOOLEAN
equicost	BOOLEAN

Description of the return values Returns set of `(seq, node, edge, cost, agg_cost)`

Column	Type	Description
seq	INT	row sequence.
node	BIGINT	Identifier of the node within the Distance from `start_pid`. If `details =: true` a negative value is the identifier of a point.
edge	BIGINT	**Identifier of the edge used to go from `node` to th** • `-1` when `start_vid` = `node`.
cost	FLOAT	**Cost to traverse `edge`.** • `0` when `start_vid` = `node`.
agg_cost	FLOAT	**Aggregate cost from `start_vid` to `node`.** • `0` when `start_vid` = `node`.

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

```
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 Does not matter driving side.

```
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.

History

- Proposed in version 2.2

See Also

- *pgr_drivingDistance* - Driving distance using dijkstra.
- *pgr_alphaShape* - Alpha shape computation.
- *pgr_pointsAsPolygon* - Polygon around set of points.

Indices and tables

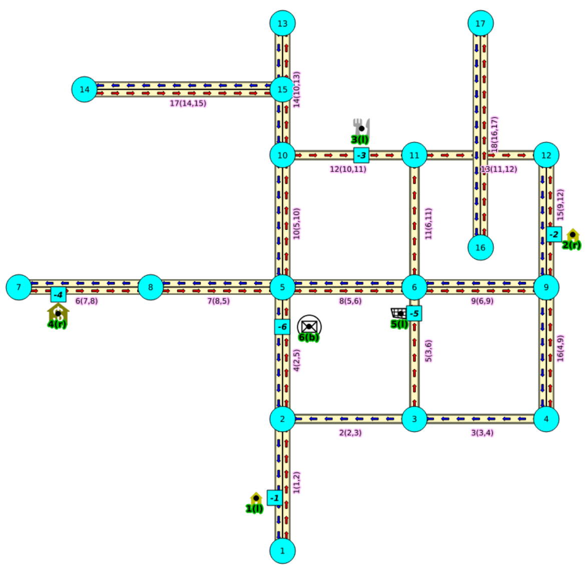
- genindex
- search

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

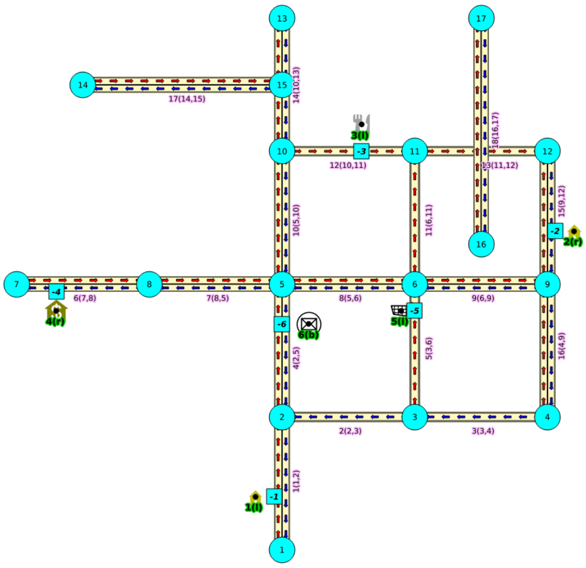
Images

The squared vertices are the temporary vertices, The temporary vertices are added according to the driving side, The following images visually show the differences on how depending on the driving side the data is interpreted.

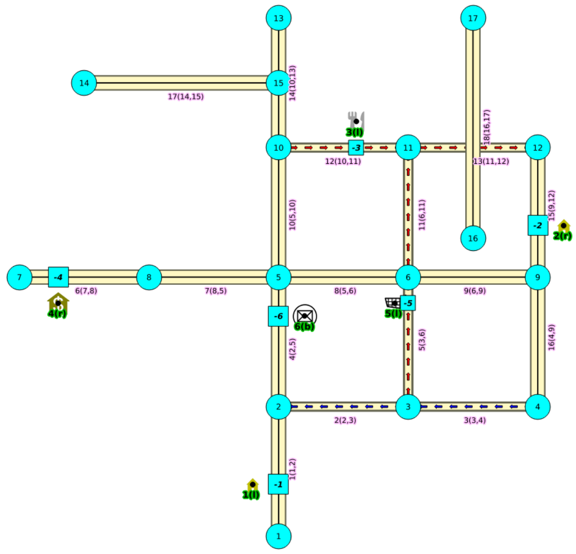
Right driving side



Left driving side



doesn't matter the driving side



Introduction

This family of functions was thought for routing vehicles, but might as well work for some other application that we can not think of.

The with points family of function give you the ability to route between arbitrary points located outside the original graph.

When given a point identified with a *pid* that its being mapped to and edge with an identifier *edge_id*, with a *fraction* along that edge (from the source to the target of the edge) and some additional information about which *side* of the edge the point is on, then routing from arbitrary points more accurately reflect routing vehicles in road networks,

I talk about a family of functions because it includes different functionalities.

- pgr_withPoints is pgr_dijkstra based
- pgr_withPointsCost is pgr_dijkstraCost based
- pgr_withPointsKSP is pgr_ksp based
- pgr_withPointsDD is pgr_drivingDistance based

In all this functions we have to take care of as many aspects as possible:

- Must work for routing:
  - Cars (directed graph)
  - Pedestrians (undirected graph)
- Arriving at the point:
  - In either side of the street.
  - Compulsory arrival on the side of the street where the point is located.
- Countries with:
  - Right side driving
  - Left side driving
- Some points are:
  - Permanent, for example the set of points of clients stored in a table in the data base
  - Temporal, for example points given through a web application
- The numbering of the points are handled with negative sign.
  - Original point identifiers are to be positive.
  - Transformation to negative is done internally.
  - For results for involving vertices identifiers
    * positive sign is a vertex of the original graph
    * negative sign is a point of the temporary points

The reason for doing this is to avoid confusion when there is a vertex with the same number as identifier as the points identifier.

Graph & edges

- Let $G_d(V,E)$ where V is the set of vertices and E is the set of edges be the original directed graph.
  - An edge of the original *edges_sql* is $(id, source, target, cost, reverse\_cost)$ will generate internally
    * $(id, source, target, cost)$
    * $(id, target, source, reverse\_cost)$

Point Definition

- A point is defined by the quadruplet: $(pid, eid, fraction, side)$
  - **pid** is the point identifier
  - **eid** is an edge id of the *edges_sql*
  - **fraction** represents where the edge *eid* will be cut.
  - **side** Indicates the side of the edge where the point is located.

Creating Temporary Vertices in the Graph

For edge (15, 9,12 10, 20), & lets insert point (2, 12, 0.3, r)

On a right hand side driving network

From first image above:

- We can arrive to the point only via vertex 9.
- It only affects the edge (15, 9,12, 10) so that edge is removed.
- Edge (15, 12,9, 20) is kept.
- Create new edges:
  - (15, 9,-1, 3) edge from vertex 9 to point 1 has cost 3
  - (15, -1,12, 7) edge from point 1 to vertex 12 has cost 7

On a left hand side driving network

From second image above:

- We can arrive to the point only via vertex 12.
- It only affects the edge (15, 12,9 20) so that edge is removed.
- Edge (15, 9,12, 10) is kept.
- Create new edges:
  - (15, 12,-1, 14) edge from vertex 12 to point 1 has cost 14
  - (15, -1,9, 6) edge from point 1 to vertex 9 has cost 6

  Remember that fraction is from vertex 9 to vertex 12

When driving side does not matter

From third image above:

- We can arrive to the point either via vertex 12 or via vertex 9
- Edge (15, 12,9 20) is removed.
- Edge (15, 9,12, 10) is removed.
- Create new edges:
  - (15, 12,-1, 14) edge from vertex 12 to point 1 has cost 14
  - (15, -1,9, 6) edge from point 1 to vertex 9 has cost 6
  - (15, 9,-1, 3) edge from vertex 9 to point 1 has cost 3
  - (15, -1,12, 7) edge from point 1 to vertex 12 has cost 7

See Also

Indices and tables

- genindex
- search

6.1.5 Cost - Category

- *pgr_aStarCost – proposed*
- *pgr_bdAstarCost - Proposed*
- *pgr_bdDijkstraCost - Proposed*
- *pgr_dijkstraCost*
- *pgr_withPointsCost - Proposed*

Warning: Proposed functions for next mayor release.

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

General Information

Characteristics

The main Characteristics are:

- Each function works as part of the family it belongs to.
- It does not return a path.
- Returns the sum of the costs of the resulting path(s) for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of *(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* int the non included values *(v, v)* is *0*.
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values *(u, v)* is ∞.
- 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

- genindex
- search

6.1.6 Cost Matrix - Category

- *pgr_aStarCostMatrix - proposed*
- *pgr_bdAstarCostMatrix - proposed*
- *pgr_bdDijkstraCostMatrix - proposed*
- *pgr_dijkstraCostMatrix - proposed*
- *pgr_withPointsCostMatrix - proposed*

Warning: Proposed functions for next mayor release.

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

General Information

Synopsis

Traveling Sales Person - Family of functions needs as input a symmetric cost matrix and no edge *(u, v)* must value ∞.

This collection of functions will return a cost matrix in form of a table.

Characteristics

The main Characteristics are:

- Can be used as input to *pgr_TSP*.
  - **directly** when the resulting matrix is symmetric and there is no ∞ value.
  - It will be the users responsibility to make the matrix symmetric.
    - By using geometric or harmonic average of the non symmetric values.
    - By using max or min the non symmetric values.
    - By setting the upper triangle to be the mirror image of the lower triangle.
    - By setting the lower triangle to be the mirror image of the upper triangle.
  - It is also the users responsibility to fix an ∞ value.
- Each function works as part of the family it belongs to.
- It does not return a path.

- Returns the sum of the costs of the shortest path for pair combination of nodes in the graph.
- Process is done only on edges with positive costs.
- Values are returned when there is a path.
  - The returned values are in the form of a set of *(start_vid, end_vid, agg_cost)*.
  - When the starting vertex and ending vertex are the same, there is no path.
    - The *agg_cost* int the non included values *(v, v)* is *0*.
  - When the starting vertex and ending vertex are the different and there is no path.
    - The *agg_cost* in the non included values *(u, v)* is ∞.
- Let be the case the values returned are stored in a table, so the unique index would be the pair: *(start_vid, end_vid)*.
- Depending on the function and its parameters, the results can be symmetric.
  - The *agg_cost* of *(u, v)* is the same as for *(v, u)*.
- Any duplicated value in the *start_vids* are ignored.
- The returned values are ordered:
  - *start_vid* ascending
  - *end_vid* ascending
- Running time: approximately $O(|start\_vids| * (V \log V + E))$

See Also

- *pgr_TSP*

Indices and tables

- genindex
- search

6.1.7 KSP Category

- *pgr_KSP* - Driving Distance based on pgr_dijkstra
- *pgr_withPointsKSP - Proposed* - Driving Distance based on pgr_dijkstra

Indices and tables

- genindex
- search

6.2 Experimental Functions

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Contraction - Family of functions - Reduce network size using contraction techniques

- *pgr_contractGraph - Experimental* - Reduce network size using contraction techniques

Graph Analysis

- *pgr_labelGraph - Experimental* - Analyze / label subgraphs within a network

Components - Family of functions - Analyze components within a graph

- *pgr_connectedComponents - Experimental* - Return the connected components of an undirected graph
- *pgr_strongComponents - Experimental* - Return the strongly connected components of a directed graph
- *pgr_biconnectedComponents - Experimental* - Return the biconnected components of an undirected graph
- *pgr_articulationPoints - Experimental* - Return the articulation points of an undirected graph
- *pgr_bridges - Experimental* - Return the bridges of an undirected graph

VRP

- *pgr_gsoc_vrppdtw - Experimental*
- *pgr_vrpOneDepot - Experimental*

6.2.1 Contraction - Family of functions

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

pgr_contractGraph - Experimental

pgr_contractGraph - Experimental

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

[14]

Fig. 6.13: Boost Graph Inside

Availability: 2.3.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

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.

Characteristics

The main Characteristics are:

- Process is done only on edges with positive costs.
- There are two types of contraction methods used namely,
  - Dead End Contraction
  - Linear Contraction
- The values returned include the added edges and contracted vertices.
- The returned values are ordered as follows:
  - column *id* ascending when type = *v*
  - column *id* descending when type = *e*

Signature Summary:

The pgr_contractGraph function has the following signatures:

```
pgr_contractGraph(edges_sql, contraction_order)
pgr_contractGraph(edges_sql, contraction_order, max_cycles, forbidden_vertices, directed)

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

Signatures

Minimal signature

```
pgr_contractGraph(edges_sql, contraction_order)
```

Example Making a dead end contraction and a linear contraction.

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

Complete signature

```
pgr_contractGraph(edges_sql, contraction_order, max_cycles, forbidden_vertices, directed)
```

Example Making a dead end contraction and a linear contraction and vertex 2 is forbidden from contraction

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

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
contraction_order	`ARRAY[ANY-INTEGER]`	**Ordered contraction operations.** • 1 = Dead end contraction • 2 = Linear contraction
forbidden_vertices	`ARRAY[ANY-INTEGER]`	(optional). Identifiers of vertices forbidden from contraction. Default is an empty array.
max_cycles	`INTEGER`	(optional). Number of times the contraction operations on *contraction_order* will be performed. Default is 1.
directed	`BOOLEAN`	• When `true` the graph is considered as *Directed*. • When `false` the graph is considered as *Undirected*.

Description of the return values

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

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

Column	Type	Description
seq	`INTEGER`	Sequential value starting from **1**.
type	`TEXT`	**Type of the *id*.** • 'v' when *id* is an identifier of a vertex. • 'e' when *id* is an identifier of an edge.
id	`BIGINT`	**Identifier of:** • the vertex when *type* = *'v'*. – The vertex belongs to the edge_table passed as a parameter. • the edge when *type* = *'e'*. – The *id* is a decreasing sequence starting from **-1**. – Representing a pseudo *id* as is not incorporated into the edge_table.
contracted_vertices	`ARRAY[BIGINT]`	Array of contracted vertex identifiers.
source	`BIGINT`	Identifier of the source vertex of the current edge *id*. Valid values when *type* = *'e'*.
target	`BIGINT`	Identifier of the target vertex of the current edge *id*. Valid values when *type* = *'e'*.
cost	`FLOAT`	Weight of the edge (*source*, *target*). Valid values when *type* = *'e'*.

Examples

Example Only dead end contraction

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

Example Only linear contraction

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

```
 seq | type | id | contracted_vertices | source | target | cost
-----+------+----+---------------------+--------+--------+------
   1 | e    | -1 | {4}                 |      9 |      3 |    2
   2 | e    | -2 | {8}                 |      5 |      7 |    2
   3 | e    | -3 | {8}                 |      7 |      5 |    2
   4 | e    | -4 | {12}                |     11 |      9 |    2
(4 rows)
```

Indices and tables

- genindex
- search

Introduction

In big 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.

Note: UNDER DISCUSSION: Forbid contraction on a set of edges

Dead end contraction

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

Dead end nodes

The definition of a dead end node is different for a directed and an undirected graph.

In case of a undirected graph, a node is considered a dead end node if

- The number of adjacent vertices is 1.

In case of an directed graph, a node is considered a dead end node if

- There are no outgoing edges and has at least one incoming edge.
- There is one incoming and one outgoing edge with the same identifier.

Examples

- The green node `B` represents a dead end node
- The node `A` is the only node connecting to `B`.
- Node `A` is part of the rest of the graph and has an unlimited number of incoming and outgoing edges.
- 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:

- Node `A` is connected to the rest of the graph by an unlimited number of edges.
- Node `B` is connected to the rest of the graph with one incoming edge.
- Node `B` is the only node connecting to `C`.
- The green node `C` represents a *Dead End* node

After contracting `C`, node `B` is now a *Dead End* node and is contracted:

Node `B` gets contracted

Nodes `B` and `C` belong to node `A`.

Not Dead End nodes

In this graph `B` is not a *dead end* node.

Linear contraction

In the algorithm, linear contraction is represented by 2.

Linear nodes

A node is considered a linear node if satisfies the following:

- The number of adjacent vertices are 2.
- Should have at least one incoming edge and one outgoing edge.

Examples

- The green node `B` represents a linear node
- The nodes `A` and `C` are the only nodes connecting to `B`.
- Node `A` is part of the rest of the graph and has an unlimited number of incoming and outgoing edges.
- Node `C` is part of the rest of the graph and has an unlimited number of incoming and outgoing edges.
- Directed graph

Operation: Linear Contraction

The linear contraction will stop until there are no more linear nodes. For example from the following graph:

- Node A is connected to the rest of the graph by an unlimited number of edges.
- Node B is connected to the rest of the graph with one incoming edge and one outgoing edge.
- Node C is connected to the rest of the graph with one incoming edge and one outgoing edge.
- Node D is connected to the rest of the graph by an unlimited number of edges.
- The green nodes B and C represents *Linear* nodes.

After contracting B, a new edge gets inserted between A and C which is represented by red color.

Node C is *linear node* and gets contracted.

Nodes B and C belong to edge connecting A and D which is represented by red color.

Not Linear nodes

In this graph B is not a *linear* node.

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` .

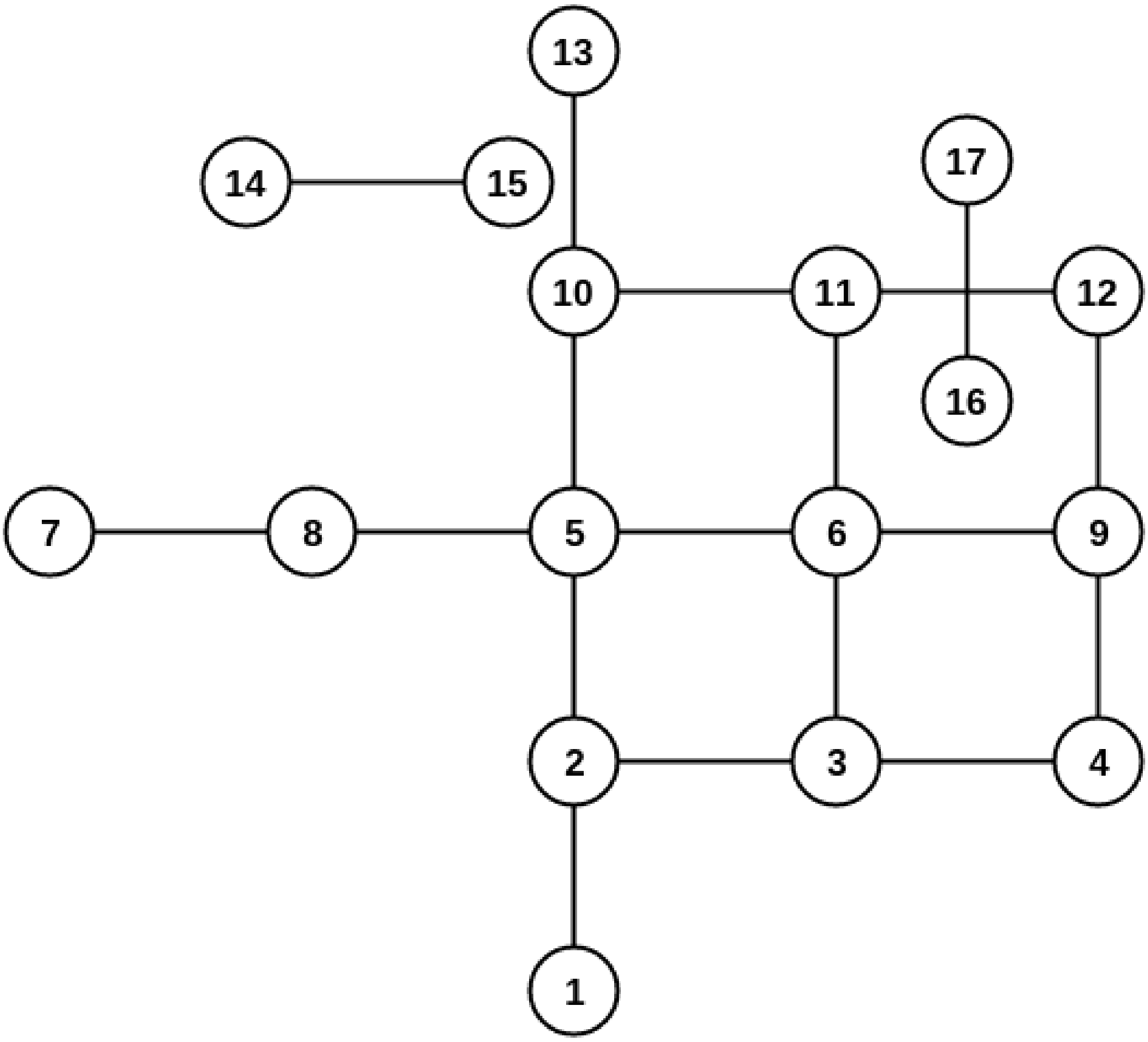
```
<input>
do max_cycles times {
    for (operation in operations_order)
     { do operation }
}
<output>
```

Contracting Sample Data

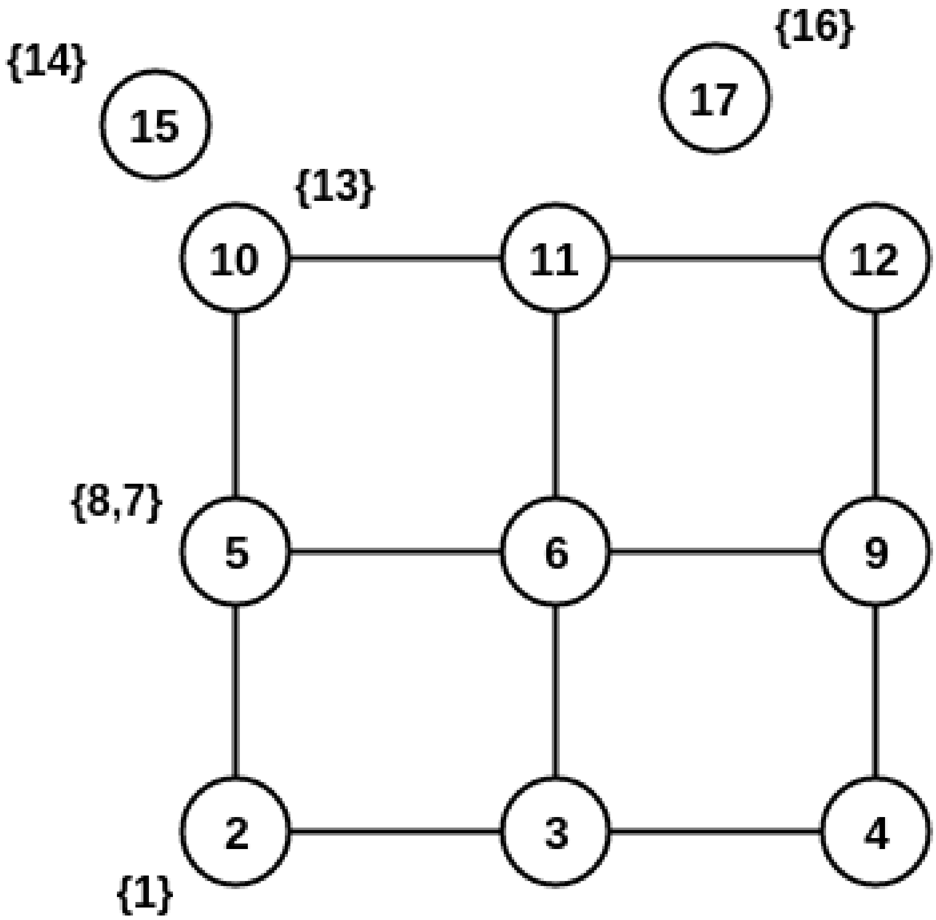
In this section, building and using a contracted graph will be shown by example.

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

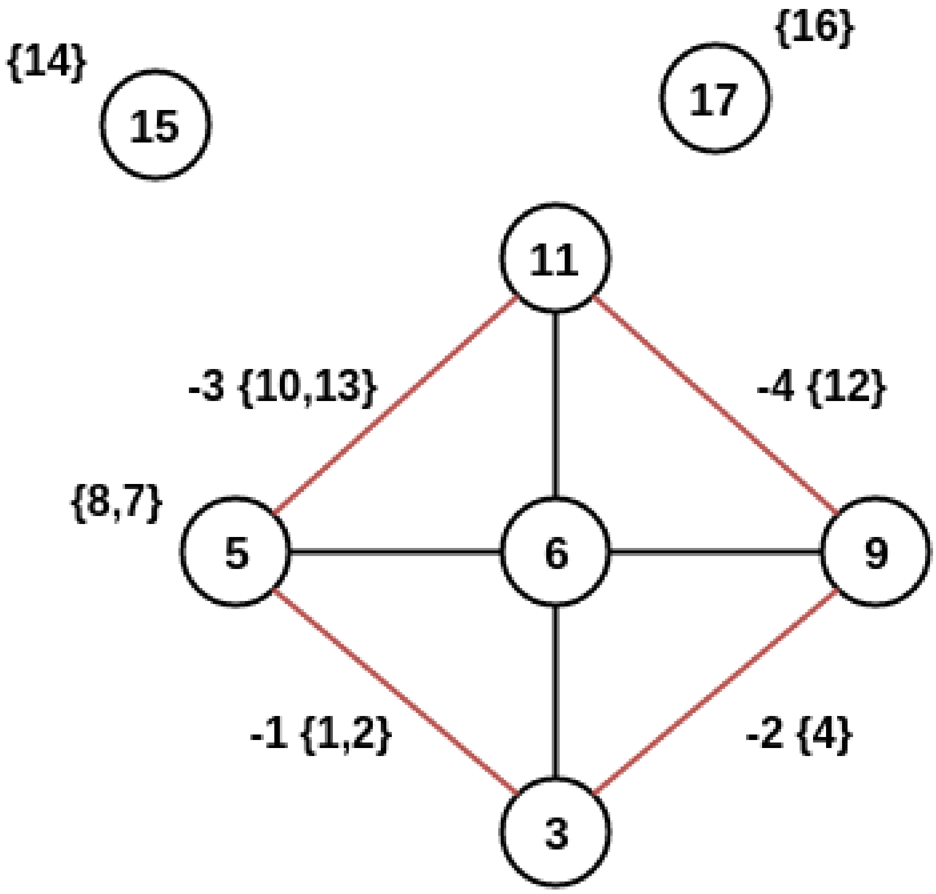
The original graph:



After doing a dead end contraction operation:



Doing a linear contraction operation to the graph above



There are five cases, in this documentation, which arise when calculating the shortest path between a given source and target. In this examples, `pgr_dijkstra` is used.

- **Case 1**: Both source and target belong to the contracted graph.
- **Case 2**: Source belongs to a contracted graph, while target belongs to a edge subgraph.
- **Case 3**: Source belongs to a vertex subgraph, while target belongs to an edge subgraph.
- **Case 4**: Source belongs to a contracted graph, while target belongs to an vertex subgraph.
- **Case 5**: The path contains a new edge added by the contraction algorithm.

Construction of the graph in the database

Original Data

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

Contraction Results

```
SELECT * FROM pgr_contractGraph(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    array[1,2], directed:=true);
 seq | type | id | contracted_vertices | source | target | cost
-----+------+----+---------------------+--------+--------+------
   1 | v    |  5 | {7,8}               |     -1 |     -1 |   -1
   2 | v    | 15 | {14}                |     -1 |     -1 |   -1
```

```
   3 | v    | 17 | {16}                    |      -1 |      -1 |    -1
   4 | e    | -1 | {1,2}                   |       3 |       5 |     2
   5 | e    | -2 | {4}                     |       9 |       3 |     2
   6 | e    | -3 | {10,13}                 |       5 |      11 |     2
   7 | e    | -4 | {12}                    |      11 |       9 |     2
(7 rows)
```

The above results do not represent the contracted graph. They represent the changes done to the graph after applying the contraction algorithm. We can see that vertices like 6 and 11 do not appear in the contraction results because they were not affected by the contraction algorithm.

step 1

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

Column	Description
contracted_vertices	The vertices set belonging to the vertex/edge
is_contracted	On a *vertex* table: when `true` the vertex is contracted, so is not part of the contracted graph.
is_contracted	On an *edge* table: when `true` the edge was generated by the contraction algorithm.

Using the following queries:

```
ALTER TABLE edge_table ADD contracted_vertices BIGINT[];
ALTER TABLE
ALTER TABLE edge_table_vertices_pgr ADD contracted_vertices BIGINT[];
ALTER TABLE
ALTER TABLE edge_table ADD is_contracted BOOLEAN DEFAULT false;
ALTER TABLE
ALTER TABLE edge_table_vertices_pgr ADD is_contracted BOOLEAN DEFAULT false;
ALTER TABLE
SET client_min_messages TO NOTICE;
SET
```

step 2

For simplicity, in this documentation, store the results of the call to pgr_contractGraph in a temporary table

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

step 3

Update the *vertex* and *edge* tables using the results of the call to pgr_contraction

- In *edge_table_vertices_pgr.is_contracted* indicate the vertices that are contracted.

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

- Add to *edge_table_vertices_pgr.contracted_vertices* the contracted vertices belonging to the vertices.

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

- Insert the new edges generated by pgr_contractGraph.

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

step 3.1

Verify visually the updates.

- On the *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)
```

- On the *edge_table*

```
SELECT id, source, target, cost, reverse_cost, contracted_vertices, is_contracted
FROM edge_table
ORDER BY id;
 id | source | target | cost | reverse_cost | contracted_vertices | is_contracted
----+--------+--------+------+--------------+---------------------+---------------
  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 |       9 |       3 |     2 |            -1 | {4}                 | t
 21 |       5 |      11 |     2 |            -1 | {10,13}             | t
 22 |      11 |       9 |     2 |            -1 | {12}                | t
(22 rows)
```

- vertices that belong to the contracted graph are the non contracted vertices

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

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

Inspecting the contracted graph above, vertex 3 and vertex 11 are part of the contracted graph. In the following query:

- vertices_in_graph hold the vertices that belong to the contracted graph.
- when selecting the edges, only edges that have the source and the target in that set are the edges belonging to the contracted graph, that is done in the WHERE clause.

Visually, looking at the original graph, going from 3 to 11: 3 -> 6 -> 11, and in the contracted graph, it is also 3 -> 6 -> 11. The results, on the contracted graph match the results as if it was done on the original graph.

```
SELECT * FROM pgr_dijkstra(
    $$
    WITH
    vertices_in_graph AS (
        SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false)
    SELECT id, source, target, cost, reverse_cost
    FROM edge_table
    WHERE source IN (SELECT * FROM vertices_in_graph)
    AND target IN (SELECT * FROM vertices_in_graph)
    $$,
    3, 11, false);
 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: Source belongs to the contracted graph, while target belongs to a edge subgraph.

Inspecting the contracted graph above, vertex 3 is part of the contracted graph and vertex 1 belongs to the contracted subgr

- expand1 holds the contracted vertices of the edge where vertex 1 belongs. (belongs to edge 19).
- vertices_in_graph hold the vertices that belong to the contracted graph and also the contracted vertices of edge 19.
- when selecting the edges, only edges that have the source and the target in that set are the edges belonging to the contracted graph, that is done in the WHERE clause.

Visually, looking at the original graph, going from 3 to 1: 3 -> 2 -> 1, and in the contracted graph, it is also 3 -> 2 -> 1. The results, on the contracted graph match the results as if it was done on the original graph.

```
SELECT * FROM pgr_dijkstra(
    $$
    WITH
    expand_edges AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table),
    expand1 AS (SELECT contracted_vertices FROM edge_table
        WHERE id IN (SELECT id FROM expand_edges WHERE vertex = 1)),
    vertices_in_graph AS (
        SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false
        UNION
        SELECT unnest(contracted_vertices) FROM expand1)
    SELECT id, source, target, cost, reverse_cost
    FROM edge_table
    WHERE source IN (SELECT * FROM vertices_in_graph)
    AND target IN (SELECT * FROM vertices_in_graph)
    $$,
    3, 1, false);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    2 |    1 |        0
   2 |        2 |    2 |    1 |    1 |        1
   3 |        3 |    1 |   -1 |    0 |        2
(3 rows)
```

case 3: Source belongs to a vertex subgraph, while target belongs to an edge subgraph.

Inspecting the contracted graph above, vertex 7 belongs to the contracted subgraph of vertex 5 and vertex 13 belongs to the contracted subgraph of edge 21. In the following query:

- expand7 holds the contracted vertices of vertex where vertex 7 belongs. (belongs to vertex 5)
- expand13 holds the contracted vertices of edge where vertex 13 belongs. (belongs to edge 21)
- vertices_in_graph hold the vertices that belong to the contracted graph, contracted vertices of vertex 5 and contracted vertices of edge 21.
- when selecting the edges, only edges that have the source and the target in that set are the edges belonging to the contracted graph, that is done in the WHERE clause.

Visually, looking at the original graph, going from 7 to 13: 7 -> 8 -> 5 -> 10 -> 13, and in the contracted graph, it is also 7 -> 8 -> 5 -> 10 -> 13. The results, on the contracted graph match the results as if it was done on the original graph.

```
SELECT * FROM pgr_dijkstra(
    $$
    WITH

    expand_vertices AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table_vertices_
```

```
    expand7 AS (SELECT contracted_vertices FROM edge_table_vertices_pgr
        WHERE id IN (SELECT id FROM expand_vertices WHERE vertex = 7)),

    expand_edges AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table),
    expand13 AS (SELECT contracted_vertices FROM edge_table
        WHERE id IN (SELECT id FROM expand_edges WHERE vertex = 13)),

    vertices_in_graph AS (
        SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false
        UNION
        SELECT unnest(contracted_vertices) FROM expand13
        UNION
        SELECT unnest(contracted_vertices) FROM expand7)

    SELECT id, source, target, cost, reverse_cost
    FROM edge_table
    WHERE source IN (SELECT * FROM vertices_in_graph)
    AND target IN (SELECT * FROM vertices_in_graph)
    $$,
    7, 13, false);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    7 |    6 |    1 |        0
   2 |        2 |    8 |    7 |    1 |        1
   3 |        3 |    5 |   10 |    1 |        2
   4 |        4 |   10 |   14 |    1 |        3
   5 |        5 |   13 |   -1 |    0 |        4
(5 rows)
```

case 4: Source belongs to the contracted graph, while target belongs to an vertex subgraph.

Inspecting the contracted graph above, vertex 3 is part of the contracted graph and vertex 7 belongs to the contracted subgraph of vertex 5. In the following query:

- expand7 holds the contracted vertices of vertex where vertex 7 belongs. (belongs to vertex 5)
- vertices_in_graph hold the vertices that belong to the contracted graph and the contracted vertices of vertex 5.
- when selecting the edges, only edges that have the source and the target in that set are the edges belonging to the contracted graph, that is done in the WHERE clause.

Visually, looking at the original graph, going from 3 to 7: 3 -> 2 -> 5 -> 8 -> 7, but in the contracted graph, it is 3 -> 5 -> 8 -> 7. The results, on the contracted graph do not match the results as if it was done on the original graph. This is because the path contains edge 19 which is added by the contraction algorithm.

```
SELECT * FROM  pgr_dijkstra(
    $$
    WITH
    expand_vertices AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table_vertices_
    expand7 AS (SELECT contracted_vertices FROM edge_table_vertices_pgr
        WHERE id IN (SELECT id FROM expand_vertices WHERE vertex = 7)),
    vertices_in_graph AS (
        SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false
        UNION
        SELECT unnest(contracted_vertices) FROM expand7)
    SELECT id, source, target, cost, reverse_cost
    FROM edge_table
    WHERE source IN (SELECT * FROM vertices_in_graph)
    AND target IN (SELECT * FROM vertices_in_graph)
    $$,
```

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

case 5: The path contains an edge added by the contraction algorithm.

In the previous example we can see that the path from vertex 3 to vertex 7 contains an edge which is added by the contraction algorithm.

```
WITH
first_dijkstra AS (
    SELECT * FROM  pgr_dijkstra(
        $$
        WITH
        expand_vertices AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table_verti
        expand7 AS (SELECT contracted_vertices FROM edge_table_vertices_pgr
            WHERE id IN (SELECT id FROM expand_vertices WHERE vertex = 7)),
        vertices_in_graph AS (
            SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false
            UNION
            SELECT unnest(contracted_vertices) FROM expand7)
        SELECT id, source, target, cost, reverse_cost
        FROM edge_table
        WHERE source IN (SELECT * FROM vertices_in_graph)
        AND target IN (SELECT * FROM vertices_in_graph)
        $$,
        3, 7, false))
SELECT edge, contracted_vertices
    FROM first_dijkstra JOIN edge_table
    ON (edge = id)
    WHERE is_contracted = true;
 edge | contracted_vertices
------+---------------------
   19 | {1,2}
(1 row)
```

Inspecting the contracted graph above, edge 19 should be expanded. In the following query:

- first_dijkstra holds the results of the dijkstra query.
- edges_to_expand holds the edges added by the contraction algorithm and included in the path.
- vertices_in_graph hold the vertices that belong to the contracted graph, vertices of the contracted solution and the contracted vertices of the edges added by the contraction algorithm and included in the contracted solution.
- when selecting the edges, only edges that have the source and the target in that set are the edges belonging to the contracted graph, that is done in the WHERE clause.

Visually, looking at the original graph, going from 3 to 7: 3 -> 2 -> 5 -> 8 -> 7, and in the contracted graph, it is also 3 -> 2 -> 5 -> 8 -> 7. The results, on the contracted graph match the results as if it was done on the original graph.

```
SELECT * FROM pgr_dijkstra($$
    WITH
    -- This returns the results from case 2
```

```
    first_dijkstra AS (
        SELECT * FROM  pgr_dijkstra(
            '
            WITH
            expand_vertices AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table_v
            expand7 AS (SELECT contracted_vertices FROM edge_table_vertices_pgr
                WHERE id IN (SELECT id FROM expand_vertices WHERE vertex = 7)),
            vertices_in_graph AS (
                SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false
                UNION
                SELECT unnest(contracted_vertices) FROM expand7)
            SELECT id, source, target, cost, reverse_cost
            FROM edge_table
            WHERE source IN (SELECT * FROM vertices_in_graph)
            AND target IN (SELECT * FROM vertices_in_graph)
            ',
            3, 7, false)),

    -- edges that need expansion and the vertices to be expanded.
    edges_to_expand AS (
        SELECT edge, contracted_vertices
        FROM first_dijkstra JOIN edge_table
        ON (edge = id)
        WHERE is_contracted = true),

    vertices_in_graph AS (
        -- the nodes of the contracted solution
        SELECT node FROM first_dijkstra
        UNION
        -- the nodes of the expanding sections
        SELECT unnest(contracted_vertices) FROM edges_to_expand)

    SELECT id, source, target, cost, reverse_cost
    FROM edge_table
    WHERE source IN (SELECT * FROM vertices_in_graph)
    AND target IN (SELECT * FROM vertices_in_graph)
    -- not including the expanded edges
    AND id NOT IN (SELECT edge FROM edges_to_expand)
    $$,
    3, 7, false);
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    2 |    1 |        0
   2 |        2 |    2 |    4 |    1 |        1
   3 |        3 |    5 |    7 |    1 |        2
   4 |        4 |    8 |    6 |    1 |        3
   5 |        5 |    7 |   -1 |    0 |        4
(5 rows)
```

See Also

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

Indices and tables

- genindex

- search

6.2.2 Flow - Family of functions

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

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

pgr_maxFlow - Proposed

Synopsis

`pgr_maxFlow` — Calculates the maximum flow in a directed graph from the source(s) to the targets(s) using the Push Relabel algorithm.

[15]

Fig. 6.14: Boost Graph Inside

Availability: 2.4.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Characteristics

- The graph is **directed**.
- 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: $O(V^3)$

Signature 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 Calculates the maximum flow from the *source* to the *target*.

```
pgr_maxFlow(edges_sql, source,  target)
RETURNS BIGINT
```

Example

```
SELECT * FROM pgr_maxFlow(
    'SELECT id,
            source,
            target,
            capacity,
            reverse_capacity
    FROM edge_table'
    , 6, 11
);
 pgr_maxflow
-------------
```

```
         230
(1 row)
```

One to Many Calculates the maximum flow from the *source* to all of the *targets*.

```
pgr_maxFlow(edges_sql, source,  targets)
RETURNS BIGINT
```

Example

```
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 Calculates the maximum flow from all the *sources* to the *target*.

```
pgr_maxFlow(edges_sql, sources,  target)
RETURNS BIGINT
```

Example

```
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 Calculates the maximum flow from all of the *sources* to all of the *targets*.

```
pgr_maxFlow(edges_sql, sources,  targets)
RETURNS BIGINT
```

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for Max-flow like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the Parameters of the Flow Signatures

Column	Type	Default	Description
edges_sql	`TEXT`		The edges SQL query as
source	`BIGINT`		Identifier of the starting
sources	`ARRAY[BIGINT]`		Array of identifiers of th
target	`BIGINT`		Identifier of the ending v
targets	`ARRAY[BIGINT]`		Array of identifiers of th

Description of the return value

Type	Description
`BIGINT`	Maximum flow possible from the source(s) to the target(s)

See Also

- *Flow - Family of functions*
- http://www.boost.org/libs/graph/doc/push_relabel_max_flow.html

- https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm

Indices and tables

- genindex
- search

pgr_pushRelabel - Proposed

Synopsis

`pgr_pushRelabel` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Push Relabel Algorithm.

[16]

Fig. 6.15: Boost Graph Inside

Availability:

- Renamed 2.5.0, Previous name pgr_maxFlowPushRelabel
- New in 2.3.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Characteristics

- 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: $O(V^3)$

Signature Summary

```
pgr_pushRelabel(edges_sql, source,  target) - Proposed
pgr_pushRelabel(edges_sql, sources, target) - Proposed
pgr_pushRelabel(edges_sql, source,  targets) - Proposed
pgr_pushRelabel(edges_sql, sources, targets) - Proposed
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One Calculates the flow on the graph edges that maximizes the flow from the *source* to the *target*.

```
pgr_pushRelabel(edges_sql, source,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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

```
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 Calculates the flow on the graph edges that maximizes the flow from all of the *sources* to the *target*.

```
pgr_pushRelabel(edges_sql, sources,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from all of the *sources* to all of the *targets*.

```
pgr_pushRelabel(edges_sql, sources,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for Max-flow like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the Parameters of the Flow Signatures

Column	Type	Default	Description
edges_sql	`TEXT`		The edges SQL query as
source	`BIGINT`		Identifier of the starting
sources	`ARRAY[BIGINT]`		Array of identifiers of th
target	`BIGINT`		Identifier of the ending v
targets	`ARRAY[BIGINT]`		Array of identifiers of th

Description of the Return Values

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

See Also

- *Flow - Family of functions*, *pgr_boykovKolmogorov*, *pgr_edmondsKarp*
- http://www.boost.org/libs/graph/doc/push_relabel_max_flow.html
- https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm

Indices and tables

- genindex
- search

pgr_edmondsKarp - Proposed

Synopsis

`pgr_edmondsKarp` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Push Relabel Algorithm.

[17]

Fig. 6.16: Boost Graph Inside

Availability:

- Renamed 2.5.0, Previous name pgr_maxFlowEdmondsKarp
- New in 2.3.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Characteristics

- The graph is **directed**.
- Process is done only on edges with positive capacities.
- When the maximum flow is 0 then there is no flow and **EMPTY SET** is returned.
  - There is no flow when a **source** is the same as a **target**.
- Any duplicated value in the source(s) or target(s) are ignored.
- Calculates the flow/residual capacity for each edge. In the output
  - Edges with zero flow are omitted.
- Creates a **super source** and edges to all the source(s), and a **super target** and the edges from all the targets(s).
- The maximum flow through the graph is guaranteed to be the value returned by *pgr_maxFlow* when executed with the same parameters and can be calculated:
  - By aggregation of the outgoing flow from the sources
  - By aggregation of the incoming flow to the targets
- Running time: $O(V * E^2)$

Signature Summary

```
pgr_edmondsKarp(edges_sql, source,  target) - Proposed
pgr_edmondsKarp(edges_sql, sources, target) - Proposed
pgr_edmondsKarp(edges_sql, source,  targets) - Proposed
pgr_edmondsKarp(edges_sql, sources, targets) - Proposed
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One Calculates the flow on the graph edges that maximizes the flow from the *source* to the *target*.

```
pgr_edmondsKarp(edges_sql, source,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from the *source* to all of the *targets*.

```
pgr_edmondsKarp(edges_sql, source,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from all of the *sources* to the *target*.

```
pgr_edmondsKarp(edges_sql, sources,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from all of the *sources* to all of the *targets*.

```
pgr_edmondsKarp(edges_sql, sources,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for Max-flow like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the Parameters of the Flow Signatures

Column	Type	Default	Description
edges_sql	`TEXT`		The edges SQL query as
source	`BIGINT`		Identifier of the starting
sources	`ARRAY[BIGINT]`		Array of identifiers of th
target	`BIGINT`		Identifier of the ending v
targets	`ARRAY[BIGINT]`		Array of identifiers of th

Description of the Return Values

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

See Also

- *Flow - Family of functions*, *pgr_boykovKolmogorov*, *pgr_PushRelabel*
- http://www.boost.org/libs/graph/doc/edmonds_karp_max_flow.html
- https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm

Indices and tables

- genindex
- search

pgr_boykovKolmogorov - Proposed

Synopsis

`pgr_boykovKolmogorov` — Calculates the flow on the graph edges that maximizes the flow from the sources to the targets using Boykov Kolmogorov algorithm.

[18]

Fig. 6.17: Boost Graph Inside

Availability:

- Renamed 2.5.0, Previous name pgr_maxFlowBoykovKolmogorov
- New in 2.3.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Characteristics

- 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

Signature Summary

```
pgr_boykovKolmogorov(edges_sql, source,  target) - Proposed
pgr_boykovKolmogorov(edges_sql, sources, target) - Proposed
pgr_boykovKolmogorov(edges_sql, source,  targets) - Proposed
pgr_boykovKolmogorov(edges_sql, sources, targets) - Proposed
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

One to One Calculates the flow on the graph edges that maximizes the flow from the *source* to the *target*.

```
pgr_boykovKolmogorov(edges_sql, source,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from the *source* to all of the *targets*.

```
pgr_boykovKolmogorov(edges_sql, source,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from all of the *sources* to the *target*.

```
pgr_boykovKolmogorov(edges_sql, sources,  target)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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 Calculates the flow on the graph edges that maximizes the flow from all of the *sources* to all of the *targets*.

```
pgr_boykovKolmogorov(edges_sql, sources,  targets)
RETURNS SET OF (seq, edge, start_vid, end_vid, flow, residual_capacity)
OR EMPTY SET
```

Example

```
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)
```

Description of the Signatures

Description of the edges_sql query for Max-flow like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the Parameters of the Flow Signatures

Column	Type	Default	Description
edges_sql	`TEXT`		The edges SQL query as
source	`BIGINT`		Identifier of the starting
sources	`ARRAY[BIGINT]`		Array of identifiers of th
target	`BIGINT`		Identifier of the ending v
targets	`ARRAY[BIGINT]`		Array of identifiers of th

Description of the Return Values

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

See Also

- *Flow - Family of functions*, *pgr_pushRelabel*, *pgr_EdmondsKarp*
- http://www.boost.org/libs/graph/doc/boykov_kolmogorov_max_flow.html

Indices and tables

- genindex
- search

pgr_maxCardinalityMatch - Proposed

Synopsis

`pgr_maxCardinalityMatch` — Calculates a maximum cardinality matching in a graph.

> **Warning:** Experimental functions
> - They are not officially of the current release.
> - They likely will not be officially be part of the next release:
>   - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
>   - Name might change.
>   - Signature might change.
>   - Functionality might change.
>   - pgTap tests might be missing.
>   - Might need c/c++ coding.
>   - May lack documentation.
>   - Documentation if any might need to be rewritten.
>   - Documentation examples might need to be automatically generated.
>   - Might need a lot of feedback from the comunity.
>   - Might depend on a proposed function of pgRouting
>   - Might depend on a deprecated function of pgRouting

[19]

Fig. 6.18: Boost Graph Inside

Availability:

- Renamed 2.5.0, Previous name pgr_maximumCardinalityMatching
- New in 2.3.0

Characteristics

- A matching or independent edge set in a graph is a set of edges without common vertices.
- A maximum matching is a matching that contains the largest possible number of edges.
  - There may be many maximum matchings.

  - Calculates **one** possible maximum cardinality matching in a graph.
- The graph can be **directed** or **undirected**.
- Running time: $O(E * V * \alpha(E, V))$
  - $\alpha(E, V)$ is the inverse of the Ackermann function[20].

Signature Summary

```
pgr_MaximumCardinalityMatching(edges_sql) - Proposed
pgr_MaximumCardinalityMatching(edges_sql, directed) - Proposed

RETURNS SET OF (seq, edge_id, source, target)
    OR EMPTY SET
```

Minimal Use

```
pgr_MaximumCardinalityMatching(edges_sql)
RETURNS SET OF (seq, edge_id, source, target) OR EMPTY SET
```

The minimal use calculates one possible maximum cardinality matching on a **directed** graph.

Example

```
SELECT * FROM pgr_maxCardinalityMatch(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table'
);
 seq | edge | source | target
-----+------+--------+--------
   1 |    1 |      1 |      2
   2 |    3 |      4 |      3
   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)
```

Complete signature

```
pgr_MaximumCardinalityMatching(edges_sql, directed)
RETURNS SET OF (seq, edge_id, source, target) OR EMPTY SET
```

The complete signature calculates one possible maximum cardinality matching.

Example

```
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
```

[20] https://en.wikipedia.org/wiki/Ackermann_function

```
   7 |    17 |      14 |      15
   8 |    18 |      16 |      17
(8 rows)
```

Description of the Signatures

Description of the SQL query

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

Column	Type	Description
id	`ANY-INTEGER`	Identifier of the edge.
source	`ANY-INTEGER`	Identifier of the first end point vertex of the edge.
target	`ANY-INTEGER`	Identifier of the second end point vertex of the edge.
going	`ANY-NUMERIC`	A positive value represents the existence of the edge (source, target).
coming	`ANY-NUMERIC`	A positive value represents the existence of the edge (target, source).

Where:

- **ANY-INTEGER** SMALLINT, INTEGER, BIGINT
- **ANY-NUMERIC** SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION

Description of the parameters of the signatures

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
directed	`BOOLEAN`	(optional) Determines the type of the graph. Defaul

Description of the Result

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
edge	`BIGINT`	Identifier of the edge in the original query(edges_sql).
source	`BIGINT`	Identifier of the first end point of the edge.
target	`BIGINT`	Identifier of the second end point of the edge.

See Also

- *Flow - Family of functions*
- http://www.boost.org/libs/graph/doc/maximum_matching.html
- https://en.wikipedia.org/wiki/Matching_%28graph_theory%29
- https://en.wikipedia.org/wiki/Ackermann_function

Indices and tables

- genindex
- search

pgr_edgeDisjointPaths - Proposed

Name

`pgr_edgeDisjointPaths` — Calculates edge disjoint paths between two groups of vertices.

[21]

Fig. 6.19: Boost Graph Inside

Availability: 2.3.0

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

Calculates the edge disjoint paths between two groups of vertices. Utilizes underlying maximum flow algorithms to calculate the paths.

Characteristics:

The main characterics are:

- Calculates the edge disjoint paths between any two groups of vertices.
- Returns EMPTY SET when source and destination are the same, or cannot be reached.
- The graph can be directed or undirected.
- One to many, many to one, many to many versions are also supported.
- Uses *pgr_boykovKolmogorov - Proposed* to calculate the paths.

Signature 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
```

Signatures

Minimal use

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

The minimal use is for a **directed** graph from one `start_vid` to one `end_vid`.

Example

```
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

This signature finds the set of dijoint paths from one `start_vid` to one `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

```
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

```
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

This signature finds the sset of disjoint paths from the `start_vid` to each one of the `end_vid` in `end_vids`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.
- The result is equivalent to the union of the results of the one to one *pgr_edgeDisjointPaths*.
- The extra `end_vid` in the result is used to distinguish to which path it belongs.

```
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

```
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

This signature finds the set of disjoint paths from each one of the `start_vid` in `start_vids` to the `end_vid`:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.
- The result is equivalent to the union of the results of the one to one *pgr_edgeDisjointPaths*.
- The extra `start_vid` in the result is used to distinguish to which path it belongs.

```
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

```
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

This signature finds the set of disjoint paths from each one of the `start_vid` in `start_vids` to each one of the `end_vid`

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.
- The result is equivalent to the union of the results of the one to one *pgr_edgeDisjointPaths*.
- The extra `start_vid` and `end_vid` in the result is used to distinguish to which path it belongs.

```
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

```
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)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Default
sql	`TEXT`	
start_vid	`BIGINT`	
start_vids	`ARRAY[BIGINT]`	
end_vid	`BIGINT`	
end_vids	`ARRAY[BIGINT]`	
directed	`BOOLEAN`	`true`

Description of the return values for a path Returns set of `(seq, path_seq [, start_vid] [, end_vid], node, edge, cost, agg_cost)`

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

See Also

- *Flow - Family of functions*

Indices and tables

- genindex
- search

Flow Functions General Information

Characteristics

- 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.

Problem definition

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 the for source which only has outgoing flow, and the destination(sink) which only has incoming flow.

Maximum flow algorithms calculate the maximum flow through the graph and the flow of each edge.

The maximum flow through the graph is guaranteed to be the same with all implementations, but the actual flow through each edge may vary. Given the following query:

pgr_maxFlow $(edges\_sql, source\_vertex, sink\_vertex)$

where $edges\_sql = \{(id_i, source_i, target_i, capacity_i, reverse\_capacity_i)\}$

Graph definition

The weighted directed graph, $G(V, E)$, is defined as:

- the set of vertices V
  - $source\_vertex \cup sink\_vertex \bigcup source_i \bigcup target_i$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, capacity_i) \text{ when } capacity > 0\} & \text{if } reverse\_capacity = \\ \\ \{(source_i, target_i, capacity_i) \text{ when } capacity > 0\} & \\ \cup\{(target_i, source_i, reverse\_capacity_i) \text{ when } reverse\_capacity_i > 0)\} & \text{if } reverse\_capacity \neq \end{cases}$

Maximum flow problem

Given:

- $G(V, E)$
- $source\_vertex \in V$ the source vertex
- $sink\_vertex \in V$ the sink vertex

Then:

$$pgr\_maxFlow(edges\_sql, source, sink) = \boldsymbol{\Phi}$$

$$\boldsymbol{\Phi} = (id_i, edge\_id_i, source_i, target_i, flow_i, residual\_capacity_i)$$

Where:

$\boldsymbol{\Phi}$ is a subset of the original edges with their residual capacity and flow. The maximum flow through the graph can be obtained by aggregating on the source or sink and summing the flow from/to it. In particular:

- $id_i = i$
- $edge\_id = id_i$ in edges_sql
- $residual\_capacity_i = capacity_i - flow_i$

See Also

- https://en.wikipedia.org/wiki/Maximum_flow_problem

Indices and tables

- genindex
- search

6.2.3 pgr_labelGraph - Experimental

Name

`pgr_labelGraph` — Locates and labels sub-networks within a network which are not topologically connected.

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

Must be run after `pgr_createTopology()`. No use of `geometry` column. Only `id`, `source` and `target` columns are required.

The function returns:

- `OK` when a column with provided name has been generated and populated successfully. All connected edges will have unique similar integer values. In case of `rows_where` condition, non participating rows will have -1 integer values.
- `FAIL` when the processing cannot be finished due to some error. Notice will be thrown accordingly.
- `rows_where condition generated 0 rows` when passed SQL condition has not been fulfilled by any row.

```
varchar pgr_labelGraph(text, text, text, text, text, text)
```

Description

A network behind any routing query may consist of sub-networks completely isolated from each other. Possible reasons could be:

- An island with no bridge connecting to the mainland.
- An edge or mesh of edges failed to connect to other networks because of human negligence during data generation.
- The data is not properly noded.
- Topology creation failed to succeed.

pgr_labelGraph() will create an integer column (with the name provided by the user) and will assign same integer values to all those edges in the network which are connected topologically. Thus better analysis regarding network structure is possible. In case of `rows_where` condition, non participating rows will have -1 integer values.

Prerequisites: Must run `pgr_createTopology()` in order to generate `source` and `target` columns. Primary key column `id` should also be there in the network table.

Function accepts the following parameters:

edge_table `text` Network table name, with optional schema name.

id `text` Primary key column name of the network table. Default is `id`.

source `text` Source column name generated after `pgr_createTopology()`. Default is `source`.

target `text` Target column name generated after `pgr_createTopology()`. Default is `target`.

subgraph `text` Column name which will hold the integer labels for each sub-graph. Default is `subgraph`.

rows_where `text` The SQL where condition. Default is `true`, means the processing will be done on the whole table.

Example Usage

The sample data, has 3 subgraphs.

```
SET client_min_messages TO WARNING;
SET
SELECT pgr_labelGraph('edge_table', 'id', 'source', 'target', 'subgraph');
 pgr_labelgraph
----------------
 OK
(1 row)

SELECT DISTINCT subgraph FROM edge_table ORDER BY subgraph;
 subgraph
----------
        1
        2
        3
(3 rows)
```

See Also

- pgr_createTopology[22] to create the topology of a table based on its geometry and tolerance value.

Indices and tables

- genindex
- search

6.2.4 Components - Family of functions

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

- *pgr_connectedComponents - Experimental* - Return the connected components of an undirected graph.
- *pgr_strongComponents - Experimental* - Return the strongly connected components of a directed graph.
- *pgr_biconnectedComponents - Experimental* - Return the biconnected components of an undirected graph.
- *pgr_articulationPoints - Experimental* - Return the articulation points of an undirected graph.
- *pgr_bridges - Experimental* - Return the bridges of an undirected graph.

pgr_connectedComponents - Experimental

`pgr_connectedComponents` — Return the connected components of an undirected graph using a DFS-based approach. In particular, the algorithm implemented by Boost.Graph.

[23]

Fig. 6.20: Boost Graph Inside

[22] https://github.com/Zia-/pgrouting/blob/develop/src/common/sql/pgrouting_topology.sql

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

A connected component of an undirected graph is a set of vertices that are all reachable from each other. This implementation can only be used with an undirected graph.

Characteristics

The main Characteristics are:

- Components are described by vertices
- The returned values are ordered:
  - *component* ascending
  - *node* ascending
- Running time: $O(V + E)$

Signatures

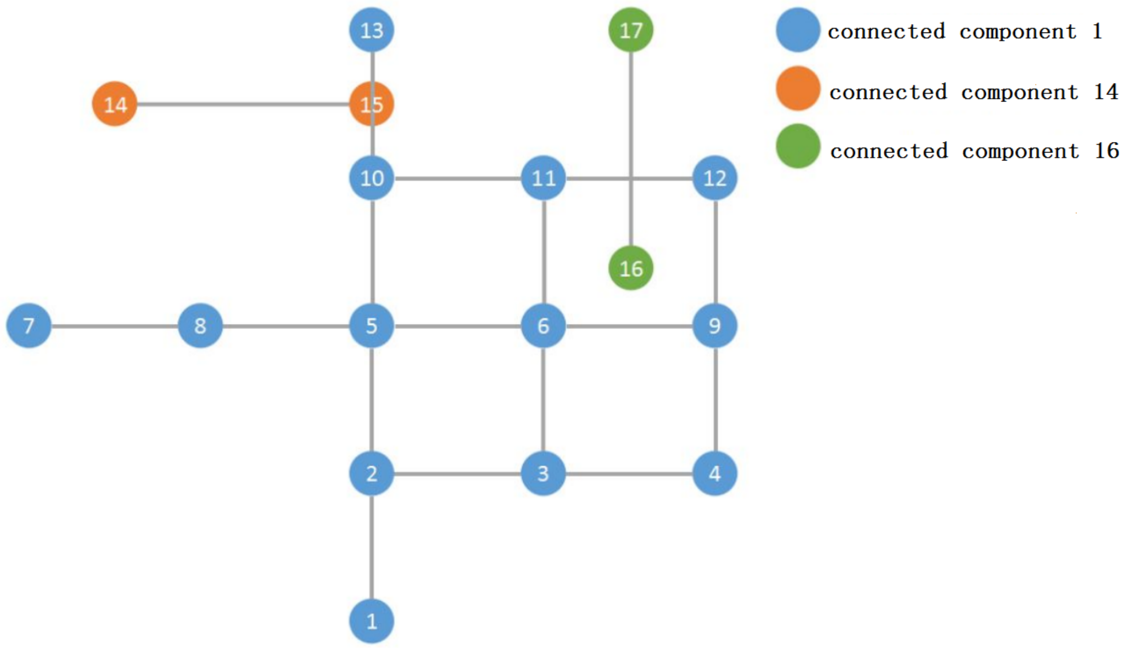
```
pgr_connectedComponents(edges_sql)

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

The signature is for a **undirected** graph.

Example

```
SELECT * FROM pgr_connectedComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | component | n_seq | node
-----+-----------+-------+------
   1 |         1 |     1 |    1
   2 |         1 |     2 |    2
   3 |         1 |     3 |    3
   4 |         1 |     4 |    4
   5 |         1 |     5 |    5
   6 |         1 |     6 |    6
   7 |         1 |     7 |    7
   8 |         1 |     8 |    8
```

```
  9 |          1 |      9 |     9
 10 |          1 |     10 |    10
 11 |          1 |     11 |    11
 12 |          1 |     12 |    12
 13 |          1 |     13 |    13
 14 |         14 |      1 |    14
 15 |         14 |      2 |    15
 16 |         16 |      1 |    16
 17 |         16 |      2 |    17
(17 rows)
```



Description of the Signatures

Description of the edges_sql query for components functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for connected components and strongly connected components Returns set of `(seq, component, n_seq, node)`

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

See Also

- http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29
- The queries use the Sample Data network.

Indices and tables

- genindex
- search

pgr_strongComponents - Experimental

`pgr_strongComponents` — Return the strongly connected components of a directed graph using Tarjan's algorithm based on DFS. In particular, the algorithm implemented by Boost.Graph.

[24]

Fig. 6.21: Boost Graph Inside

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

A strongly connected component of a directed graph is a set of vertices that are all reachable from each other. This implementation can only be used with a directed graph.

Characteristics

The main Characteristics are:

- Components are described by vertices
- The returned values are ordered:
  - *component* ascending
  - *node* ascending
- Running time: $O(V + E)$

Signatures

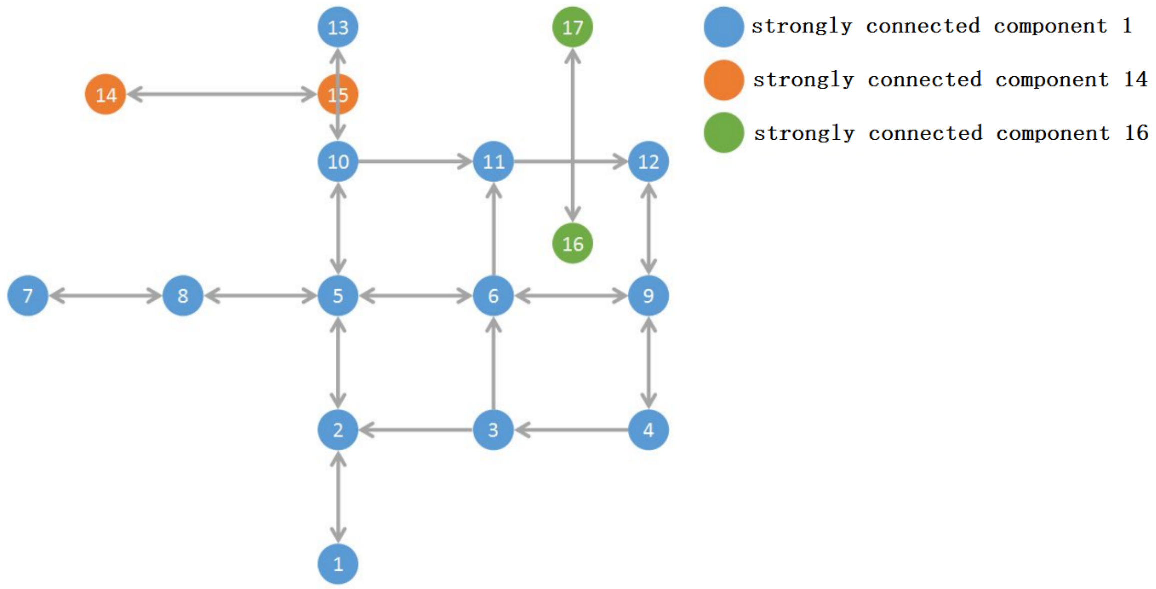
```
pgr_strongComponents(edges_sql)

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

The signature is for a **directed** graph.

Example

```
SELECT * FROM pgr_strongComponents(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | component | n_seq | node
```

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



Description of the Signatures

Description of the edges_sql query for components functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for connected components and strongly connected components Returns set of `(seq, component, n_seq, node)`

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
component	`BIGINT`	Component identifier. It is equal to the minimum node identifier in the component.
n_seq	`INT`	It is a sequential value starting from **1** in a component.
node	`BIGINT`	Identifier of the vertex.

See Also

- http://en.wikipedia.org/wiki/Strongly_connected_component
- The queries use the Sample Data network.

Indices and tables

- genindex
- search

pgr_biconnectedComponents - Experimental

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

[25]

Fig. 6.22: Boost Graph Inside

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

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. So, the output only has edge version.

This implementation can only be used with an undirected graph.

Characteristics

The main Characteristics are:

- Components are described by edges
- The returned values are ordered:
  - *component* ascending
  - *edge* ascending
- Running time: $O(V + E)$

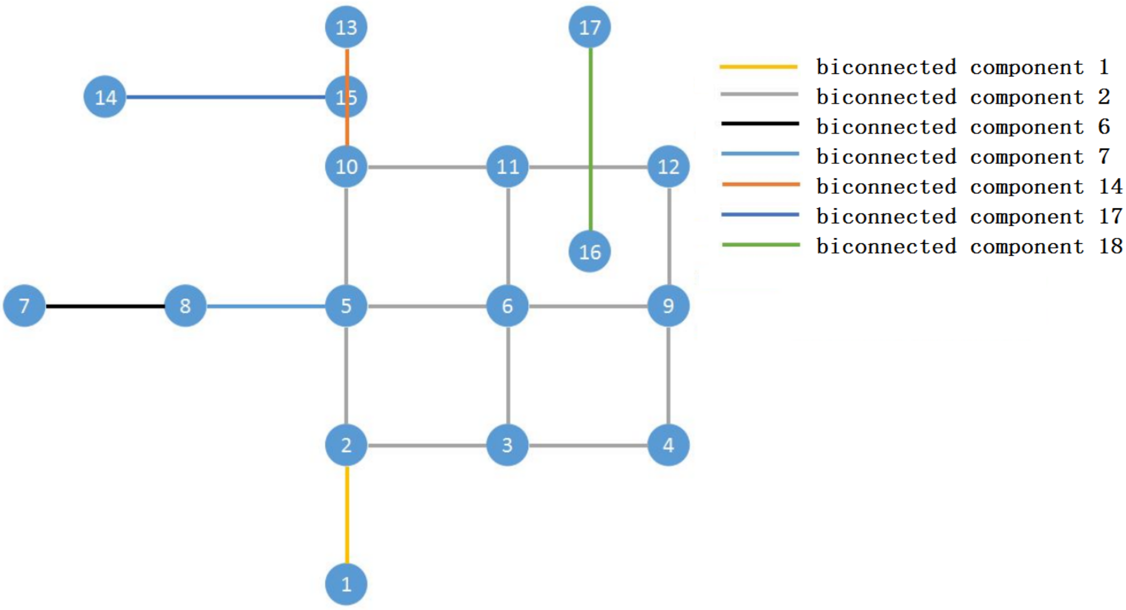
Signatures

```
pgr_biconnectedComponents(edges_sql)

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

The signature is for a **undirected** graph.

Example

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



Description of the Signatures

Description of the edges_sql query for components functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for biconnected components, connected components (edge version) and strongly connected components Returns set of `(seq, component, n_seq, edge)`

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

See Also

- http://en.wikipedia.org/wiki/Biconnected_component
- The queries use the Sample Data network.

Indices and tables

- genindex
- search

pgr_articulationPoints - Experimental

`pgr_articulationPoints` - Return the articulation points of an undirected graph. In particular, the algorithm implemented by Boost.Graph.

[26]

Fig. 6.23: Boost Graph Inside

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

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.

Characteristics

The main Characteristics are:

- The returned values are ordered:
  - *node* ascending
- Running time: $O(V + E)$

Signatures

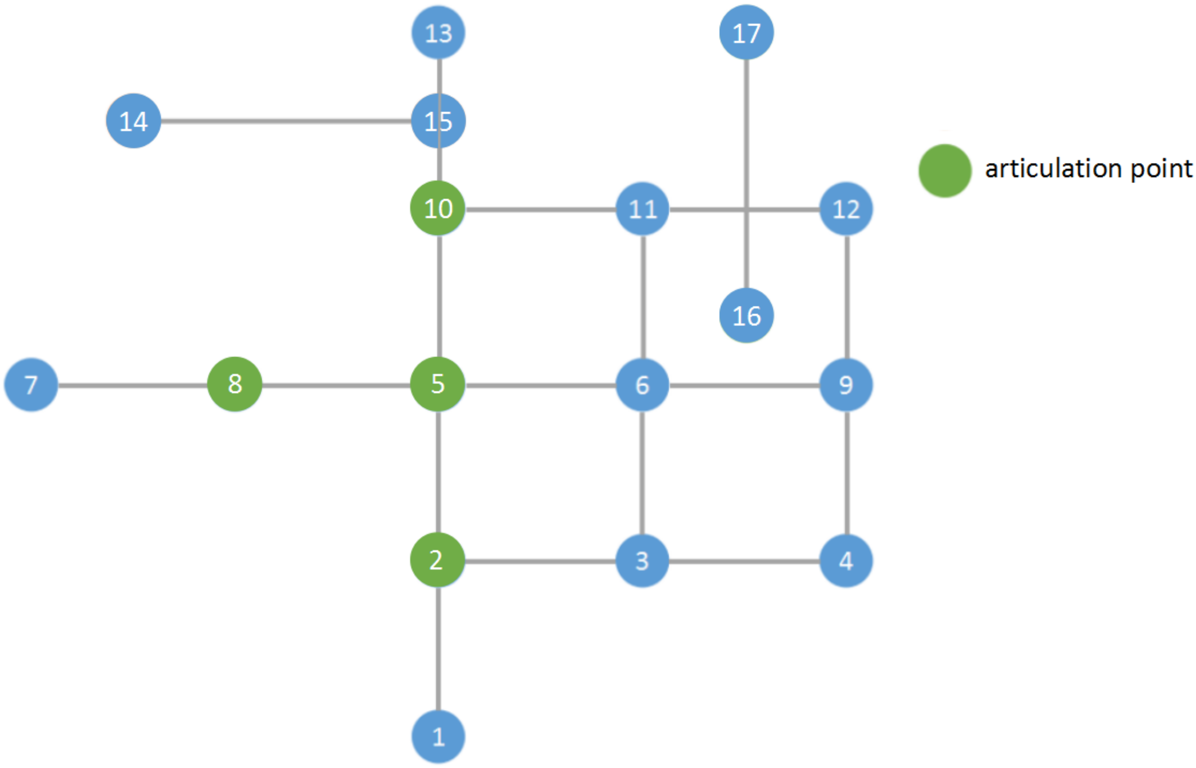
```
pgr_articulationPoints(edges_sql)

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

The signature is for a **undirected** graph.

Example

```
SELECT * FROM pgr_articulationPoints(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | node
-----+------
   1 |    2
```

```
   2 |     5
   3 |     8
   4 |    10
(4 rows)
```



Description of the Signatures

Description of the edges_sql query for components functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for articulation points Returns set of `(seq, node)`

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
node	`BIGINT`	Identifier of the vertex.

See Also

- http://en.wikipedia.org/wiki/Biconnected_component
- The queries use the Sample Data network.

Indices and tables

- genindex
- search

pgr_bridges - Experimental

`pgr_bridges` - Return the bridges of an undirected graph.

[27]

Fig. 6.24: Boost Graph Inside

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

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.

Characteristics

The main Characteristics are:

- The returned values are ordered:
  - *edge* ascending
- Running time: $O(E * (V + E))$

Signatures

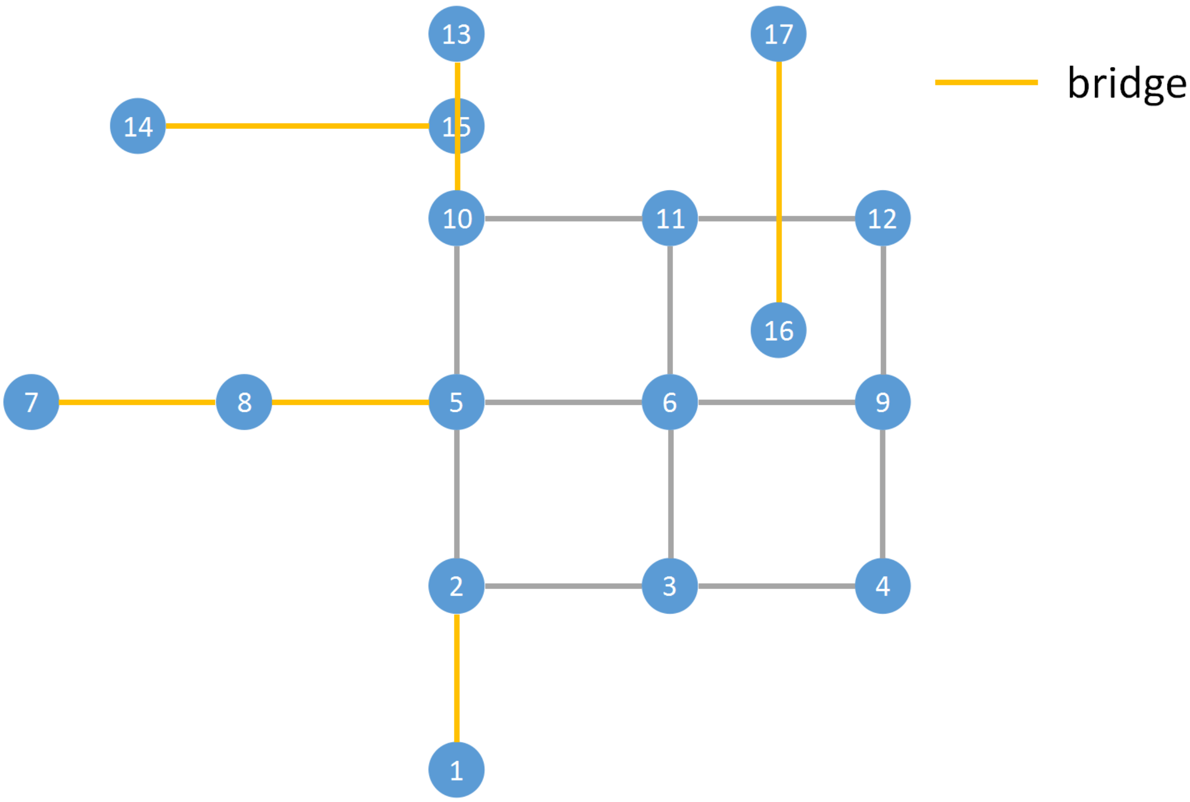
```
pgr_bridges(edges_sql)

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

The signature is for a **undirected** graph.

Example

```
SELECT * FROM pgr_bridges(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | edge
-----+------
   1 |    1
   2 |    6
```

```
   3 |     7
   4 |    14
   5 |    17
   6 |    18
(6 rows)
```



Description of the Signatures

Description of the edges_sql query for components functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for bridges Returns set of `(seq, node)`

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
edge	`BIGINT`	Identifier of the edge.

See Also

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

Indices and tables

- genindex
- search

The problem definition

Connected components

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

Notice: This problem defines on an undirected graph.

Given the following query:

pgr_connectedComponentsV(sql)

where $sql = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $source = \bigcup source_i$,
- $target = \bigcup target_i$,

The graphs are defined as follows:

The weighted undirected graph, $G(V, E)$, is definied by:

- the set of vertices V
  - $V = source \cup target$
- the set of edges E
  - $$E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ & \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \\ \cup\{(source_i, target_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \text{if } reverse\_cost \neq \end{cases}$$

Given:

- $G(V, E)$

Then:

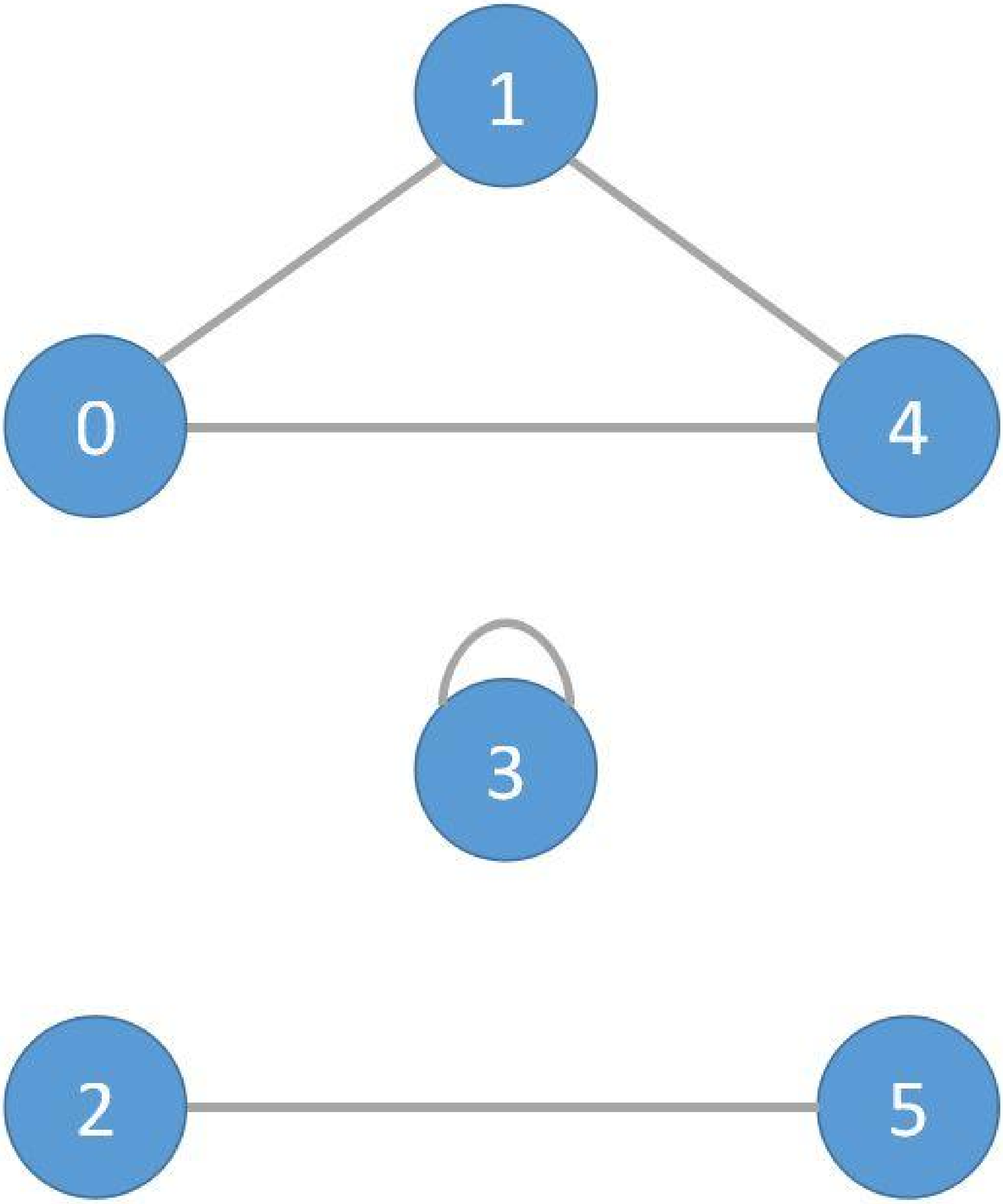
$\boldsymbol{\pi} = \{(component_i, n\_seq_i, node_i)\}$

where:

- $component_i = \min\{node_j | node_j \in component_i\}$
- n_seq_i is a sequential value starting from **1** in a component.
- $node_i \in component_i$
- The returned values are ordered:
  - *component* ascending
  - *node* ascending

Example:

- The first component is composed of nodes `0`, `1` and `4`.
- The second component is composed of node `3`.
- The third component is composed of nodes `2` and `5`.



Strongly connected components

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

Notice: This problem defines on a directed graph.

Given the following query:

pgr_strongComponentsV(sql)

where $sql = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $source = \bigcup source_i$,

- $target = \bigcup target_i$,

The graphs are defined as follows:

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

- the set of vertices V
  - $V = source \cup target \cup start_{vid} \cup end_{vid}$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ & \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} & \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \text{if } reverse\_cost \neq \end{cases}$

Given:

- $G(V, E)$

Then:

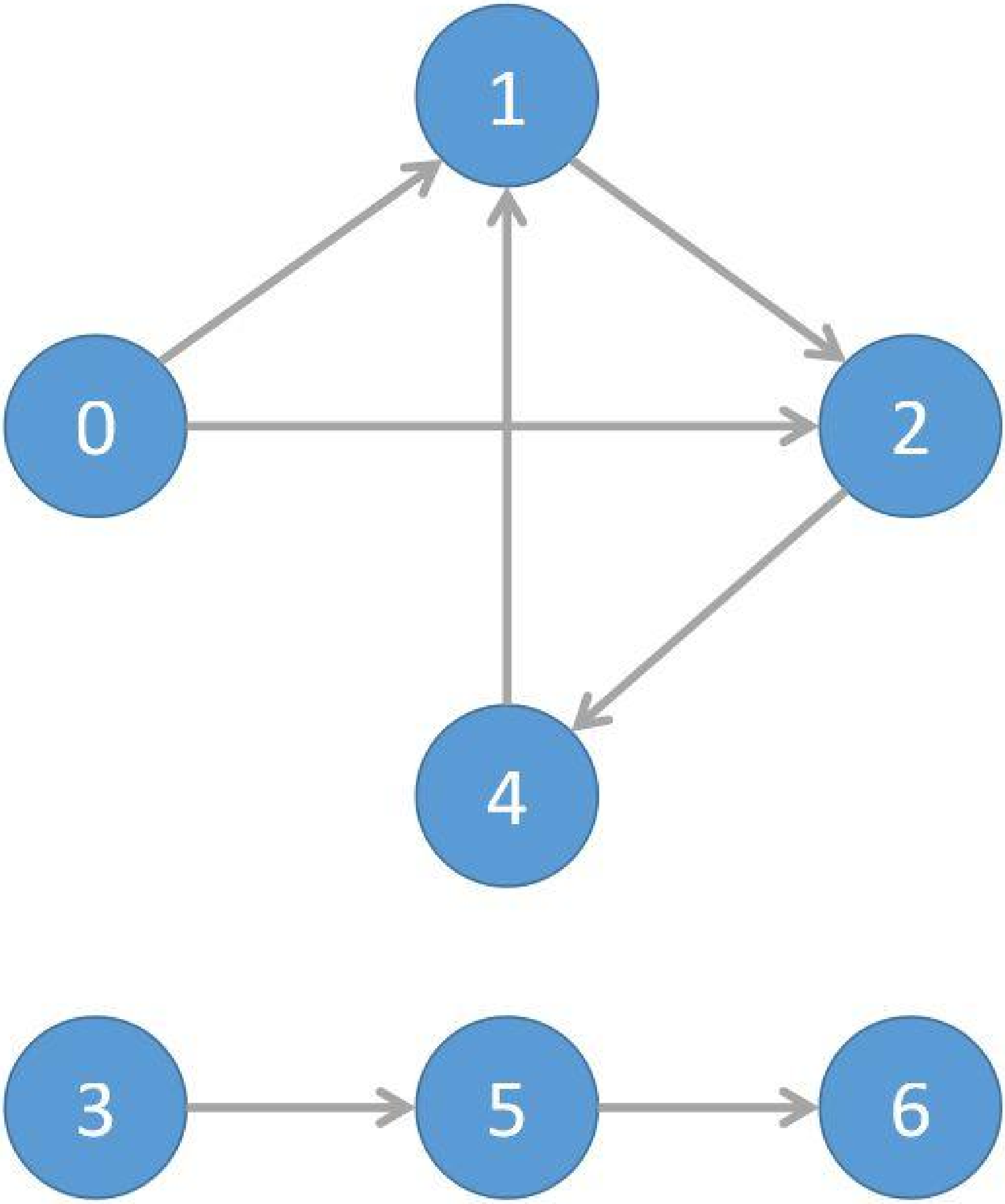
$\boldsymbol{\pi} = \{(component_i, n\_seq_i, node_i)\}$

where:

- $component_i = \min node_j | node_j \in component_i$
- n_seq_i is a sequential value starting from **1** in a component.
- $node_i \in component_i$
- The returned values are ordered:
  - *component* ascending
  - *node* ascending

Example:

- The first component is composed of nodes `1`, `2` and `4`.
- The second component is composed of node `0`.
- The third component is composed of node `3`.
- The fourth component is composed of node `5`.
- The fifth component is composed of node `6`.



Biconnected components

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. So, the output only has edge version.

Notice: This problem defines on an undirected graph.

Given the following query:

pgr_biconnectedComponents(sql)

where $sql = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $source = \bigcup source_i$,
- $target = \bigcup target_i$,

The graphs are defined as follows:

The weighted undirected graph, $G(V, E)$, is definied by:

- the set of vertices V
  - $V = source \cup target$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} \\ \cup\{(source_i, target_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \text{if } reverse\_cost \neq \end{cases}$

Given:

- $G(V, E)$

Then:

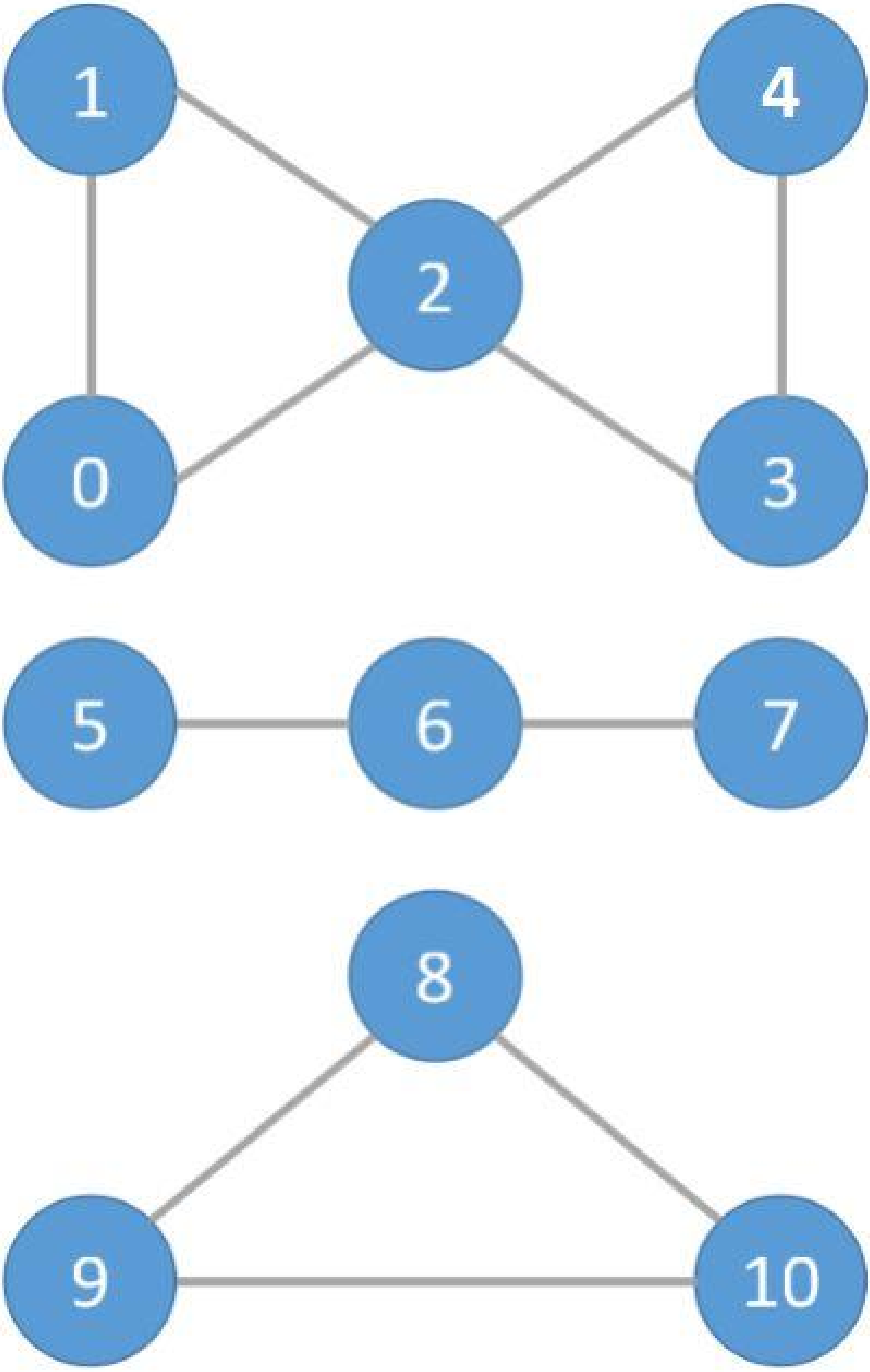
$\boldsymbol{\pi} = \{(component_i, n\_seq_i, node_i)\}$

where:

- $component_i = \min node_j | node_j \in component_i$
- n_seq_i is a sequential value starting from **1** in a component.
- $edge_i \in component_i$
- The returned values are ordered:
  - *component* ascending
  - *edge* ascending

Example:

- The first component is composed of edges `1 - 2`, `0 - 1` and `0 - 2`.
- The second component is composed of edges `2 - 4`, `2 - 3` and `3 - 4`.
- The third component is composed of edge `5 - 6`.
- The fourth component is composed of edge `6 - 7`.
- The fifth component is composed of edges `8 - 9`, `9 - 10` and `8 - 10`.


1
4
2
0
3
5
6
7
8
9
10

Articulation Points

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.

Notice: This problem defines on an undirected graph.

Given the following query:

pgr_articulationPoints(sql)

where $sql = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $source = \bigcup source_i$,
- $target = \bigcup target_i$,

The graphs are defined as follows:

The weighted undirected graph, $G(V, E)$, is definied by:

- the set of vertices V
  - $V = source \cup target$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} \\ \cup\{(source_i, target_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \text{if } reverse\_cost \neq \end{cases}$

Given:

- $G(V, E)$

Then:

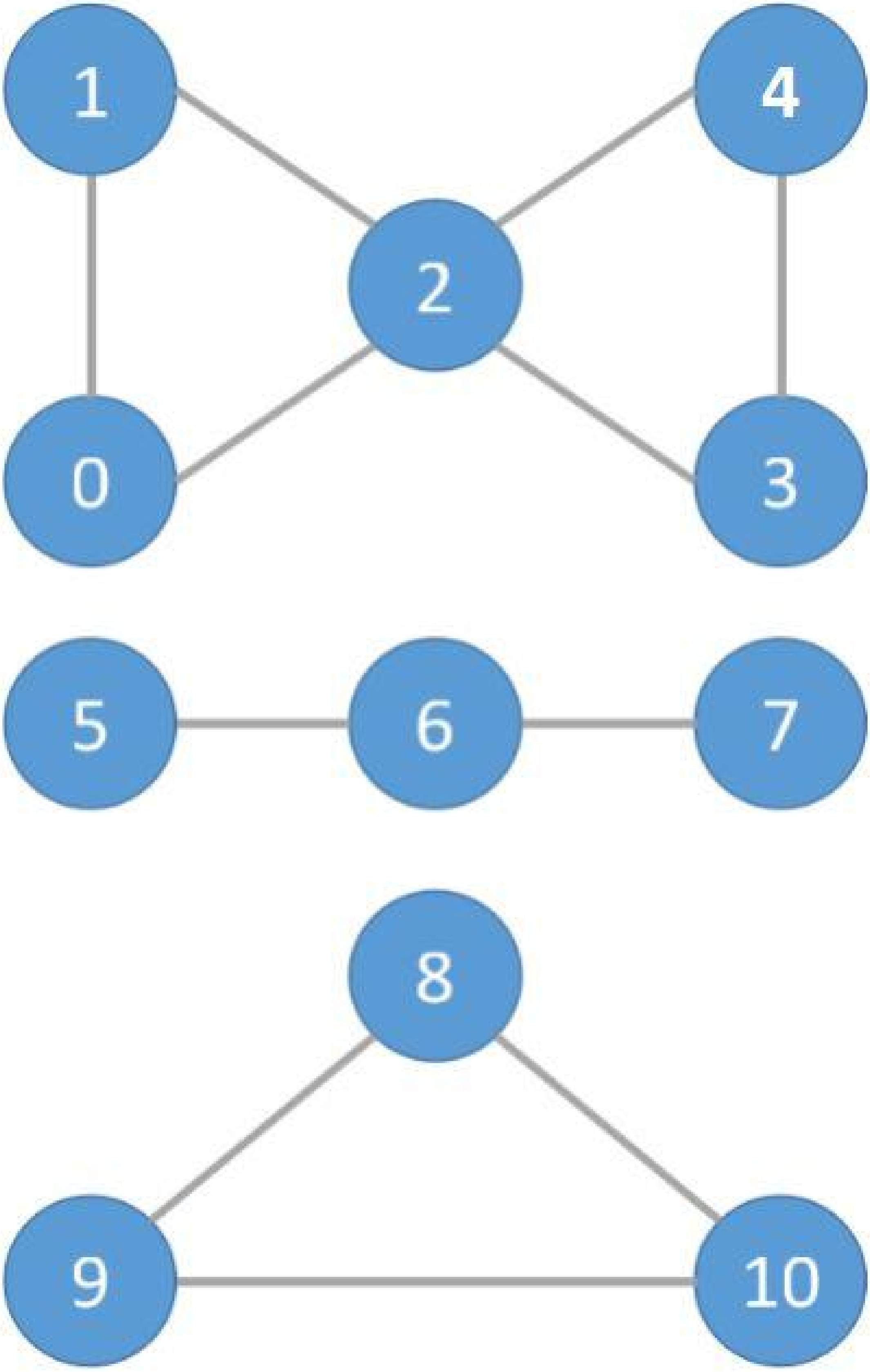
$\boldsymbol{\pi} = \{node_i\}$

where:

- $node_i$ is an articulation point.
- The returned values are ordered:
  - $node$ ascending

Example:

- Articulation points are nodes `2` and `6`.


1
4
2
0
3
5
6
7
8
9
10

Bridges

A bridge is an edge of an undirected graph whose deletion increases its number of connected components.

Notice: This problem defines on an undirected graph.

Given the following query:

pgr_bridges(sql)

where $sql = \{(id_i, source_i, target_i, cost_i, reverse\_cost_i)\}$

and

- $source = \bigcup source_i$,
- $target = \bigcup target_i$,

The graphs are defined as follows:

The weighted undirected graph, $G(V, E)$, is definied by:

- the set of vertices V
  - $V = source \cup target$
- the set of edges E
  - $E = \begin{cases} \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} & \text{if } reverse\_cost = \\ \\ \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, cost_i) \text{ when } cost >= 0\} \\ \cup\{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} \\ \cup\{(source_i, target_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} & \text{if } reverse\_cost \neq \end{cases}$

Given:

- $G(V, E)$

Then:

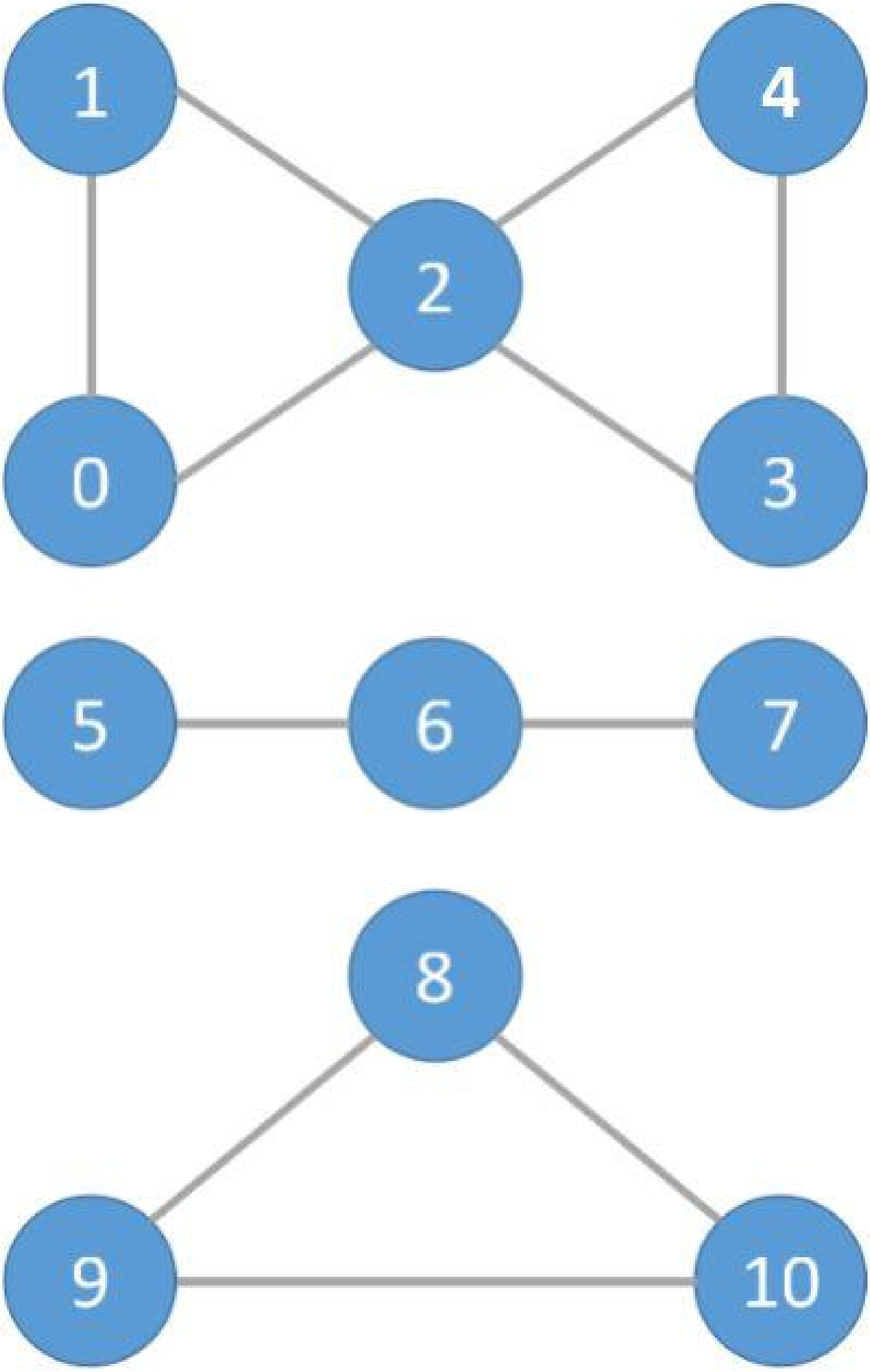
$\boldsymbol{\pi} = \{edge_i\}$

where:

- $edge_i$ is an edge.
- The returned values are ordered:
  - $edge$ ascending

Example:

- Bridges are edges `5 <--> 6` and `6 <--> 7`.


1
4
2
0
3
5
6
7
8
9
10

Description of the edges_sql query for components functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

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

Description of the return values for connected components and strongly connected components

Returns set of `(seq, component, n_seq, node)`

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
component	`BIGINT`	Component identifier. It is equal to the minimum node identifier in the component.
n_seq	`INT`	It is a sequential value starting from **1** in a component.
node	`BIGINT`	Identifier of the vertex.

Description of the return values for biconnected components, connected components (edge version) and strongly connected components

Returns set of `(seq, component, n_seq, edge)`

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

Description of the return values for articulation points

Returns set of (seq, node)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
node	BIGINT	Identifier of the vertex.

Description of the return values for bridges

Returns set of (seq, node)

Column	Type	Description
seq	INT	Sequential value starting from **1**.
edge	BIGINT	Identifier of the edge.

See Also

Indices and tables

- genindex
- search

6.2.5 pgr_gsoc_vrppdtw - Experimental

Name

pgr_gsoc_vrppdtw — Returns a solution for *Pick and Delivery* with *time windows* Vehicle Routing Problem

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Signature Summary

```
pgr_gsoc_vrppdtw(sql, vehicle_num, capacity)
RETURNS SET OF pgr_costResult[]:
```

Signatures

Complete signature

```
pgr_gsoc_vrppdtw(sql, vehicle_num, capacity)
Returns set of pgr_costResult[]:
```

Example: Show the id1

```
SELECT DISTINCT(id1) FROM pgr_gsoc_vrppdtw(
    'SELECT * FROM customer ORDER BY id', 25, 200)
ORDER BY id1;
 id1
-----
   1
   2
   3
   4
   5
   6
   7
   8
   9
  10
(10 rows)
```

Description of the Signatures

Description of the sql query

Column	Type	Description
id	ANY-INTEGER	Identifier of the customer. • A value of 0 identifies the starting location
x	ANY-NUMERICAL	X coordinate of the location.
y	ANY-NUMERICAL	Y coordinate of the location.
demand	ANY-NUMERICAL	How much is added / removed from the vehicle. • Negative value is a delivery, • Positive value is a pickup,
openTime	ANY-NUMERICAL	The time relative to 0, when the customer opens.
closeTime	ANY-NUMERICAL	The time relative to 0, when the customer closes.
serviceTime	ANY-NUMERICAL	The duration of the loading / unloading.
pIndex	ANY-INTEGER	Value used when the current customer is a Delivery to find the corresponding Pickup
dIndex	ANY-INTEGER	Value used when the current customer is a Pickup to find the corresponding Delivery

Description of the parameters of the signatures

Column	Type	Description
sql	`TEXT`	SQL query as described above.
vehicle_num	`INTEGER`	Maximum number of vehicles in the result. (currently is ignored)
capacity	`INTEGER`	Capacity of the vehicle.

Description of the result

RETURNS SET OF pgr_costResult[]:

Column	Type	Description
seq	`INTEGER`	Sequential value starting from **1**.
id1	`INTEGER`	Current vehicle identifier.
id2	`INTEGER`	Customer identifier.
cost	`FLOAT`	**Previous `cost` plus *travel time* plus *wait time* plu** • when `id2 = 0` for the second time for the same `id1`, then has the total time for the current `id1`

Examples

Example: Total number of rows returned

```
SELECT count(*) FROM pgr_gsoc_vrppdtw(
    'SELECT * FROM customer ORDER BY id', 25, 200);
 count
-------
   126
(1 row)
```

Example: Results for only id1 values: 1, 5, and 9

```
SELECT * FROM pgr_gsoc_vrppdtw(
    'SELECT * FROM customer ORDER BY id', 25, 200)
    WHERE id1 in (1, 5, 9);
 seq | id1 | id2 |       cost
-----+-----+-----+------------------
   1 |   1 |   0 |                0
   2 |   1 |  13 | 120.805843601499
   3 |   1 |  17 | 214.805843601499
   4 |   1 |  18 | 307.805843601499
   5 |   1 |  19 | 402.805843601499
   6 |   1 |  15 | 497.805843601499
   7 |   1 |  16 | 592.805843601499
   8 |   1 |  14 | 684.805843601499
   9 |   1 |  12 | 777.805843601499
  10 |   1 |  50 | 920.815276724293
  11 |   1 |  52 | 1013.97755438446
  12 |   1 |  49 | 1106.97755438446
  13 |   1 |  47 | 1198.97755438446
```

```
  14 |    1 |    0 | 1217.00531076178
  57 |    5 |    0 |                0
  58 |    5 |   90 | 110.615528128088
  59 |    5 |   87 | 205.615528128088
  60 |    5 |   86 | 296.615528128088
  61 |    5 |   83 | 392.615528128088
  62 |    5 |   82 | 485.615528128088
  63 |    5 |   84 | 581.446480022934
  64 |    5 |   85 |  674.27490714768
  65 |    5 |   88 |  767.27490714768
  66 |    5 |   89 | 860.103334272426
  67 |    5 |   91 |  953.70888554789
  68 |    5 |    0 | 976.069565322888
 105 |    9 |    0 |                0
 106 |    9 |   67 | 102.206555615734
 107 |    9 |   65 | 193.206555615734
 108 |    9 |   63 | 285.206555615734
 109 |    9 |   62 | 380.206555615734
 110 |    9 |   74 | 473.206555615734
 111 |    9 |   72 | 568.206555615734
 112 |    9 |   61 | 661.206555615734
 113 |    9 |   64 | 663.206555615734
 114 |    9 |  102 | 753.206555615734
 115 |    9 |   68 | 846.206555615734
 116 |    9 |    0 | 866.822083743822
(38 rows)
```

See Also

- The examples use *Pick & Deliver Data*
- http://en.wikipedia.org/wiki/Vehicle_routing_problem

Indices and tables

- genindex
- search

6.2.6 pgr_vrpOneDepot - Experimental

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

No documentation available

Example:

Current Result

```
BEGIN;
BEGIN
SET client_min_messages TO NOTICE;
SET
SELECT * FROM pgr_vrpOneDepot(
    'SELECT * FROM vrp_orders',
    'SELECT * FROM vrp_vehicles',
    'SELECT * FROM vrp_distance',
    1);
 oid | opos | vid | tarrival | tdepart
-----+------+-----+----------+---------
  -1 |    1 |   5 |        0 |       0
  66 |    2 |   5 |        0 |       0
  25 |    3 |   5 |        0 |       0
  21 |    4 |   5 |        0 |       0
  84 |    5 |   5 |        0 |       0
  50 |    6 |   5 |        0 |       0
  49 |    7 |   5 |        0 |       0
  24 |    8 |   5 |        0 |       0
  22 |    9 |   5 |        0 |       0
  20 |   10 |   5 |        0 |       0
  19 |   11 |   5 |        0 |       0
  66 |   12 |   5 |       11 |      21
  84 |   13 |   5 |       30 |      45
  24 |   14 |   5 |       71 |      81
  22 |   15 |   5 |       83 |      93
  20 |   16 |   5 |       98 |     108
  19 |   17 |   5 |      114 |     124
  50 |   18 |   5 |      131 |     141
  21 |   19 |   5 |      144 |     154
  25 |   20 |   5 |      158 |     168
  49 |   21 |   5 |      179 |     189
  -1 |   22 |   5 |      234 |     234
  -1 |    1 |   6 |        0 |       0
  31 |    2 |   6 |        0 |       0
  32 |    3 |   6 |        0 |       0
  81 |    4 |   6 |        0 |       0
  94 |    5 |   6 |        0 |       0
  93 |    6 |   6 |        0 |       0
  35 |    7 |   6 |        0 |       0
  33 |    8 |   6 |        0 |       0
  28 |    9 |   6 |        0 |       0
  27 |   10 |   6 |        0 |       0
  93 |   11 |   6 |       15 |      25
  32 |   12 |   6 |       61 |      71
  28 |   13 |   6 |       78 |      88
  31 |   14 |   6 |       97 |     107
  35 |   15 |   6 |      112 |     122
  27 |   16 |   6 |      134 |     144
  33 |   17 |   6 |      152 |     162
  94 |   18 |   6 |      196 |     206
  81 |   19 |   6 |      221 |     231
  -1 |   20 |   6 |      238 |     238
  -1 |    1 |   3 |        0 |       0
  16 |    2 |   3 |        0 |       0
```

```
  14 |    3 |   3 |        0 |        0
  48 |    4 |   3 |        0 |        0
  18 |    5 |   3 |        0 |        0
  17 |    6 |   3 |        0 |        0
  15 |    7 |   3 |        0 |        0
  13 |    8 |   3 |        0 |        0
  11 |    9 |   3 |        0 |        0
  10 |   10 |   3 |        0 |        0
  15 |   11 |   3 |       35 |       45
  48 |   12 |   3 |       48 |       58
  13 |   13 |   3 |       64 |       74
  16 |   14 |   3 |       82 |       92
  17 |   15 |   3 |       94 |      104
  10 |   16 |   3 |      115 |      125
  11 |   17 |   3 |      130 |      140
  14 |   18 |   3 |      147 |      157
  18 |   19 |   3 |      169 |      179
  -1 |   20 |   3 |      219 |      219
  -1 |    1 |   8 |        0 |        0
  71 |    2 |   8 |        0 |        0
  55 |    3 |   8 |        0 |        0
  44 |    4 |   8 |        0 |        0
  43 |    5 |   8 |        0 |        0
  42 |    6 |   8 |        0 |        0
  41 |    7 |   8 |        0 |        0
  40 |    8 |   8 |        0 |        0
  39 |    9 |   8 |        0 |        0
  43 |   10 |   8 |       34 |       44
  40 |   11 |   8 |       49 |       59
  39 |   12 |   8 |       61 |       85
  41 |   13 |   8 |       90 |      100
  42 |   14 |   8 |      111 |      121
  44 |   15 |   8 |      131 |      141
  55 |   16 |   8 |      166 |      176
  71 |   17 |   8 |      198 |      208
  -1 |   18 |   8 |      228 |      228
  -1 |    1 |   1 |        0 |        0
   4 |    2 |   1 |        0 |        0
 101 |    3 |   1 |        0 |        0
  46 |    4 |   1 |        0 |        0
   5 |    5 |   1 |        0 |        0
   3 |    6 |   1 |        0 |        0
  46 |    7 |   1 |       38 |       48
   3 |    8 |   1 |       55 |       65
   2 |    9 |   1 |       96 |       96
   4 |   10 |   1 |      135 |      145
   2 |   11 |   1 |      148 |      158
   5 |   12 |   1 |      165 |      175
 101 |   13 |   1 |      192 |      202
  -1 |   14 |   1 |      222 |      222
  -1 |    1 |  13 |        0 |        0
  92 |    2 |  13 |        0 |        0
  52 |    3 |  13 |        0 |        0
  57 |    4 |  13 |        0 |        0
  85 |    5 |  13 |        0 |        0
  68 |    6 |  13 |        0 |        0
  63 |    7 |  13 |        0 |        0
  63 |    8 |  13 |       29 |       62
  68 |    9 |  13 |       69 |       80
  52 |   10 |  13 |      104 |      114
  85 |   11 |  13 |      123 |      133
  57 |   12 |  13 |      142 |      152
  92 |   13 |  13 |      159 |      177
```

```
  -1 |   14 |  13 |      189 |      189
  -1 |    1 |   7 |        0 |        0
  30 |    2 |   7 |        0 |        0
  29 |    3 |   7 |        0 |        0
  38 |    4 |   7 |        0 |        0
  36 |    5 |   7 |        0 |        0
  34 |    6 |   7 |        0 |        0
  34 |    7 |   7 |       51 |       61
  29 |    8 |   7 |       70 |       80
  30 |    9 |   7 |       85 |       95
  38 |   10 |   7 |      149 |      159
  36 |   11 |   7 |      162 |      172
  -1 |   12 |   7 |      217 |      217
  -1 |    1 |   2 |        0 |        0
  89 |    2 |   2 |        0 |        0
  47 |    3 |   2 |        0 |        0
  61 |    4 |   2 |        0 |        0
   9 |    5 |   2 |        0 |        0
   8 |    6 |   2 |        0 |        0
  89 |    7 |   2 |       18 |       77
   8 |    8 |   2 |       96 |      106
   9 |    9 |   2 |      111 |      121
  47 |   10 |   2 |      124 |      134
  61 |   11 |   2 |      154 |      165
  -1 |   12 |   2 |      192 |      192
  -1 |    1 |  14 |        0 |        0
  97 |    2 |  14 |        0 |        0
  64 |    3 |  14 |        0 |        0
  51 |    4 |  14 |        0 |        0
  96 |    5 |  14 |        0 |        0
  77 |    6 |  14 |        0 |        0
  96 |    7 |  14 |       21 |       44
  64 |    8 |  14 |       63 |       73
  77 |    9 |  14 |       83 |       93
  51 |   10 |  14 |      119 |      129
  97 |   11 |  14 |      154 |      164
  -1 |   12 |  14 |      180 |      180
  -1 |    1 |  15 |        0 |        0
  67 |    2 |  15 |        0 |        0
  73 |    3 |  15 |        0 |        0
  95 |    4 |  15 |        0 |        0
  82 |    5 |  15 |        0 |        0
  72 |    6 |  15 |        0 |        0
  73 |    7 |  15 |       27 |       40
  72 |    8 |  15 |       50 |       75
  82 |    9 |  15 |       91 |      101
  95 |   10 |  15 |      114 |      124
  67 |   11 |  15 |      144 |      154
  -1 |   12 |  15 |      167 |      167
  -1 |    1 |  11 |        0 |        0
  78 |    2 |  11 |        0 |        0
  26 |    3 |  11 |        0 |        0
  87 |    4 |  11 |        0 |        0
  23 |    5 |  11 |        0 |        0
  87 |    6 |  11 |       32 |       97
  23 |    7 |  11 |      118 |      128
  78 |    8 |  11 |      149 |      160
  26 |    9 |  11 |      172 |      182
  -1 |   10 |  11 |      227 |      227
  -1 |    1 |   4 |        0 |        0
  60 |    2 |   4 |        0 |        0
  59 |    3 |   4 |        0 |        0
 100 |    4 |   4 |        0 |        0
```

```
  54 |     5 |    4 |         0 |         0
  60 |     6 |    4 |        42 |        52
 100 |     7 |    4 |        74 |        87
  54 |     8 |    4 |       103 |       113
  59 |     9 |    4 |       153 |       163
  -1 |    10 |    4 |       211 |       211
  -1 |     1 |   10 |         0 |         0
  86 |     2 |   10 |         0 |         0
  90 |     3 |   10 |         0 |         0
  65 |     4 |   10 |         0 |         0
  53 |     5 |   10 |         0 |         0
  53 |     6 |   10 |        25 |        62
  65 |     7 |   10 |        82 |        92
  86 |     8 |   10 |       111 |       121
  90 |     9 |   10 |       140 |       154
  -1 |    10 |   10 |       206 |       206
  -1 |     1 |   12 |         0 |         0
   6 |     2 |   12 |         0 |         0
  80 |     3 |   12 |         0 |         0
   7 |     4 |   12 |         0 |         0
  56 |     5 |   12 |         0 |         0
   6 |     6 |   12 |        40 |        51
  80 |     7 |   12 |        73 |        99
   7 |     8 |   12 |       113 |       123
  56 |     9 |   12 |       142 |       152
  -1 |    10 |   12 |       166 |       166
  -1 |     1 |   19 |         0 |         0
  88 |     2 |   19 |         0 |         0
  70 |     3 |   19 |         0 |         0
  58 |     4 |   19 |         0 |         0
  99 |     5 |   19 |         0 |         0
  70 |     6 |   19 |         9 |        51
  99 |     7 |   19 |        56 |        66
  88 |     8 |   19 |        97 |       107
  58 |     9 |   19 |       125 |       135
  -1 |    10 |   19 |       162 |       162
  -1 |     1 |   17 |         0 |         0
  75 |     2 |   17 |         0 |         0
  98 |     3 |   17 |         0 |         0
  76 |     4 |   17 |         0 |         0
  76 |     5 |   17 |        57 |        84
  98 |     6 |   17 |        97 |       130
  75 |     7 |   17 |       146 |       156
  -1 |     8 |   17 |       192 |       192
  -1 |     1 |   16 |         0 |         0
  69 |     2 |   16 |         0 |         0
  79 |     3 |   16 |         0 |         0
  74 |     4 |   16 |         0 |         0
  74 |     5 |   16 |        39 |        87
  79 |     6 |   16 |        94 |       104
  69 |     7 |   16 |       136 |       154
  -1 |     8 |   16 |       164 |       164
  -1 |     1 |    9 |         0 |         0
  62 |     2 |    9 |         0 |         0
  37 |     3 |    9 |         0 |         0
  45 |     4 |    9 |         0 |         0
  37 |     5 |    9 |        43 |        53
  45 |     6 |    9 |        63 |        74
  62 |     7 |    9 |        94 |       104
  -1 |     8 |    9 |       120 |       120
  -1 |     1 |   18 |         0 |         0
  91 |     2 |   18 |         0 |         0
  12 |     3 |   18 |         0 |         0
```

```
  12 |    4 |  18 |       34 |      69
  91 |    5 |  18 |       99 |     109
  -1 |    6 |  18 |      113 |     113
  -1 |    1 |  20 |        0 |       0
  83 |    2 |  20 |        0 |       0
  83 |    3 |  20 |       15 |      52
  -1 |    4 |  20 |       67 |      67
  -1 |    0 |   0 |       -1 |    3712
(241 rows)

ROLLBACK;
ROLLBACK
```

Data

```
drop table if exists vrp_orders cascade;
create table vrp_orders (
    id integer not null primary key,
    order_unit integer,
    open_time integer,
    close_time integer,
    service_time integer,
    x float8,
    y float8

);

copy vrp_orders (id, x, y, order_unit, open_time, close_time, service_time) from stdin;
1        40.000000        50.000000        0        0        240        0
2        25.000000        85.000000        20        145        175        10
3        22.000000        75.000000        30        50        80        10
4        22.000000        85.000000        10        109        139        10
5        20.000000        80.000000        40        141        171        10
6        20.000000        85.000000        20        41        71        10
7        18.000000        75.000000        20        95        125        10
8        15.000000        75.000000        20        79        109        10
9        15.000000        80.000000        10        91        121        10
10        10.000000        35.000000        20        91        121        10
11        10.000000        40.000000        30        119        149        10
12        8.000000        40.000000        40        59        89        10
13        8.000000        45.000000        20        64        94        10
14        5.000000        35.000000        10        142        172        10
15        5.000000        45.000000        10        35        65        10
16        2.000000        40.000000        20        58        88        10
17        0.000000        40.000000        20        72        102        10
18        0.000000        45.000000        20        149        179        10
19        44.000000        5.000000        20        87        117        10
20        42.000000        10.000000        40        72        102        10
21        42.000000        15.000000        10        122        152        10
22        40.000000        5.000000        10        67        97        10
23        40.000000        15.000000        40        92        122        10
24        38.000000        5.000000        30        65        95        10
25        38.000000        15.000000        10        148        178        10
26        35.000000        5.000000        20        154        184        10
27        95.000000        30.000000        30        115        145        10
28        95.000000        35.000000        20        62        92        10
29        92.000000        30.000000        10        62        92        10
30        90.000000        35.000000        10        67        97        10
31        88.000000        30.000000        10        74        104        10
32        88.000000        35.000000        20        61        91        10
33        87.000000        30.000000        10        131        161        10
```

```
34      85.000000       25.000000       10      51      81      10
35      85.000000       35.000000       30      111     141     10
36      67.000000       85.000000       20      139     169     10
37      65.000000       85.000000       40      43      73      10
38      65.000000       82.000000       10      124     154     10
39      62.000000       80.000000       30      75      105     10
40      60.000000       80.000000       10      37      67      10
41      60.000000       85.000000       30      85      115     10
42      58.000000       75.000000       20      92      122     10
43      55.000000       80.000000       10      33      63      10
44      55.000000       85.000000       20      128     158     10
45      55.000000       82.000000       10      64      94      10
46      20.000000       82.000000       10      37      67      10
47      18.000000       80.000000       10      113     143     10
48      2.000000        45.000000       10      45      75      10
49      42.000000       5.000000        10      151     181     10
50      42.000000       12.000000       10      104     134     10
51      72.000000       35.000000       30      116     146     10
52      55.000000       20.000000       19      83      113     10
53      25.000000       30.000000       3       52      82      10
54      20.000000       50.000000       5       91      121     10
55      55.000000       60.000000       16      139     169     10
56      30.000000       60.000000       16      140     170     10
57      50.000000       35.000000       19      130     160     10
58      30.000000       25.000000       23      96      126     10
59      15.000000       10.000000       20      152     182     10
60      10.000000       20.000000       19      42      72      10
61      15.000000       60.000000       17      155     185     10
62      45.000000       65.000000       9       66      96      10
63      65.000000       35.000000       3       52      82      10
64      65.000000       20.000000       6       39      69      10
65      45.000000       30.000000       17      53      83      10
66      35.000000       40.000000       16      11      41      10
67      41.000000       37.000000       16      133     163     10
68      64.000000       42.000000       9       70      100     10
69      40.000000       60.000000       21      144     174     10
70      31.000000       52.000000       27      41      71      10
71      35.000000       69.000000       23      180     210     10
72      65.000000       55.000000       14      65      95      10
73      63.000000       65.000000       8       30      60      10
74      2.000000        60.000000       5       77      107     10
75      20.000000       20.000000       8       141     171     10
76      5.000000        5.000000        16      74      104     10
77      60.000000       12.000000       31      75      105     10
78      23.000000       3.000000        7       150     180     10
79      8.000000        56.000000       27      90      120     10
80      6.000000        68.000000       30      89      119     10
81      47.000000       47.000000       13      192     222     10
82      49.000000       58.000000       10      86      116     10
83      27.000000       43.000000       9       42      72      10
84      37.000000       31.000000       14      35      65      10
85      57.000000       29.000000       18      96      126     10
86      63.000000       23.000000       2       87      117     10
87      21.000000       24.000000       28      87      117     10
88      12.000000       24.000000       13      90      120     10
89      24.000000       58.000000       19      67      97      10
90      67.000000       5.000000        25      144     174     10
91      37.000000       47.000000       6       86      116     10
92      49.000000       42.000000       13      167     197     10
93      53.000000       43.000000       14      14      44      10
94      61.000000       52.000000       3       178     208     10
95      57.000000       48.000000       23      95      125     10
96      56.000000       37.000000       6       34      64      10
```

```
97        55.000000        54.000000        26        132        162        10
98        4.000000        18.000000        35        120        150        10
99        26.000000        52.000000        9        46        76        10
100        26.000000        35.000000        15        77        107        10
101        31.000000        67.000000        3        180        210        10
\.

drop table if exists vrp_vehicles cascade;
create table vrp_vehicles (
    vehicle_id integer not null primary key,
    capacity integer,
    case_no integer
);

copy vrp_vehicles (vehicle_id, capacity, case_no) from stdin;
1        200        5
2        200        5
3        200        5
4        200        5
5        200        5
6        200        5
7        200        5
8        200        5
9        200        5
10        200        5
11        200        5
12        200        5
13        200        5
14        200        5
15        200        5
16        200        5
17        200        5
18        200        5
19        200        5
20        200        5
\.

drop table if exists vrp_distance cascade;
create table vrp_distance (
    src_id integer,
    dest_id integer,
    cost Float8,
    distance Float8,
    traveltime Float8
);

copy vrp_distance (src_id, dest_id, cost, distance, traveltime) from stdin;
1        2        38.078866        38.078866        38.078866
1        3        30.805844        30.805844        30.805844
1        4        39.357337        39.357337        39.357337
1        5        36.055513        36.055513        36.055513
1        6        40.311289        40.311289        40.311289
1        7        33.301652        33.301652        33.301652
1        8        35.355339        35.355339        35.355339
1        9        39.051248        39.051248        39.051248
1        10        33.541020        33.541020        33.541020
1        11        31.622777        31.622777        31.622777
1        12        33.526109        33.526109        33.526109
1        13        32.388269        32.388269        32.388269
1        14        38.078866        38.078866        38.078866
1        15        35.355339        35.355339        35.355339
1        16        39.293765        39.293765        39.293765
1        17        41.231056        41.231056        41.231056
```

```
1        18         40.311289         40.311289         40.311289
1        19         45.177428         45.177428         45.177428
1        20         40.049969         40.049969         40.049969
1        21         35.057096         35.057096         35.057096
1        22         45.000000         45.000000         45.000000
1        23         35.000000         35.000000         35.000000
1        24         45.044423         45.044423         45.044423
1        25         35.057096         35.057096         35.057096
1        26         45.276926         45.276926         45.276926
1        27         58.523500         58.523500         58.523500
1        28         57.008771         57.008771         57.008771
1        29         55.713553         55.713553         55.713553
1        30         52.201533         52.201533         52.201533
1        31         52.000000         52.000000         52.000000
1        32         50.289164         50.289164         50.289164
1        33         51.078371         51.078371         51.078371
1        34         51.478151         51.478151         51.478151
1        35         47.434165         47.434165         47.434165
1        36         44.204072         44.204072         44.204072
1        37         43.011626         43.011626         43.011626
1        38         40.607881         40.607881         40.607881
1        39         37.202150         37.202150         37.202150
1        40         36.055513         36.055513         36.055513
1        41         40.311289         40.311289         40.311289
1        42         30.805844         30.805844         30.805844
1        43         33.541020         33.541020         33.541020
1        44         38.078866         38.078866         38.078866
1        45         35.341194         35.341194         35.341194
1        46         37.735925         37.735925         37.735925
1        47         37.202150         37.202150         37.202150
1        48         38.327536         38.327536         38.327536
1        49         45.044423         45.044423         45.044423
1        50         38.052595         38.052595         38.052595
1        51         35.341194         35.341194         35.341194
1        52         33.541020         33.541020         33.541020
1        53         25.000000         25.000000         25.000000
1        54         20.000000         20.000000         20.000000
1        55         18.027756         18.027756         18.027756
1        56         14.142136         14.142136         14.142136
1        57         18.027756         18.027756         18.027756
1        58         26.925824         26.925824         26.925824
1        59         47.169906         47.169906         47.169906
1        60         42.426407         42.426407         42.426407
1        61         26.925824         26.925824         26.925824
1        62         15.811388         15.811388         15.811388
1        63         29.154759         29.154759         29.154759
1        64         39.051248         39.051248         39.051248
1        65         20.615528         20.615528         20.615528
1        66         11.180340         11.180340         11.180340
1        67         13.038405         13.038405         13.038405
1        68         25.298221         25.298221         25.298221
1        69         10.000000         10.000000         10.000000
1        70         9.219544         9.219544         9.219544
1        71         19.646883         19.646883         19.646883
1        72         25.495098         25.495098         25.495098
1        73         27.459060         27.459060         27.459060
1        74         39.293765         39.293765         39.293765
1        75         36.055513         36.055513         36.055513
1        76         57.008771         57.008771         57.008771
1        77         42.941821         42.941821         42.941821
1        78         49.979996         49.979996         49.979996
1        79         32.557641         32.557641         32.557641
1        80         38.470768         38.470768         38.470768
```

```
1        81        7.615773        7.615773        7.615773
1        82        12.041595        12.041595        12.041595
1        83        14.764823        14.764823        14.764823
1        84        19.235384        19.235384        19.235384
1        85        27.018512        27.018512        27.018512
1        86        35.468296        35.468296        35.468296
1        87        32.202484        32.202484        32.202484
1        88        38.209946        38.209946        38.209946
1        89        17.888544        17.888544        17.888544
1        90        52.478567        52.478567        52.478567
1        91        4.242641        4.242641        4.242641
1        92        12.041595        12.041595        12.041595
1        93        14.764823        14.764823        14.764823
1        94        21.095023        21.095023        21.095023
1        95        17.117243        17.117243        17.117243
1        96        20.615528        20.615528        20.615528
1        97        15.524175        15.524175        15.524175
1        98        48.166378        48.166378        48.166378
1        99        14.142136        14.142136        14.142136
1        100        20.518285        20.518285        20.518285
1        101        19.235384        19.235384        19.235384
2        1        38.078866        38.078866        38.078866
2        3        10.440307        10.440307        10.440307
2        4        3.000000        3.000000        3.000000
2        5        7.071068        7.071068        7.071068
2        6        5.000000        5.000000        5.000000
2        7        12.206556        12.206556        12.206556
2        8        14.142136        14.142136        14.142136
2        9        11.180340        11.180340        11.180340
2        10        52.201533        52.201533        52.201533
2        11        47.434165        47.434165        47.434165
2        12        48.104054        48.104054        48.104054
2        13        43.462628        43.462628        43.462628
2        14        53.851648        53.851648        53.851648
2        15        44.721360        44.721360        44.721360
2        16        50.537115        50.537115        50.537115
2        17        51.478151        51.478151        51.478151
2        18        47.169906        47.169906        47.169906
2        19        82.225300        82.225300        82.225300
2        20        76.902536        76.902536        76.902536
2        21        72.034714        72.034714        72.034714
2        22        81.394103        81.394103        81.394103
2        23        71.589105        71.589105        71.589105
2        24        81.049368        81.049368        81.049368
2        25        71.196910        71.196910        71.196910
2        26        80.622577        80.622577        80.622577
2        27        89.022469        89.022469        89.022469
2        28        86.023253        86.023253        86.023253
2        29        86.683332        86.683332        86.683332
2        30        82.006097        82.006097        82.006097
2        31        83.630138        83.630138        83.630138
2        32        80.430094        80.430094        80.430094
2        33        82.879430        82.879430        82.879430
2        34        84.852814        84.852814        84.852814
2        35        78.102497        78.102497        78.102497
2        36        42.000000        42.000000        42.000000
2        37        40.000000        40.000000        40.000000
2        38        40.112342        40.112342        40.112342
2        39        37.336309        37.336309        37.336309
2        40        35.355339        35.355339        35.355339
2        41        35.000000        35.000000        35.000000
2        42        34.481879        34.481879        34.481879
2        43        30.413813        30.413813        30.413813
```

```
2        44        30.000000        30.000000        30.000000
2        45        30.149627        30.149627        30.149627
2        46        5.830952        5.830952        5.830952
2        47        8.602325        8.602325        8.602325
2        48        46.141088        46.141088        46.141088
2        49        81.786307        81.786307        81.786307
2        50        74.953319        74.953319        74.953319
2        51        68.622154        68.622154        68.622154
2        52        71.589105        71.589105        71.589105
2        53        55.000000        55.000000        55.000000
2        54        35.355339        35.355339        35.355339
2        55        39.051248        39.051248        39.051248
2        56        25.495098        25.495098        25.495098
2        57        55.901699        55.901699        55.901699
2        58        60.207973        60.207973        60.207973
2        59        75.663730        75.663730        75.663730
2        60        66.708320        66.708320        66.708320
2        61        26.925824        26.925824        26.925824
2        62        28.284271        28.284271        28.284271
2        63        64.031242        64.031242        64.031242
2        64        76.321688        76.321688        76.321688
2        65        58.523500        58.523500        58.523500
2        66        46.097722        46.097722        46.097722
2        67        50.596443        50.596443        50.596443
2        68        58.051701        58.051701        58.051701
2        69        29.154759        29.154759        29.154759
2        70        33.541020        33.541020        33.541020
2        71        18.867962        18.867962        18.867962
2        72        50.000000        50.000000        50.000000
2        73        42.941821        42.941821        42.941821
2        74        33.970576        33.970576        33.970576
2        75        65.192024        65.192024        65.192024
2        76        82.462113        82.462113        82.462113
2        77        80.956779        80.956779        80.956779
2        78        82.024387        82.024387        82.024387
2        79        33.615473        33.615473        33.615473
2        80        25.495098        25.495098        25.495098
2        81        43.908997        43.908997        43.908997
2        82        36.124784        36.124784        36.124784
2        83        42.047592        42.047592        42.047592
2        84        55.317267        55.317267        55.317267
2        85        64.498062        64.498062        64.498062
2        86        72.718636        72.718636        72.718636
2        87        61.131007        61.131007        61.131007
2        88        62.369865        62.369865        62.369865
2        89        27.018512        27.018512        27.018512
2        90        90.354856        90.354856        90.354856
2        91        39.849718        39.849718        39.849718
2        92        49.244289        49.244289        49.244289
2        93        50.477718        50.477718        50.477718
2        94        48.836462        48.836462        48.836462
2        95        48.918299        48.918299        48.918299
2        96        57.140179        57.140179        57.140179
2        97        43.139309        43.139309        43.139309
2        98        70.213959        70.213959        70.213959
2        99        33.015148        33.015148        33.015148
2        100        50.009999        50.009999        50.009999
2        101        18.973666        18.973666        18.973666
3        1        30.805844        30.805844        30.805844
3        2        10.440307        10.440307        10.440307
3        4        10.000000        10.000000        10.000000
3        5        5.385165        5.385165        5.385165
3        6        10.198039        10.198039        10.198039
```

```
3        7        4.000000        4.000000        4.000000
3        8        7.000000        7.000000        7.000000
3        9        8.602325        8.602325        8.602325
3        10        41.761226        41.761226        41.761226
3        11        37.000000        37.000000        37.000000
3        12        37.696154        37.696154        37.696154
3        13        33.105891        33.105891        33.105891
3        14        43.462628        43.462628        43.462628
3        15        34.481879        34.481879        34.481879
3        16        40.311289        40.311289        40.311289
3        17        41.340053        41.340053        41.340053
3        18        37.202150        37.202150        37.202150
3        19        73.375745        73.375745        73.375745
3        20        68.007353        68.007353        68.007353
3        21        63.245553        63.245553        63.245553
3        22        72.277244        72.277244        72.277244
3        23        62.641839        62.641839        62.641839
3        24        71.805292        71.805292        71.805292
3        25        62.096699        62.096699        62.096699
3        26        71.196910        71.196910        71.196910
3        27        85.755466        85.755466        85.755466
3        28        83.240615        83.240615        83.240615
3        29        83.216585        83.216585        83.216585
3        30        78.892332        78.892332        78.892332
3        31        79.881162        79.881162        79.881162
3        32        77.175126        77.175126        77.175126
3        33        79.056942        79.056942        79.056942
3        34        80.430094        80.430094        80.430094
3        35        74.625733        74.625733        74.625733
3        36        46.097722        46.097722        46.097722
3        37        44.147480        44.147480        44.147480
3        38        43.566042        43.566042        43.566042
3        39        40.311289        40.311289        40.311289
3        40        38.327536        38.327536        38.327536
3        41        39.293765        39.293765        39.293765
3        42        36.000000        36.000000        36.000000
3        43        33.376639        33.376639        33.376639
3        44        34.481879        34.481879        34.481879
3        45        33.734256        33.734256        33.734256
3        46        7.280110        7.280110        7.280110
3        47        6.403124        6.403124        6.403124
3        48        36.055513        36.055513        36.055513
3        49        72.801099        72.801099        72.801099
3        50        66.098411        66.098411        66.098411
3        51        64.031242        64.031242        64.031242
3        52        64.140471        64.140471        64.140471
3        53        45.099889        45.099889        45.099889
3        54        25.079872        25.079872        25.079872
3        55        36.249138        36.249138        36.249138
3        56        17.000000        17.000000        17.000000
3        57        48.826222        48.826222        48.826222
3        58        50.635956        50.635956        50.635956
3        59        65.375837        65.375837        65.375837
3        60        56.293872        56.293872        56.293872
3        61        16.552945        16.552945        16.552945
3        62        25.079872        25.079872        25.079872
3        63        58.728187        58.728187        58.728187
3        64        69.814039        69.814039        69.814039
3        65        50.537115        50.537115        50.537115
3        66        37.336309        37.336309        37.336309
3        67        42.485292        42.485292        42.485292
3        68        53.413481        53.413481        53.413481
3        69        23.430749        23.430749        23.430749
```

```
3        70        24.698178        24.698178        24.698178
3        71        14.317821        14.317821        14.317821
3        72        47.423623        47.423623        47.423623
3        73        42.201896        42.201896        42.201896
3        74        25.000000        25.000000        25.000000
3        75        55.036352        55.036352        55.036352
3        76        72.034714        72.034714        72.034714
3        77        73.573093        73.573093        73.573093
3        78        72.006944        72.006944        72.006944
3        79        23.600847        23.600847        23.600847
3        80        17.464249        17.464249        17.464249
3        81        37.536649        37.536649        37.536649
3        82        31.906112        31.906112        31.906112
3        83        32.388269        32.388269        32.388269
3        84        46.486557        46.486557        46.486557
3        85        57.801384        57.801384        57.801384
3        86        66.219333        66.219333        66.219333
3        87        51.009803        51.009803        51.009803
3        88        51.971146        51.971146        51.971146
3        89        17.117243        17.117243        17.117243
3        90        83.216585        83.216585        83.216585
3        91        31.764760        31.764760        31.764760
3        92        42.638011        42.638011        42.638011
3        93        44.553339        44.553339        44.553339
3        94        45.276926        45.276926        45.276926
3        95        44.204072        44.204072        44.204072
3        96        50.990195        50.990195        50.990195
3        97        39.115214        39.115214        39.115214
3        98        59.774577        59.774577        59.774577
3        99        23.345235        23.345235        23.345235
3        100        40.199502        40.199502        40.199502
3        101        12.041595        12.041595        12.041595
4        1        39.357337        39.357337        39.357337
4        2        3.000000        3.000000        3.000000
4        3        10.000000        10.000000        10.000000
4        5        5.385165        5.385165        5.385165
4        6        2.000000        2.000000        2.000000
4        7        10.770330        10.770330        10.770330
4        8        12.206556        12.206556        12.206556
4        9        8.602325        8.602325        8.602325
4        10        51.419841        51.419841        51.419841
4        11        46.572524        46.572524        46.572524
4        12        47.127487        47.127487        47.127487
4        13        42.379240        42.379240        42.379240
4        14        52.810984        52.810984        52.810984
4        15        43.462628        43.462628        43.462628
4        16        49.244289        49.244289        49.244289
4        17        50.089919        50.089919        50.089919
4        18        45.650849        45.650849        45.650849
4        19        82.969874        82.969874        82.969874
4        20        77.620873        77.620873        77.620873
4        21        72.801099        72.801099        72.801099
4        22        82.000000        82.000000        82.000000
4        23        72.277244        72.277244        72.277244
4        24        81.584312        81.584312        81.584312
4        25        71.805292        71.805292        71.805292
4        26        81.049368        81.049368        81.049368
4        27        91.400219        91.400219        91.400219
4        28        88.481637        88.481637        88.481637
4        29        89.022469        89.022469        89.022469
4        30        84.403791        84.403791        84.403791
4        31        85.912746        85.912746        85.912746
4        32        82.800966        82.800966        82.800966
```

```
4        33       85.146932        85.146932        85.146932
4        34       87.000000        87.000000        87.000000
4        35       80.430094        80.430094        80.430094
4        36       45.000000        45.000000        45.000000
4        37       43.000000        43.000000        43.000000
4        38       43.104524        43.104524        43.104524
4        39       40.311289        40.311289        40.311289
4        40       38.327536        38.327536        38.327536
4        41       38.000000        38.000000        38.000000
4        42       37.363083        37.363083        37.363083
4        43       33.376639        33.376639        33.376639
4        44       33.000000        33.000000        33.000000
4        45       33.136083        33.136083        33.136083
4        46       3.605551        3.605551        3.605551
4        47       6.403124        6.403124        6.403124
4        48       44.721360        44.721360        44.721360
4        49       82.462113        82.462113        82.462113
4        50       75.690158        75.690158        75.690158
4        51       70.710678        70.710678        70.710678
4        52       72.897188        72.897188        72.897188
4        53       55.081757        55.081757        55.081757
4        54       35.057096        35.057096        35.057096
4        55       41.400483        41.400483        41.400483
4        56       26.248809        26.248809        26.248809
4        57       57.306195        57.306195        57.306195
4        58       60.530984        60.530984        60.530984
4        59       75.325958        75.325958        75.325958
4        60       66.098411        66.098411        66.098411
4        61       25.961510        25.961510        25.961510
4        62       30.479501        30.479501        30.479501
4        63       65.946948        65.946948        65.946948
4        64       77.935871        77.935871        77.935871
4        65       59.615434        59.615434        59.615434
4        66       46.840154        46.840154        46.840154
4        67       51.623638        51.623638        51.623638
4        68       60.108236        60.108236        60.108236
4        69       30.805844        30.805844        30.805844
4        70       34.205263        34.205263        34.205263
4        71       20.615528        20.615528        20.615528
4        72       52.430907        52.430907        52.430907
4        73       45.617979        45.617979        45.617979
4        74       32.015621        32.015621        32.015621
4        75       65.030762        65.030762        65.030762
4        76       81.786307        81.786307        81.786307
4        77       82.298238        82.298238        82.298238
4        78       82.006097        82.006097        82.006097
4        79       32.202484        32.202484        32.202484
4        80       23.345235        23.345235        23.345235
4        81       45.486262        45.486262        45.486262
4        82       38.183766        38.183766        38.183766
4        83       42.296572        42.296572        42.296572
4        84       56.044625        56.044625        56.044625
4        85       66.037868        66.037868        66.037868
4        86       74.330344        74.330344        74.330344
4        87       61.008196        61.008196        61.008196
4        88       61.814238        61.814238        61.814238
4        89       27.073973        27.073973        27.073973
4        90       91.787799        91.787799        91.787799
4        91       40.853396        40.853396        40.853396
4        92       50.774009        50.774009        50.774009
4        93       52.201533        52.201533        52.201533
4        94       51.088159        51.088159        51.088159
4        95       50.931326        50.931326        50.931326
```

```
4        96        58.821765        58.821765        58.821765
4        97        45.276926        45.276926        45.276926
4        98        69.375788        69.375788        69.375788
4        99        33.241540        33.241540        33.241540
4        100        50.159745        50.159745        50.159745
4        101        20.124612        20.124612        20.124612
5        1        36.055513        36.055513        36.055513
5        2        7.071068        7.071068        7.071068
5        3        5.385165        5.385165        5.385165
5        4        5.385165        5.385165        5.385165
5        6        5.000000        5.000000        5.000000
5        7        5.385165        5.385165        5.385165
5        8        7.071068        7.071068        7.071068
5        9        5.000000        5.000000        5.000000
5        10        46.097722        46.097722        46.097722
5        11        41.231056        41.231056        41.231056
5        12        41.761226        41.761226        41.761226
5        13        37.000000        37.000000        37.000000
5        14        47.434165        47.434165        47.434165
5        15        38.078866        38.078866        38.078866
5        16        43.863424        43.863424        43.863424
5        17        44.721360        44.721360        44.721360
5        18        40.311289        40.311289        40.311289
5        19        78.746428        78.746428        78.746428
5        20        73.375745        73.375745        73.375745
5        21        68.622154        68.622154        68.622154
5        22        77.620873        77.620873        77.620873
5        23        68.007353        68.007353        68.007353
5        24        77.129761        77.129761        77.129761
5        25        67.446275        67.446275        67.446275
5        26        76.485293        76.485293        76.485293
5        27        90.138782        90.138782        90.138782
5        28        87.464278        87.464278        87.464278
5        29        87.658428        87.658428        87.658428
5        30        83.216585        83.216585        83.216585
5        31        84.403791        84.403791        84.403791
5        32        81.541401        81.541401        81.541401
5        33        83.600239        83.600239        83.600239
5        34        85.146932        85.146932        85.146932
5        35        79.056942        79.056942        79.056942
5        36        47.265209        47.265209        47.265209
5        37        45.276926        45.276926        45.276926
5        38        45.044423        45.044423        45.044423
5        39        42.000000        42.000000        42.000000
5        40        40.000000        40.000000        40.000000
5        41        40.311289        40.311289        40.311289
5        42        38.327536        38.327536        38.327536
5        43        35.000000        35.000000        35.000000
5        44        35.355339        35.355339        35.355339
5        45        35.057096        35.057096        35.057096
5        46        2.000000        2.000000        2.000000
5        47        2.000000        2.000000        2.000000
5        48        39.357337        39.357337        39.357337
5        49        78.160092        78.160092        78.160092
5        50        71.470274        71.470274        71.470274
5        51        68.767725        68.767725        68.767725
5        52        69.462220        69.462220        69.462220
5        53        50.249378        50.249378        50.249378
5        54        30.000000        30.000000        30.000000
5        55        40.311289        40.311289        40.311289
5        56        22.360680        22.360680        22.360680
5        57        54.083269        54.083269        54.083269
5        58        55.901699        55.901699        55.901699
```

```
5        59         70.178344        70.178344        70.178344
5        60         60.827625        60.827625        60.827625
5        61         20.615528        20.615528        20.615528
5        62         29.154759        29.154759        29.154759
5        63         63.639610        63.639610        63.639610
5        64         75.000000        75.000000        75.000000
5        65         55.901699        55.901699        55.901699
5        66         42.720019        42.720019        42.720019
5        67         47.853944        47.853944        47.853944
5        68         58.137767        58.137767        58.137767
5        69         28.284271        28.284271        28.284271
5        70         30.083218        30.083218        30.083218
5        71         18.601075        18.601075        18.601075
5        72         51.478151        51.478151        51.478151
5        73         45.541190        45.541190        45.541190
5        74         26.907248        26.907248        26.907248
5        75         60.000000        60.000000        60.000000
5        76         76.485293        76.485293        76.485293
5        77         78.892332        78.892332        78.892332
5        78         77.058419        77.058419        77.058419
5        79         26.832816        26.832816        26.832816
5        80         18.439089        18.439089        18.439089
5        81         42.638011        42.638011        42.638011
5        82         36.400549        36.400549        36.400549
5        83         37.656341        37.656341        37.656341
5        84         51.865210        51.865210        51.865210
5        85         63.007936        63.007936        63.007936
5        86         71.400280        71.400280        71.400280
5        87         56.008928        56.008928        56.008928
5        88         56.568542        56.568542        56.568542
5        89         22.360680        22.360680        22.360680
5        90         88.509886        88.509886        88.509886
5        91         37.121422        37.121422        37.121422
5        92         47.801674        47.801674        47.801674
5        93         49.578221        49.578221        49.578221
5        94         49.648766        49.648766        49.648766
5        95         48.918299        48.918299        48.918299
5        96         56.080300        56.080300        56.080300
5        97         43.600459        43.600459        43.600459
5        98         64.031242        64.031242        64.031242
5        99         28.635642        28.635642        28.635642
5        100         45.398238        45.398238        45.398238
5        101         17.029386        17.029386        17.029386
6        1         40.311289        40.311289        40.311289
6        2         5.000000        5.000000        5.000000
6        3         10.198039        10.198039        10.198039
6        4         2.000000        2.000000        2.000000
6        5         5.000000        5.000000        5.000000
6        7         10.198039        10.198039        10.198039
6        8         11.180340        11.180340        11.180340
6        9         7.071068        7.071068        7.071068
6        10         50.990195        50.990195        50.990195
6        11         46.097722        46.097722        46.097722
6        12         46.572524        46.572524        46.572524
6        13         41.761226        41.761226        41.761226
6        14         52.201533        52.201533        52.201533
6        15         42.720019        42.720019        42.720019
6        16         48.466483        48.466483        48.466483
6        17         49.244289        49.244289        49.244289
6        18         44.721360        44.721360        44.721360
6        19         83.522452        83.522452        83.522452
6        20         78.160092        78.160092        78.160092
6        21         73.375745        73.375745        73.375745
```

```
6        22        82.462113        82.462113        82.462113
6        23        72.801099        72.801099        72.801099
6        24        82.000000        82.000000        82.000000
6        25        72.277244        72.277244        72.277244
6        26        81.394103        81.394103        81.394103
6        27        93.005376        93.005376        93.005376
6        28        90.138782        90.138782        90.138782
6        29        90.603532        90.603532        90.603532
6        30        86.023253        86.023253        86.023253
6        31        87.458562        87.458562        87.458562
6        32        84.403791        84.403791        84.403791
6        33        86.683332        86.683332        86.683332
6        34        88.459030        88.459030        88.459030
6        35        82.006097        82.006097        82.006097
6        36        47.000000        47.000000        47.000000
6        37        45.000000        45.000000        45.000000
6        38        45.099889        45.099889        45.099889
6        39        42.296572        42.296572        42.296572
6        40        40.311289        40.311289        40.311289
6        41        40.000000        40.000000        40.000000
6        42        39.293765        39.293765        39.293765
6        43        35.355339        35.355339        35.355339
6        44        35.000000        35.000000        35.000000
6        45        35.128336        35.128336        35.128336
6        46        3.000000        3.000000        3.000000
6        47        5.385165        5.385165        5.385165
6        48        43.863424        43.863424        43.863424
6        49        82.969874        82.969874        82.969874
6        50        76.243032        76.243032        76.243032
6        51        72.138755        72.138755        72.138755
6        52        73.824115        73.824115        73.824115
6        53        55.226805        55.226805        55.226805
6        54        35.000000        35.000000        35.000000
6        55        43.011626        43.011626        43.011626
6        56        26.925824        26.925824        26.925824
6        57        58.309519        58.309519        58.309519
6        58        60.827625        60.827625        60.827625
6        59        75.166482        75.166482        75.166482
6        60        65.764732        65.764732        65.764732
6        61        25.495098        25.495098        25.495098
6        62        32.015621        32.015621        32.015621
6        63        67.268120        67.268120        67.268120
6        64        79.056942        79.056942        79.056942
6        65        60.415230        60.415230        60.415230
6        66        47.434165        47.434165        47.434165
6        67        52.392748        52.392748        52.392748
6        68        61.522354        61.522354        61.522354
6        69        32.015621        32.015621        32.015621
6        70        34.785054        34.785054        34.785054
6        71        21.931712        21.931712        21.931712
6        72        54.083269        54.083269        54.083269
6        73        47.423623        47.423623        47.423623
6        74        30.805844        30.805844        30.805844
6        75        65.000000        65.000000        65.000000
6        76        81.394103        81.394103        81.394103
6        77        83.240615        83.240615        83.240615
6        78        82.054860        82.054860        82.054860
6        79        31.384710        31.384710        31.384710
6        80        22.022716        22.022716        22.022716
6        81        46.615448        46.615448        46.615448
6        82        39.623226        39.623226        39.623226
6        83        42.579338        42.579338        42.579338
6        84        56.612719        56.612719        56.612719
```

```
6        85        67.119297        67.119297        67.119297
6        86        75.451971        75.451971        75.451971
6        87        61.008196        61.008196        61.008196
6        88        61.522354        61.522354        61.522354
6        89        27.294688        27.294688        27.294688
6        90        92.784697        92.784697        92.784697
6        91        41.629317        41.629317        41.629317
6        92        51.865210        51.865210        51.865210
6        93        53.413481        53.413481        53.413481
6        94        52.630789        52.630789        52.630789
6        95        52.325902        52.325902        52.325902
6        96        60.000000        60.000000        60.000000
6        97        46.754679        46.754679        46.754679
6        98        68.883960        68.883960        68.883960
6        99        33.541020        33.541020        33.541020
6        100        50.358713        50.358713        50.358713
6        101        21.095023        21.095023        21.095023
7        1        33.301652        33.301652        33.301652
7        2        12.206556        12.206556        12.206556
7        3        4.000000        4.000000        4.000000
7        4        10.770330        10.770330        10.770330
7        5        5.385165        5.385165        5.385165
7        6        10.198039        10.198039        10.198039
7        8        3.000000        3.000000        3.000000
7        9        5.830952        5.830952        5.830952
7        10        40.792156        40.792156        40.792156
7        11        35.902646        35.902646        35.902646
7        12        36.400549        36.400549        36.400549
7        13        31.622777        31.622777        31.622777
7        14        42.059482        42.059482        42.059482
7        15        32.695565        32.695565        32.695565
7        16        38.483763        38.483763        38.483763
7        17        39.357337        39.357337        39.357337
7        18        34.985711        34.985711        34.985711
7        19        74.672619        74.672619        74.672619
7        20        69.289249        69.289249        69.289249
7        21        64.621978        64.621978        64.621978
7        22        73.375745        73.375745        73.375745
7        23        63.906181        63.906181        63.906181
7        24        72.801099        72.801099        72.801099
7        25        63.245553        63.245553        63.245553
7        26        72.034714        72.034714        72.034714
7        27        89.185201        89.185201        89.185201
7        28        86.769810        86.769810        86.769810
7        29        86.608314        86.608314        86.608314
7        30        82.365041        82.365041        82.365041
7        31        83.216585        83.216585        83.216585
7        32        80.622577        80.622577        80.622577
7        33        82.377181        82.377181        82.377181
7        34        83.600239        83.600239        83.600239
7        35        78.032045        78.032045        78.032045
7        36        50.009999        50.009999        50.009999
7        37        48.052055        48.052055        48.052055
7        38        47.518417        47.518417        47.518417
7        39        44.283180        44.283180        44.283180
7        40        42.296572        42.296572        42.296572
7        41        43.174066        43.174066        43.174066
7        42        40.000000        40.000000        40.000000
7        43        37.336309        37.336309        37.336309
7        44        38.327536        38.327536        38.327536
7        45        37.656341        37.656341        37.656341
7        46        7.280110        7.280110        7.280110
7        47        5.000000        5.000000        5.000000
```

```
7        48        34.000000        34.000000        34.000000
7        49        74.000000        74.000000        74.000000
7        50        67.416615        67.416615        67.416615
7        51        67.201190        67.201190        67.201190
7        52        66.287254        66.287254        66.287254
7        53        45.541190        45.541190        45.541190
7        54        25.079872        25.079872        25.079872
7        55        39.924930        39.924930        39.924930
7        56        19.209373        19.209373        19.209373
7        57        51.224994        51.224994        51.224994
7        58        51.419841        51.419841        51.419841
7        59        65.069194        65.069194        65.069194
7        60        55.578773        55.578773        55.578773
7        61        15.297059        15.297059        15.297059
7        62        28.792360        28.792360        28.792360
7        63        61.717096        61.717096        61.717096
7        64        72.346389        72.346389        72.346389
7        65        52.478567        52.478567        52.478567
7        66        38.910153        38.910153        38.910153
7        67        44.418465        44.418465        44.418465
7        68        56.612719        56.612719        56.612719
7        69        26.627054        26.627054        26.627054
7        70        26.419690        26.419690        26.419690
7        71        18.027756        18.027756        18.027756
7        72        51.078371        51.078371        51.078371
7        73        46.097722        46.097722        46.097722
7        74        21.931712        21.931712        21.931712
7        75        55.036352        55.036352        55.036352
7        76        71.196910        71.196910        71.196910
7        77        75.716577        75.716577        75.716577
7        78        72.173402        72.173402        72.173402
7        79        21.470911        21.470911        21.470911
7        80        13.892444        13.892444        13.892444
7        81        40.311289        40.311289        40.311289
7        82        35.355339        35.355339        35.355339
7        83        33.241540        33.241540        33.241540
7        84        47.927028        47.927028        47.927028
7        85        60.307545        60.307545        60.307545
7        86        68.767725        68.767725        68.767725
7        87        51.088159        51.088159        51.088159
7        88        51.351728        51.351728        51.351728
7        89        18.027756        18.027756        18.027756
7        90        85.445889        85.445889        85.445889
7        91        33.837849        33.837849        33.837849
7        92        45.276926        45.276926        45.276926
7        93        47.423623        47.423623        47.423623
7        94        48.764741        48.764741        48.764741
7        95        47.434165        47.434165        47.434165
7        96        53.740115        53.740115        53.740115
7        97        42.544095        42.544095        42.544095
7        98        58.694122        58.694122        58.694122
7        99        24.351591        24.351591        24.351591
7        100        40.792156        40.792156        40.792156
7        101        15.264338        15.264338        15.264338
8        1        35.355339        35.355339        35.355339
8        2        14.142136        14.142136        14.142136
8        3        7.000000        7.000000        7.000000
8        4        12.206556        12.206556        12.206556
8        5        7.071068        7.071068        7.071068
8        6        11.180340        11.180340        11.180340
8        7        3.000000        3.000000        3.000000
8        9        5.000000        5.000000        5.000000
8        10        40.311289        40.311289        40.311289
```

```
8        11        35.355339        35.355339        35.355339
8        12        35.693137        35.693137        35.693137
8        13        30.805844        30.805844        30.805844
8        14        41.231056        41.231056        41.231056
8        15        31.622777        31.622777        31.622777
8        16        37.336309        37.336309        37.336309
8        17        38.078866        38.078866        38.078866
8        18        33.541020        33.541020        33.541020
8        19        75.769387        75.769387        75.769387
8        20        70.384657        70.384657        70.384657
8        21        65.795137        65.795137        65.795137
8        22        74.330344        74.330344        74.330344
8        23        65.000000        65.000000        65.000000
8        24        73.681748        73.681748        73.681748
8        25        64.257295        64.257295        64.257295
8        26        72.801099        72.801099        72.801099
8        27        91.787799        91.787799        91.787799
8        28        89.442719        89.442719        89.442719
8        29        89.185201        89.185201        89.185201
8        30        85.000000        85.000000        85.000000
8        31        85.755466        85.755466        85.755466
8        32        83.240615        83.240615        83.240615
8        33        84.905830        84.905830        84.905830
8        34        86.023253        86.023253        86.023253
8        35        80.622577        80.622577        80.622577
8        36        52.952809        52.952809        52.952809
8        37        50.990195        50.990195        50.990195
8        38        50.487622        50.487622        50.487622
8        39        47.265209        47.265209        47.265209
8        40        45.276926        45.276926        45.276926
8        41        46.097722        46.097722        46.097722
8        42        43.000000        43.000000        43.000000
8        43        40.311289        40.311289        40.311289
8        44        41.231056        41.231056        41.231056
8        45        40.607881        40.607881        40.607881
8        46        8.602325        8.602325        8.602325
8        47        5.830952        5.830952        5.830952
8        48        32.695565        32.695565        32.695565
8        49        75.026662        75.026662        75.026662
8        50        68.541958        68.541958        68.541958
8        51        69.634761        69.634761        69.634761
8        52        68.007353        68.007353        68.007353
8        53        46.097722        46.097722        46.097722
8        54        25.495098        25.495098        25.495098
8        55        42.720019        42.720019        42.720019
8        56        21.213203        21.213203        21.213203
8        57        53.150729        53.150729        53.150729
8        58        52.201533        52.201533        52.201533
8        59        65.000000        65.000000        65.000000
8        60        55.226805        55.226805        55.226805
8        61        15.000000        15.000000        15.000000
8        62        31.622777        31.622777        31.622777
8        63        64.031242        64.031242        64.031242
8        64        74.330344        74.330344        74.330344
8        65        54.083269        54.083269        54.083269
8        66        40.311289        40.311289        40.311289
8        67        46.043458        46.043458        46.043458
8        68        59.076222        59.076222        59.076222
8        69        29.154759        29.154759        29.154759
8        70        28.017851        28.017851        28.017851
8        71        20.880613        20.880613        20.880613
8        72        53.851648        53.851648        53.851648
8        73        49.030603        49.030603        49.030603
```

```
8        74        19.849433        19.849433        19.849433
8        75        55.226805        55.226805        55.226805
8        76        70.710678        70.710678        70.710678
8        77        77.420927        77.420927        77.420927
8        78        72.443081        72.443081        72.443081
8        79        20.248457        20.248457        20.248457
8        80        11.401754        11.401754        11.401754
8        81        42.520583        42.520583        42.520583
8        82        38.013156        38.013156        38.013156
8        83        34.176015        34.176015        34.176015
8        84        49.193496        49.193496        49.193496
8        85        62.289646        62.289646        62.289646
8        86        70.767224        70.767224        70.767224
8        87        51.351728        51.351728        51.351728
8        88        51.088159        51.088159        51.088159
8        89        19.235384        19.235384        19.235384
8        90        87.200917        87.200917        87.200917
8        91        35.608988        35.608988        35.608988
8        92        47.381431        47.381431        47.381431
8        93        49.678969        49.678969        49.678969
8        94        51.429563        51.429563        51.429563
8        95        49.929951        49.929951        49.929951
8        96        55.901699        55.901699        55.901699
8        97        45.177428        45.177428        45.177428
8        98        58.051701        58.051701        58.051701
8        99        25.495098        25.495098        25.495098
8        100        41.484937        41.484937        41.484937
8        101        17.888544        17.888544        17.888544
9        1        39.051248        39.051248        39.051248
9        2        11.180340        11.180340        11.180340
9        3        8.602325        8.602325        8.602325
9        4        8.602325        8.602325        8.602325
9        5        5.000000        5.000000        5.000000
9        6        7.071068        7.071068        7.071068
9        7        5.830952        5.830952        5.830952
9        8        5.000000        5.000000        5.000000
9        10        45.276926        45.276926        45.276926
9        11        40.311289        40.311289        40.311289
9        12        40.607881        40.607881        40.607881
9        13        35.693137        35.693137        35.693137
9        14        46.097722        46.097722        46.097722
9        15        36.400549        36.400549        36.400549
9        16        42.059482        42.059482        42.059482
9        17        42.720019        42.720019        42.720019
9        18        38.078866        38.078866        38.078866
9        19        80.411442        80.411442        80.411442
9        20        75.026662        75.026662        75.026662
9        21        70.384657        70.384657        70.384657
9        22        79.056942        79.056942        79.056942
9        23        69.641941        69.641941        69.641941
9        24        78.447435        78.447435        78.447435
9        25        68.949257        68.949257        68.949257
9        26        77.620873        77.620873        77.620873
9        27        94.339811        94.339811        94.339811
9        28        91.787799        91.787799        91.787799
9        29        91.809586        91.809586        91.809586
9        30        87.464278        87.464278        87.464278
9        31        88.481637        88.481637        88.481637
9        32        85.755466        85.755466        85.755466
9        33        87.658428        87.658428        87.658428
9        34        89.022469        89.022469        89.022469
9        35        83.216585        83.216585        83.216585
9        36        52.239832        52.239832        52.239832
```

```
9        37         50.249378         50.249378         50.249378
9        38         50.039984         50.039984         50.039984
9        39         47.000000         47.000000         47.000000
9        40         45.000000         45.000000         45.000000
9        41         45.276926         45.276926         45.276926
9        42         43.289722         43.289722         43.289722
9        43         40.000000         40.000000         40.000000
9        44         40.311289         40.311289         40.311289
9        45         40.049969         40.049969         40.049969
9        46         5.385165         5.385165         5.385165
9        47         3.000000         3.000000         3.000000
9        48         37.336309         37.336309         37.336309
9        49         79.711982         79.711982         79.711982
9        50         73.164199         73.164199         73.164199
9        51         72.622311         72.622311         72.622311
9        52         72.111026         72.111026         72.111026
9        53         50.990195         50.990195         50.990195
9        54         30.413813         30.413813         30.413813
9        55         44.721360         44.721360         44.721360
9        56         25.000000         25.000000         25.000000
9        57         57.008771         57.008771         57.008771
9        58         57.008771         57.008771         57.008771
9        59         70.000000         70.000000         70.000000
9        60         60.207973         60.207973         60.207973
9        61         20.000000         20.000000         20.000000
9        62         33.541020         33.541020         33.541020
9        63         67.268120         67.268120         67.268120
9        64         78.102497         78.102497         78.102497
9        65         58.309519         58.309519         58.309519
9        66         44.721360         44.721360         44.721360
9        67         50.249378         50.249378         50.249378
9        68         62.008064         62.008064         62.008064
9        69         32.015621         32.015621         32.015621
9        70         32.249031         32.249031         32.249031
9        71         22.825424         22.825424         22.825424
9        72         55.901699         55.901699         55.901699
9        73         50.289164         50.289164         50.289164
9        74         23.853721         23.853721         23.853721
9        75         60.207973         60.207973         60.207973
9        76         75.663730         75.663730         75.663730
9        77         81.541401         81.541401         81.541401
9        78         77.414469         77.414469         77.414469
9        79         25.000000         25.000000         25.000000
9        80         15.000000         15.000000         15.000000
9        81         45.967380         45.967380         45.967380
9        82         40.496913         40.496913         40.496913
9        83         38.897301         38.897301         38.897301
9        84         53.712196         53.712196         53.712196
9        85         66.068147         66.068147         66.068147
9        86         74.518454         74.518454         74.518454
9        87         56.320511         56.320511         56.320511
9        88         56.080300         56.080300         56.080300
9        89         23.769729         23.769729         23.769729
9        90         91.263355         91.263355         91.263355
9        91         39.661064         39.661064         39.661064
9        92         50.990195         50.990195         50.990195
9        93         53.037722         53.037722         53.037722
9        94         53.851648         53.851648         53.851648
9        95         52.801515         52.801515         52.801515
9        96         59.413803         59.413803         59.413803
9        97         47.707442         47.707442         47.707442
9        98         62.968246         62.968246         62.968246
9        99         30.083218         30.083218         30.083218
```

```
9       100      46.324939      46.324939      46.324939
9       101      20.615528      20.615528      20.615528
10        1     33.541020      33.541020      33.541020
10        2     52.201533      52.201533      52.201533
10        3     41.761226      41.761226      41.761226
10        4     51.419841      51.419841      51.419841
10        5     46.097722      46.097722      46.097722
10        6     50.990195      50.990195      50.990195
10        7     40.792156      40.792156      40.792156
10        8     40.311289      40.311289      40.311289
10        9     45.276926      45.276926      45.276926
10       11      5.000000      5.000000      5.000000
10       12      5.385165      5.385165      5.385165
10       13      10.198039      10.198039      10.198039
10       14      5.000000      5.000000      5.000000
10       15      11.180340      11.180340      11.180340
10       16      9.433981      9.433981      9.433981
10       17      11.180340      11.180340      11.180340
10       18      14.142136      14.142136      14.142136
10       19      45.343136      45.343136      45.343136
10       20      40.607881      40.607881      40.607881
10       21      37.735925      37.735925      37.735925
10       22      42.426407      42.426407      42.426407
10       23      36.055513      36.055513      36.055513
10       24      41.036569      41.036569      41.036569
10       25      34.409301      34.409301      34.409301
10       26      39.051248      39.051248      39.051248
10       27      85.146932      85.146932      85.146932
10       28      85.000000      85.000000      85.000000
10       29      82.152298      82.152298      82.152298
10       30      80.000000      80.000000      80.000000
10       31      78.160092      78.160092      78.160092
10       32      78.000000      78.000000      78.000000
10       33      77.162167      77.162167      77.162167
10       34      75.663730      75.663730      75.663730
10       35      75.000000      75.000000      75.000000
10       36      75.822160      75.822160      75.822160
10       37      74.330344      74.330344      74.330344
10       38      72.346389      72.346389      72.346389
10       39      68.767725      68.767725      68.767725
10       40      67.268120      67.268120      67.268120
10       41      70.710678      70.710678      70.710678
10       42      62.481997      62.481997      62.481997
10       43      63.639610      63.639610      63.639610
10       44      67.268120      67.268120      67.268120
10       45      65.069194      65.069194      65.069194
10       46      48.052055      48.052055      48.052055
10       47      45.705580      45.705580      45.705580
10       48      12.806248      12.806248      12.806248
10       49      43.863424      43.863424      43.863424
10       50      39.408121      39.408121      39.408121
10       51      62.000000      62.000000      62.000000
10       52      47.434165      47.434165      47.434165
10       53      15.811388      15.811388      15.811388
10       54      18.027756      18.027756      18.027756
10       55      51.478151      51.478151      51.478151
10       56      32.015621      32.015621      32.015621
10       57      40.000000      40.000000      40.000000
10       58      22.360680      22.360680      22.360680
10       59      25.495098      25.495098      25.495098
10       60      15.000000      15.000000      15.000000
10       61      25.495098      25.495098      25.495098
10       62      46.097722      46.097722      46.097722
```

```
10        63        55.000000        55.000000        55.000000
10        64        57.008771        57.008771        57.008771
10        65        35.355339        35.355339        35.355339
10        66        25.495098        25.495098        25.495098
10        67        31.064449        31.064449        31.064449
10        68        54.451814        54.451814        54.451814
10        69        39.051248        39.051248        39.051248
10        70        27.018512        27.018512        27.018512
10        71        42.201896        42.201896        42.201896
10        72        58.523500        58.523500        58.523500
10        73        60.901560        60.901560        60.901560
10        74        26.248809        26.248809        26.248809
10        75        18.027756        18.027756        18.027756
10        76        30.413813        30.413813        30.413813
10        77        55.036352        55.036352        55.036352
10        78        34.539832        34.539832        34.539832
10        79        21.095023        21.095023        21.095023
10        80        33.241540        33.241540        33.241540
10        81        38.897301        38.897301        38.897301
10        82        45.276926        45.276926        45.276926
10        83        18.788294        18.788294        18.788294
10        84        27.294688        27.294688        27.294688
10        85        47.381431        47.381431        47.381431
10        86        54.341513        54.341513        54.341513
10        87        15.556349        15.556349        15.556349
10        88        11.180340        11.180340        11.180340
10        89        26.925824        26.925824        26.925824
10        90        64.412732        64.412732        64.412732
10        91        29.546573        29.546573        29.546573
10        92        39.623226        39.623226        39.623226
10        93        43.737855        43.737855        43.737855
10        94        53.758720        53.758720        53.758720
10        95        48.764741        48.764741        48.764741
10        96        46.043458        46.043458        46.043458
10        97        48.846699        48.846699        48.846699
10        98        18.027756        18.027756        18.027756
10        99        23.345235        23.345235        23.345235
10        100        16.000000        16.000000        16.000000
10        101        38.275318        38.275318        38.275318
11        1        31.622777        31.622777        31.622777
11        2        47.434165        47.434165        47.434165
11        3        37.000000        37.000000        37.000000
11        4        46.572524        46.572524        46.572524
11        5        41.231056        41.231056        41.231056
11        6        46.097722        46.097722        46.097722
11        7        35.902646        35.902646        35.902646
11        8        35.355339        35.355339        35.355339
11        9        40.311289        40.311289        40.311289
11        10        5.000000        5.000000        5.000000
11        12        2.000000        2.000000        2.000000
11        13        5.385165        5.385165        5.385165
11        14        7.071068        7.071068        7.071068
11        15        7.071068        7.071068        7.071068
11        16        8.000000        8.000000        8.000000
11        17        10.000000        10.000000        10.000000
11        18        11.180340        11.180340        11.180340
11        19        48.795492        48.795492        48.795492
11        20        43.863424        43.863424        43.863424
11        21        40.607881        40.607881        40.607881
11        22        46.097722        46.097722        46.097722
11        23        39.051248        39.051248        39.051248
11        24        44.821870        44.821870        44.821870
11        25        37.536649        37.536649        37.536649
```

```
11        26        43.011626        43.011626        43.011626
11        27        85.586214        85.586214        85.586214
11        28        85.146932        85.146932        85.146932
11        29        82.607506        82.607506        82.607506
11        30        80.156098        80.156098        80.156098
11        31        78.638413        78.638413        78.638413
11        32        78.160092        78.160092        78.160092
11        33        77.646635        77.646635        77.646635
11        34        76.485293        76.485293        76.485293
11        35        75.166482        75.166482        75.166482
11        36        72.622311        72.622311        72.622311
11        37        71.063352        71.063352        71.063352
11        38        69.202601        69.202601        69.202601
11        39        65.604878        65.604878        65.604878
11        40        64.031242        64.031242        64.031242
11        41        67.268120        67.268120        67.268120
11        42        59.405387        59.405387        59.405387
11        43        60.207973        60.207973        60.207973
11        44        63.639610        63.639610        63.639610
11        45        61.554854        61.554854        61.554854
11        46        43.174066        43.174066        43.174066
11        47        40.792156        40.792156        40.792156
11        48        9.433981        9.433981        9.433981
11        49        47.423623        47.423623        47.423623
11        50        42.520583        42.520583        42.520583
11        51        62.201286        62.201286        62.201286
11        52        49.244289        49.244289        49.244289
11        53        18.027756        18.027756        18.027756
11        54        14.142136        14.142136        14.142136
11        55        49.244289        49.244289        49.244289
11        56        28.284271        28.284271        28.284271
11        57        40.311289        40.311289        40.311289
11        58        25.000000        25.000000        25.000000
11        59        30.413813        30.413813        30.413813
11        60        20.000000        20.000000        20.000000
11        61        20.615528        20.615528        20.615528
11        62        43.011626        43.011626        43.011626
11        63        55.226805        55.226805        55.226805
11        64        58.523500        58.523500        58.523500
11        65        36.400549        36.400549        36.400549
11        66        25.000000        25.000000        25.000000
11        67        31.144823        31.144823        31.144823
11        68        54.037024        54.037024        54.037024
11        69        36.055513        36.055513        36.055513
11        70        24.186773        24.186773        24.186773
11        71        38.288379        38.288379        38.288379
11        72        57.008771        57.008771        57.008771
11        73        58.600341        58.600341        58.600341
11        74        21.540659        21.540659        21.540659
11        75        22.360680        22.360680        22.360680
11        76        35.355339        35.355339        35.355339
11        77        57.306195        57.306195        57.306195
11        78        39.217343        39.217343        39.217343
11        79        16.124515        16.124515        16.124515
11        80        28.284271        28.284271        28.284271
11        81        37.656341        37.656341        37.656341
11        82        42.953463        42.953463        42.953463
11        83        17.262677        17.262677        17.262677
11        84        28.460499        28.460499        28.460499
11        85        48.270074        48.270074        48.270074
11        86        55.659680        55.659680        55.659680
11        87        19.416488        19.416488        19.416488
11        88        16.124515        16.124515        16.124515
```

```
11        89        22.803509        22.803509        22.803509
11        90        66.887966        66.887966        66.887966
11        91        27.892651        27.892651        27.892651
11        92        39.051248        39.051248        39.051248
11        93        43.104524        43.104524        43.104524
11        94        52.392748        52.392748        52.392748
11        95        47.675990        47.675990        47.675990
11        96        46.097722        46.097722        46.097722
11        97        47.127487        47.127487        47.127487
11        98        22.803509        22.803509        22.803509
11        99        20.000000        20.000000        20.000000
11        100        16.763055        16.763055        16.763055
11        101        34.205263        34.205263        34.205263
12        1        33.526109        33.526109        33.526109
12        2        48.104054        48.104054        48.104054
12        3        37.696154        37.696154        37.696154
12        4        47.127487        47.127487        47.127487
12        5        41.761226        41.761226        41.761226
12        6        46.572524        46.572524        46.572524
12        7        36.400549        36.400549        36.400549
12        8        35.693137        35.693137        35.693137
12        9        40.607881        40.607881        40.607881
12        10        5.385165        5.385165        5.385165
12        11        2.000000        2.000000        2.000000
12        13        5.000000        5.000000        5.000000
12        14        5.830952        5.830952        5.830952
12        15        5.830952        5.830952        5.830952
12        16        6.000000        6.000000        6.000000
12        17        8.000000        8.000000        8.000000
12        18        9.433981        9.433981        9.433981
12        19        50.209561        50.209561        50.209561
12        20        45.343136        45.343136        45.343136
12        21        42.201896        42.201896        42.201896
12        22        47.423623        47.423623        47.423623
12        23        40.607881        40.607881        40.607881
12        24        46.097722        46.097722        46.097722
12        25        39.051248        39.051248        39.051248
12        26        44.204072        44.204072        44.204072
12        27        87.572827        87.572827        87.572827
12        28        87.143560        87.143560        87.143560
12        29        84.593144        84.593144        84.593144
12        30        82.152298        82.152298        82.152298
12        31        80.622577        80.622577        80.622577
12        32        80.156098        80.156098        80.156098
12        33        79.630396        79.630396        79.630396
12        34        78.447435        78.447435        78.447435
12        35        77.162167        77.162167        77.162167
12        36        74.202426        74.202426        74.202426
12        37        72.622311        72.622311        72.622311
12        38        70.802542        70.802542        70.802542
12        39        67.201190        67.201190        67.201190
12        40        65.604878        65.604878        65.604878
12        41        68.767725        68.767725        68.767725
12        42        61.032778        61.032778        61.032778
12        43        61.717096        61.717096        61.717096
12        44        65.069194        65.069194        65.069194
12        45        63.031738        63.031738        63.031738
12        46        43.680659        43.680659        43.680659
12        47        41.231056        41.231056        41.231056
12        48        7.810250        7.810250        7.810250
12        49        48.795492        48.795492        48.795492
12        50        44.045431        44.045431        44.045431
12        51        64.195015        64.195015        64.195015
```

```
12         52         51.078371         51.078371         51.078371
12         53         19.723083         19.723083         19.723083
12         54         15.620499         15.620499         15.620499
12         55         51.078371         51.078371         51.078371
12         56         29.732137         29.732137         29.732137
12         57         42.296572         42.296572         42.296572
12         58         26.627054         26.627054         26.627054
12         59         30.805844         30.805844         30.805844
12         60         20.099751         20.099751         20.099751
12         61         21.189620         21.189620         21.189620
12         62         44.654227         44.654227         44.654227
12         63         57.218878         57.218878         57.218878
12         64         60.406953         60.406953         60.406953
12         65         38.327536         38.327536         38.327536
12         66         27.000000         27.000000         27.000000
12         67         33.136083         33.136083         33.136083
12         68         56.035703         56.035703         56.035703
12         69         37.735925         37.735925         37.735925
12         70         25.942244         25.942244         25.942244
12         71         39.623226         39.623226         39.623226
12         72         58.940648         58.940648         58.940648
12         73         60.415230         60.415230         60.415230
12         74         20.880613         20.880613         20.880613
12         75         23.323808         23.323808         23.323808
12         76         35.128336         35.128336         35.128336
12         77         59.059292         59.059292         59.059292
12         78         39.924930         39.924930         39.924930
12         79         16.000000         16.000000         16.000000
12         80         28.071338         28.071338         28.071338
12         81         39.623226         39.623226         39.623226
12         82         44.777226         44.777226         44.777226
12         83         19.235384         19.235384         19.235384
12         84         30.364453         30.364453         30.364453
12         85         50.219518         50.219518         50.219518
12         86         57.567352         57.567352         57.567352
12         87         20.615528         20.615528         20.615528
12         88         16.492423         16.492423         16.492423
12         89         24.083189         24.083189         24.083189
12         90         68.600292         68.600292         68.600292
12         91         29.832868         29.832868         29.832868
12         92         41.048752         41.048752         41.048752
12         93         45.099889         45.099889         45.099889
12         94         54.341513         54.341513         54.341513
12         95         49.648766         49.648766         49.648766
12         96         48.093659         48.093659         48.093659
12         97         49.040799         49.040799         49.040799
12         98         22.360680         22.360680         22.360680
12         99         21.633308         21.633308         21.633308
12         100         18.681542         18.681542         18.681542
12         101         35.468296         35.468296         35.468296
13         1         32.388269         32.388269         32.388269
13         2         43.462628         43.462628         43.462628
13         3         33.105891         33.105891         33.105891
13         4         42.379240         42.379240         42.379240
13         5         37.000000         37.000000         37.000000
13         6         41.761226         41.761226         41.761226
13         7         31.622777         31.622777         31.622777
13         8         30.805844         30.805844         30.805844
13         9         35.693137         35.693137         35.693137
13         10         10.198039         10.198039         10.198039
13         11         5.385165         5.385165         5.385165
13         12         5.000000         5.000000         5.000000
13         14         10.440307         10.440307         10.440307
```

```
13         15         3.000000         3.000000         3.000000
13         16         7.810250         7.810250         7.810250
13         17         9.433981         9.433981         9.433981
13         18         8.000000         8.000000         8.000000
13         19         53.814496         53.814496         53.814496
13         20         48.795492         48.795492         48.795492
13         21         45.343136         45.343136         45.343136
13         22         51.224994         51.224994         51.224994
13         23         43.863424         43.863424         43.863424
13         24         50.000000         50.000000         50.000000
13         25         42.426407         42.426407         42.426407
13         26         48.259714         48.259714         48.259714
13         27         88.283634         88.283634         88.283634
13         28         87.572827         87.572827         87.572827
13         29         85.328776         85.328776         85.328776
13         30         82.607506         82.607506         82.607506
13         31         81.394103         81.394103         81.394103
13         32         80.622577         80.622577         80.622577
13         33         80.411442         80.411442         80.411442
13         34         79.555012         79.555012         79.555012
13         35         77.646635         77.646635         77.646635
13         36         71.281134         71.281134         71.281134
13         37         69.634761         69.634761         69.634761
13         38         67.955868         67.955868         67.955868
13         39         64.350602         64.350602         64.350602
13         40         62.681736         62.681736         62.681736
13         41         65.604878         65.604878         65.604878
13         42         58.309519         58.309519         58.309519
13         43         58.600341         58.600341         58.600341
13         44         61.717096         61.717096         61.717096
13         45         59.816386         59.816386         59.816386
13         46         38.897301         38.897301         38.897301
13         47         36.400549         36.400549         36.400549
13         48         6.000000         6.000000         6.000000
13         49         52.497619         52.497619         52.497619
13         50         47.381431         47.381431         47.381431
13         51         64.776539         64.776539         64.776539
13         52         53.235327         53.235327         53.235327
13         53         22.671568         22.671568         22.671568
13         54         13.000000         13.000000         13.000000
13         55         49.335586         49.335586         49.335586
13         56         26.627054         26.627054         26.627054
13         57         43.174066         43.174066         43.174066
13         58         29.732137         29.732137         29.732137
13         59         35.693137         35.693137         35.693137
13         60         25.079872         25.079872         25.079872
13         61         16.552945         16.552945         16.552945
13         62         42.059482         42.059482         42.059482
13         63         57.870545         57.870545         57.870545
13         64         62.241465         62.241465         62.241465
13         65         39.924930         39.924930         39.924930
13         66         27.459060         27.459060         27.459060
13         67         33.955854         33.955854         33.955854
13         68         56.080300         56.080300         56.080300
13         69         35.341194         35.341194         35.341194
13         70         24.041631         24.041631         24.041631
13         71         36.124784         36.124784         36.124784
13         72         57.870545         57.870545         57.870545
13         73         58.523500         58.523500         58.523500
13         74         16.155494         16.155494         16.155494
13         75         27.730849         27.730849         27.730849
13         76         40.112342         40.112342         40.112342
13         77         61.587336         61.587336         61.587336
```

```
13        78        44.598206        44.598206        44.598206
13        79        11.000000        11.000000        11.000000
13        80        23.086793        23.086793        23.086793
13        81        39.051248        39.051248        39.051248
13        82        43.011626        43.011626        43.011626
13        83        19.104973        19.104973        19.104973
13        84        32.202484        32.202484        32.202484
13        85        51.546096        51.546096        51.546096
13        86        59.236813        59.236813        59.236813
13        87        24.698178        24.698178        24.698178
13        88        21.377558        21.377558        21.377558
13        89        20.615528        20.615528        20.615528
13        90        71.281134        71.281134        71.281134
13        91        29.068884        29.068884        29.068884
13        92        41.109610        41.109610        41.109610
13        93        45.044423        45.044423        45.044423
13        94        53.460266        53.460266        53.460266
13        95        49.091751        49.091751        49.091751
13        96        48.662100        48.662100        48.662100
13        97        47.853944        47.853944        47.853944
13        98        27.294688        27.294688        27.294688
13        99        19.313208        19.313208        19.313208
13        100        20.591260        20.591260        20.591260
13        101        31.827661        31.827661        31.827661
14        1        38.078866        38.078866        38.078866
14        2        53.851648        53.851648        53.851648
14        3        43.462628        43.462628        43.462628
14        4        52.810984        52.810984        52.810984
14        5        47.434165        47.434165        47.434165
14        6        52.201533        52.201533        52.201533
14        7        42.059482        42.059482        42.059482
14        8        41.231056        41.231056        41.231056
14        9        46.097722        46.097722        46.097722
14        10        5.000000        5.000000        5.000000
14        11        7.071068        7.071068        7.071068
14        12        5.830952        5.830952        5.830952
14        13        10.440307        10.440307        10.440307
14        15        10.000000        10.000000        10.000000
14        16        5.830952        5.830952        5.830952
14        17        7.071068        7.071068        7.071068
14        18        11.180340        11.180340        11.180340
14        19        49.203658        49.203658        49.203658
14        20        44.654227        44.654227        44.654227
14        21        42.059482        42.059482        42.059482
14        22        46.097722        46.097722        46.097722
14        23        40.311289        40.311289        40.311289
14        24        44.598206        44.598206        44.598206
14        25        38.587563        38.587563        38.587563
14        26        42.426407        42.426407        42.426407
14        27        90.138782        90.138782        90.138782
14        28        90.000000        90.000000        90.000000
14        29        87.143560        87.143560        87.143560
14        30        85.000000        85.000000        85.000000
14        31        83.150466        83.150466        83.150466
14        32        83.000000        83.000000        83.000000
14        33        82.152298        82.152298        82.152298
14        34        80.622577        80.622577        80.622577
14        35        80.000000        80.000000        80.000000
14        36        79.649231        79.649231        79.649231
14        37        78.102497        78.102497        78.102497
14        38        76.216796        76.216796        76.216796
14        39        72.622311        72.622311        72.622311
14        40        71.063352        71.063352        71.063352
```

```
14         41        74.330344        74.330344        74.330344
14         42        66.400301        66.400301        66.400301
14         43        67.268120        67.268120        67.268120
14         44        70.710678        70.710678        70.710678
14         45        68.622154        68.622154        68.622154
14         46        49.335586        49.335586        49.335586
14         47        46.840154        46.840154        46.840154
14         48        10.440307        10.440307        10.440307
14         49        47.634021        47.634021        47.634021
14         50        43.566042        43.566042        43.566042
14         51        67.000000        67.000000        67.000000
14         52        52.201533        52.201533        52.201533
14         53        20.615528        20.615528        20.615528
14         54        21.213203        21.213203        21.213203
14         55        55.901699        55.901699        55.901699
14         56        35.355339        35.355339        35.355339
14         57        45.000000        45.000000        45.000000
14         58        26.925824        26.925824        26.925824
14         59        26.925824        26.925824        26.925824
14         60        15.811388        15.811388        15.811388
14         61        26.925824        26.925824        26.925824
14         62        50.000000        50.000000        50.000000
14         63        60.000000        60.000000        60.000000
14         64        61.846584        61.846584        61.846584
14         65        40.311289        40.311289        40.311289
14         66        30.413813        30.413813        30.413813
14         67        36.055513        36.055513        36.055513
14         68        59.413803        59.413803        59.413803
14         69        43.011626        43.011626        43.011626
14         70        31.064449        31.064449        31.064449
14         71        45.343136        45.343136        45.343136
14         72        63.245553        63.245553        63.245553
14         73        65.299311        65.299311        65.299311
14         74        25.179357        25.179357        25.179357
14         75        21.213203        21.213203        21.213203
14         76        30.000000        30.000000        30.000000
14         77        59.615434        59.615434        59.615434
14         78        36.715120        36.715120        36.715120
14         79        21.213203        21.213203        21.213203
14         80        33.015148        33.015148        33.015148
14         81        43.680659        43.680659        43.680659
14         82        49.648766        49.648766        49.648766
14         83        23.409400        23.409400        23.409400
14         84        32.249031        32.249031        32.249031
14         85        52.345009        52.345009        52.345009
14         86        59.228372        59.228372        59.228372
14         87        19.416488        19.416488        19.416488
14         88        13.038405        13.038405        13.038405
14         89        29.832868        29.832868        29.832868
14         90        68.876701        68.876701        68.876701
14         91        34.176015        34.176015        34.176015
14         92        44.553339        44.553339        44.553339
14         93        48.662100        48.662100        48.662100
14         94        58.523500        58.523500        58.523500
14         95        53.600373        53.600373        53.600373
14         96        51.039201        51.039201        51.039201
14         97        53.488316        53.488316        53.488316
14         98        17.029386        17.029386        17.029386
14         99        27.018512        27.018512        27.018512
14         100        21.000000        21.000000        21.000000
14         101        41.231056        41.231056        41.231056
15         1        35.355339        35.355339        35.355339
15         2        44.721360        44.721360        44.721360
```

```
15          3         34.481879         34.481879         34.481879
15          4         43.462628         43.462628         43.462628
15          5         38.078866         38.078866         38.078866
15          6         42.720019         42.720019         42.720019
15          7         32.695565         32.695565         32.695565
15          8         31.622777         31.622777         31.622777
15          9         36.400549         36.400549         36.400549
15          10         11.180340         11.180340         11.180340
15          11         7.071068         7.071068         7.071068
15          12         5.830952         5.830952         5.830952
15          13         3.000000         3.000000         3.000000
15          14         10.000000         10.000000         10.000000
15          16         5.830952         5.830952         5.830952
15          17         7.071068         7.071068         7.071068
15          18         5.000000         5.000000         5.000000
15          19         55.865911         55.865911         55.865911
15          20         50.931326         50.931326         50.931326
15          21         47.634021         47.634021         47.634021
15          22         53.150729         53.150729         53.150729
15          23         46.097722         46.097722         46.097722
15          24         51.855569         51.855569         51.855569
15          25         44.598206         44.598206         44.598206
15          26         50.000000         50.000000         50.000000
15          27         91.241438         91.241438         91.241438
15          28         90.553851         90.553851         90.553851
15          29         88.283634         88.283634         88.283634
15          30         85.586214         85.586214         85.586214
15          31         84.344532         84.344532         84.344532
15          32         83.600239         83.600239         83.600239
15          33         83.360662         83.360662         83.360662
15          34         82.462113         82.462113         82.462113
15          35         80.622577         80.622577         80.622577
15          36         73.783467         73.783467         73.783467
15          37         72.111026         72.111026         72.111026
15          38         70.491134         70.491134         70.491134
15          39         66.887966         66.887966         66.887966
15          40         65.192024         65.192024         65.192024
15          41         68.007353         68.007353         68.007353
15          42         60.901560         60.901560         60.901560
15          43         61.032778         61.032778         61.032778
15          44         64.031242         64.031242         64.031242
15          45         62.201286         62.201286         62.201286
15          46         39.924930         39.924930         39.924930
15          47         37.336309         37.336309         37.336309
15          48         3.000000         3.000000         3.000000
15          49         54.488531         54.488531         54.488531
15          50         49.578221         49.578221         49.578221
15          51         67.742158         67.742158         67.742158
15          52         55.901699         55.901699         55.901699
15          53         25.000000         25.000000         25.000000
15          54         15.811388         15.811388         15.811388
15          55         52.201533         52.201533         52.201533
15          56         29.154759         29.154759         29.154759
15          57         46.097722         46.097722         46.097722
15          58         32.015621         32.015621         32.015621
15          59         36.400549         36.400549         36.400549
15          60         25.495098         25.495098         25.495098
15          61         18.027756         18.027756         18.027756
15          62         44.721360         44.721360         44.721360
15          63         60.827625         60.827625         60.827625
15          64         65.000000         65.000000         65.000000
15          65         42.720019         42.720019         42.720019
15          66         30.413813         30.413813         30.413813
```

```
15         67        36.878178        36.878178        36.878178
15         68        59.076222        59.076222        59.076222
15         69        38.078866        38.078866        38.078866
15         70        26.925824        26.925824        26.925824
15         71        38.418745        38.418745        38.418745
15         72        60.827625        60.827625        60.827625
15         73        61.351447        61.351447        61.351447
15         74        15.297059        15.297059        15.297059
15         75        29.154759        29.154759        29.154759
15         76        40.000000        40.000000        40.000000
15         77        64.140471        64.140471        64.140471
15         78        45.694639        45.694639        45.694639
15         79        11.401754        11.401754        11.401754
15         80        23.021729        23.021729        23.021729
15         81        42.047592        42.047592        42.047592
15         82        45.880279        45.880279        45.880279
15         83        22.090722        22.090722        22.090722
15         84        34.928498        34.928498        34.928498
15         85        54.405882        54.405882        54.405882
15         86        62.032250        62.032250        62.032250
15         87        26.400758        26.400758        26.400758
15         88        22.135944        22.135944        22.135944
15         89        23.021729        23.021729        23.021729
15         90        73.783467        73.783467        73.783467
15         91        32.062439        32.062439        32.062439
15         92        44.102154        44.102154        44.102154
15         93        48.041649        48.041649        48.041649
15         94        56.435804        56.435804        56.435804
15         95        52.086467        52.086467        52.086467
15         96        51.623638        51.623638        51.623638
15         97        50.803543        50.803543        50.803543
15         98        27.018512        27.018512        27.018512
15         99        22.135944        22.135944        22.135944
15         100        23.259407        23.259407        23.259407
15         101        34.058773        34.058773        34.058773
16         1        39.293765        39.293765        39.293765
16         2        50.537115        50.537115        50.537115
16         3        40.311289        40.311289        40.311289
16         4        49.244289        49.244289        49.244289
16         5        43.863424        43.863424        43.863424
16         6        48.466483        48.466483        48.466483
16         7        38.483763        38.483763        38.483763
16         8        37.336309        37.336309        37.336309
16         9        42.059482        42.059482        42.059482
16         10        9.433981        9.433981        9.433981
16         11        8.000000        8.000000        8.000000
16         12        6.000000        6.000000        6.000000
16         13        7.810250        7.810250        7.810250
16         14        5.830952        5.830952        5.830952
16         15        5.830952        5.830952        5.830952
16         17        2.000000        2.000000        2.000000
16         18        5.385165        5.385165        5.385165
16         19        54.671748        54.671748        54.671748
16         20        50.000000        50.000000        50.000000
16         21        47.169906        47.169906        47.169906
16         22        51.662365        51.662365        51.662365
16         23        45.486262        45.486262        45.486262
16         24        50.209561        50.209561        50.209561
16         25        43.829214        43.829214        43.829214
16         26        48.104054        48.104054        48.104054
16         27        93.536089        93.536089        93.536089
16         28        93.134312        93.134312        93.134312
16         29        90.553851        90.553851        90.553851
```

```
16         30         88.141931        88.141931        88.141931
16         31         86.579443        86.579443        86.579443
16         32         86.145226        86.145226        86.145226
16         33         85.586214        85.586214        85.586214
16         34         84.344532        84.344532        84.344532
16         35         83.150466        83.150466        83.150466
16         36         79.056942        79.056942        79.056942
16         37         77.420927        77.420927        77.420927
16         38         75.716577        75.716577        75.716577
16         39         72.111026        72.111026        72.111026
16         40         70.455660        70.455660        70.455660
16         41         73.409809        73.409809        73.409809
16         42         66.037868        66.037868        66.037868
16         43         66.400301        66.400301        66.400301
16         44         69.526973        69.526973        69.526973
16         45         67.623960        67.623960        67.623960
16         46         45.694639        45.694639        45.694639
16         47         43.081318        43.081318        43.081318
16         48         5.000000        5.000000        5.000000
16         49         53.150729        53.150729        53.150729
16         50         48.826222        48.826222        48.826222
16         51         70.178344        70.178344        70.178344
16         52         56.648036        56.648036        56.648036
16         53         25.079872        25.079872        25.079872
16         54         20.591260        20.591260        20.591260
16         55         56.648036        56.648036        56.648036
16         56         34.409301        34.409301        34.409301
16         57         48.259714        48.259714        48.259714
16         58         31.764760        31.764760        31.764760
16         59         32.695565        32.695565        32.695565
16         60         21.540659        21.540659        21.540659
16         61         23.853721        23.853721        23.853721
16         62         49.739320        49.739320        49.739320
16         63         63.198101        63.198101        63.198101
16         64         66.098411        66.098411        66.098411
16         65         44.147480        44.147480        44.147480
16         66         33.000000        33.000000        33.000000
16         67         39.115214        39.115214        39.115214
16         68         62.032250        62.032250        62.032250
16         69         42.941821        42.941821        42.941821
16         70         31.384710        31.384710        31.384710
16         71         43.931765        43.931765        43.931765
16         72         64.761099        64.761099        64.761099
16         73         65.924199        65.924199        65.924199
16         74         20.000000        20.000000        20.000000
16         75         26.907248        26.907248        26.907248
16         76         35.128336        35.128336        35.128336
16         77         64.404969        64.404969        64.404969
16         78         42.544095        42.544095        42.544095
16         79         17.088007        17.088007        17.088007
16         80         28.284271        28.284271        28.284271
16         81         45.541190        45.541190        45.541190
16         82         50.328918        50.328918        50.328918
16         83         25.179357        25.179357        25.179357
16         84         36.138622        36.138622        36.138622
16         85         56.089215        56.089215        56.089215
16         86         63.324561        63.324561        63.324561
16         87         24.839485        24.839485        24.839485
16         88         18.867962        18.867962        18.867962
16         89         28.425341        28.425341        28.425341
16         90         73.824115        73.824115        73.824115
16         91         35.693137        35.693137        35.693137
16         92         47.042534        47.042534        47.042534
```

```
16        93        51.088159        51.088159        51.088159
16        94        60.207973        60.207973        60.207973
16        95        55.578773        55.578773        55.578773
16        96        54.083269        54.083269        54.083269
16        97        54.817880        54.817880        54.817880
16        98        22.090722        22.090722        22.090722
16        99        26.832816        26.832816        26.832816
16        100        24.515301        24.515301        24.515301
16        101        39.623226        39.623226        39.623226
17        1        41.231056        41.231056        41.231056
17        2        51.478151        51.478151        51.478151
17        3        41.340053        41.340053        41.340053
17        4        50.089919        50.089919        50.089919
17        5        44.721360        44.721360        44.721360
17        6        49.244289        49.244289        49.244289
17        7        39.357337        39.357337        39.357337
17        8        38.078866        38.078866        38.078866
17        9        42.720019        42.720019        42.720019
17        10        11.180340        11.180340        11.180340
17        11        10.000000        10.000000        10.000000
17        12        8.000000        8.000000        8.000000
17        13        9.433981        9.433981        9.433981
17        14        7.071068        7.071068        7.071068
17        15        7.071068        7.071068        7.071068
17        16        2.000000        2.000000        2.000000
17        18        5.000000        5.000000        5.000000
17        19        56.222771        56.222771        56.222771
17        20        51.613952        51.613952        51.613952
17        21        48.877398        48.877398        48.877398
17        22        53.150729        53.150729        53.150729
17        23        47.169906        47.169906        47.169906
17        24        51.662365        51.662365        51.662365
17        25        45.486262        45.486262        45.486262
17        26        49.497475        49.497475        49.497475
17        27        95.524866        95.524866        95.524866
17        28        95.131488        95.131488        95.131488
17        29        92.541882        92.541882        92.541882
17        30        90.138782        90.138782        90.138782
17        31        88.566359        88.566359        88.566359
17        32        88.141931        88.141931        88.141931
17        33        87.572827        87.572827        87.572827
17        34        86.313383        86.313383        86.313383
17        35        85.146932        85.146932        85.146932
17        36        80.709355        80.709355        80.709355
17        37        79.056942        79.056942        79.056942
17        38        77.388630        77.388630        77.388630
17        39        73.783467        73.783467        73.783467
17        40        72.111026        72.111026        72.111026
17        41        75.000000        75.000000        75.000000
17        42        67.742158        67.742158        67.742158
17        43        68.007353        68.007353        68.007353
17        44        71.063352        71.063352        71.063352
17        45        69.202601        69.202601        69.202601
17        46        46.518813        46.518813        46.518813
17        47        43.863424        43.863424        43.863424
17        48        5.385165        5.385165        5.385165
17        49        54.671748        54.671748        54.671748
17        50        50.477718        50.477718        50.477718
17        51        72.173402        72.173402        72.173402
17        52        58.523500        58.523500        58.523500
17        53        26.925824        26.925824        26.925824
17        54        22.360680        22.360680        22.360680
17        55        58.523500        58.523500        58.523500
```

```
17        56        36.055513        36.055513        36.055513
17        57        50.249378        50.249378        50.249378
17        58        33.541020        33.541020        33.541020
17        59        33.541020        33.541020        33.541020
17        60        22.360680        22.360680        22.360680
17        61        25.000000        25.000000        25.000000
17        62        51.478151        51.478151        51.478151
17        63        65.192024        65.192024        65.192024
17        64        68.007353        68.007353        68.007353
17        65        46.097722        46.097722        46.097722
17        66        35.000000        35.000000        35.000000
17        67        41.109610        41.109610        41.109610
17        68        64.031242        64.031242        64.031242
17        69        44.721360        44.721360        44.721360
17        70        33.241540        33.241540        33.241540
17        71        45.453273        45.453273        45.453273
17        72        66.708320        66.708320        66.708320
17        73        67.779053        67.779053        67.779053
17        74        20.099751        20.099751        20.099751
17        75        28.284271        28.284271        28.284271
17        76        35.355339        35.355339        35.355339
17        77        66.211781        66.211781        66.211781
17        78        43.566042        43.566042        43.566042
17        79        17.888544        17.888544        17.888544
17        80        28.635642        28.635642        28.635642
17        81        47.518417        47.518417        47.518417
17        82        52.201533        52.201533        52.201533
17        83        27.166155        27.166155        27.166155
17        84        38.078866        38.078866        38.078866
17        85        58.051701        58.051701        58.051701
17        86        65.253352        65.253352        65.253352
17        87        26.400758        26.400758        26.400758
17        88        20.000000        20.000000        20.000000
17        89        30.000000        30.000000        30.000000
17        90        75.591005        75.591005        75.591005
17        91        37.656341        37.656341        37.656341
17        92        49.040799        49.040799        49.040799
17        93        53.084838        53.084838        53.084838
17        94        62.169124        62.169124        62.169124
17        95        57.558666        57.558666        57.558666
17        96        56.080300        56.080300        56.080300
17        97        56.753854        56.753854        56.753854
17        98        22.360680        22.360680        22.360680
17        99        28.635642        28.635642        28.635642
17        100        26.476405        26.476405        26.476405
17        101        41.109610        41.109610        41.109610
18        1        40.311289        40.311289        40.311289
18        2        47.169906        47.169906        47.169906
18        3        37.202150        37.202150        37.202150
18        4        45.650849        45.650849        45.650849
18        5        40.311289        40.311289        40.311289
18        6        44.721360        44.721360        44.721360
18        7        34.985711        34.985711        34.985711
18        8        33.541020        33.541020        33.541020
18        9        38.078866        38.078866        38.078866
18        10        14.142136        14.142136        14.142136
18        11        11.180340        11.180340        11.180340
18        12        9.433981        9.433981        9.433981
18        13        8.000000        8.000000        8.000000
18        14        11.180340        11.180340        11.180340
18        15        5.000000        5.000000        5.000000
18        16        5.385165        5.385165        5.385165
18        17        5.000000        5.000000        5.000000
```

```
18        19        59.464275        59.464275        59.464275
18        20        54.671748        54.671748        54.671748
18        21        51.613952        51.613952        51.613952
18        22        56.568542        56.568542        56.568542
18        23        50.000000        50.000000        50.000000
18        24        55.172457        55.172457        55.172457
18        25        48.414874        48.414874        48.414874
18        26        53.150729        53.150729        53.150729
18        27        96.176920        96.176920        96.176920
18        28        95.524866        95.524866        95.524866
18        29        93.214806        93.214806        93.214806
18        30        90.553851        90.553851        90.553851
18        31        89.269256        89.269256        89.269256
18        32        88.566359        88.566359        88.566359
18        33        88.283634        88.283634        88.283634
18        34        87.321246        87.321246        87.321246
18        35        85.586214        85.586214        85.586214
18        36        78.032045        78.032045        78.032045
18        37        76.321688        76.321688        76.321688
18        38        74.793048        74.793048        74.793048
18        39        71.196910        71.196910        71.196910
18        40        69.462220        69.462220        69.462220
18        41        72.111026        72.111026        72.111026
18        42        65.299311        65.299311        65.299311
18        43        65.192024        65.192024        65.192024
18        44        68.007353        68.007353        68.007353
18        45        66.287254        66.287254        66.287254
18        46        42.059482        42.059482        42.059482
18        47        39.357337        39.357337        39.357337
18        48        2.000000        2.000000        2.000000
18        49        58.000000        58.000000        58.000000
18        50        53.413481        53.413481        53.413481
18        51        72.691127        72.691127        72.691127
18        52        60.415230        60.415230        60.415230
18        53        29.154759        29.154759        29.154759
18        54        20.615528        20.615528        20.615528
18        55        57.008771        57.008771        57.008771
18        56        33.541020        33.541020        33.541020
18        57        50.990195        50.990195        50.990195
18        58        36.055513        36.055513        36.055513
18        59        38.078866        38.078866        38.078866
18        60        26.925824        26.925824        26.925824
18        61        21.213203        21.213203        21.213203
18        62        49.244289        49.244289        49.244289
18        63        65.764732        65.764732        65.764732
18        64        69.641941        69.641941        69.641941
18        65        47.434165        47.434165        47.434165
18        66        35.355339        35.355339        35.355339
18        67        41.773197        41.773197        41.773197
18        68        64.070274        64.070274        64.070274
18        69        42.720019        42.720019        42.720019
18        70        31.780497        31.780497        31.780497
18        71        42.438190        42.438190        42.438190
18        72        65.764732        65.764732        65.764732
18        73        66.098411        66.098411        66.098411
18        74        15.132746        15.132746        15.132746
18        75        32.015621        32.015621        32.015621
18        76        40.311289        40.311289        40.311289
18        77        68.476273        68.476273        68.476273
18        78        47.885280        47.885280        47.885280
18        79        13.601471        13.601471        13.601471
18        80        23.769729        23.769729        23.769729
18        81        47.042534        47.042534        47.042534
```

```
18        82        50.695167        50.695167        50.695167
18        83        27.073973        27.073973        27.073973
18        84        39.560081        39.560081        39.560081
18        85        59.203040        59.203040        59.203040
18        86        66.730802        66.730802        66.730802
18        87        29.698485        29.698485        29.698485
18        88        24.186773        24.186773        24.186773
18        89        27.294688        27.294688        27.294688
18        90        78.032045        78.032045        78.032045
18        91        37.054015        37.054015        37.054015
18        92        49.091751        49.091751        49.091751
18        93        53.037722        53.037722        53.037722
18        94        61.400326        61.400326        61.400326
18        95        57.078893        57.078893        57.078893
18        96        56.568542        56.568542        56.568542
18        97        55.731499        55.731499        55.731499
18        98        27.294688        27.294688        27.294688
18        99        26.925824        26.925824        26.925824
18        100        27.856777        27.856777        27.856777
18        101        38.013156        38.013156        38.013156
19        1        45.177428        45.177428        45.177428
19        2        82.225300        82.225300        82.225300
19        3        73.375745        73.375745        73.375745
19        4        82.969874        82.969874        82.969874
19        5        78.746428        78.746428        78.746428
19        6        83.522452        83.522452        83.522452
19        7        74.672619        74.672619        74.672619
19        8        75.769387        75.769387        75.769387
19        9        80.411442        80.411442        80.411442
19        10        45.343136        45.343136        45.343136
19        11        48.795492        48.795492        48.795492
19        12        50.209561        50.209561        50.209561
19        13        53.814496        53.814496        53.814496
19        14        49.203658        49.203658        49.203658
19        15        55.865911        55.865911        55.865911
19        16        54.671748        54.671748        54.671748
19        17        56.222771        56.222771        56.222771
19        18        59.464275        59.464275        59.464275
19        20        5.385165        5.385165        5.385165
19        21        10.198039        10.198039        10.198039
19        22        4.000000        4.000000        4.000000
19        23        10.770330        10.770330        10.770330
19        24        6.000000        6.000000        6.000000
19        25        11.661904        11.661904        11.661904
19        26        9.000000        9.000000        9.000000
19        27        56.797887        56.797887        56.797887
19        28        59.169249        59.169249        59.169249
19        29        54.120237        54.120237        54.120237
19        30        54.918121        54.918121        54.918121
19        31        50.606324        50.606324        50.606324
19        32        53.254108        53.254108        53.254108
19        33        49.739320        49.739320        49.739320
19        34        45.617979        45.617979        45.617979
19        35        50.803543        50.803543        50.803543
19        36        83.240615        83.240615        83.240615
19        37        82.710338        82.710338        82.710338
19        38        79.812280        79.812280        79.812280
19        39        77.129761        77.129761        77.129761
19        40        76.687678        76.687678        76.687678
19        41        81.584312        81.584312        81.584312
19        42        71.386273        71.386273        71.386273
19        43        75.802375        75.802375        75.802375
19        44        80.752709        80.752709        80.752709
```

```
19        45        77.781746        77.781746        77.781746
19        46        80.653580        80.653580        80.653580
19        47        79.378838        79.378838        79.378838
19        48        58.000000        58.000000        58.000000
19        49        2.000000        2.000000        2.000000
19        50        7.280110        7.280110        7.280110
19        51        41.036569        41.036569        41.036569
19        52        18.601075        18.601075        18.601075
19        53        31.400637        31.400637        31.400637
19        54        51.000000        51.000000        51.000000
19        55        56.089215        56.089215        56.089215
19        56        56.753854        56.753854        56.753854
19        57        30.594117        30.594117        30.594117
19        58        24.413111        24.413111        24.413111
19        59        29.427878        29.427878        29.427878
19        60        37.161808        37.161808        37.161808
19        61        62.177166        62.177166        62.177166
19        62        60.008333        60.008333        60.008333
19        63        36.619667        36.619667        36.619667
19        64        25.806976        25.806976        25.806976
19        65        25.019992        25.019992        25.019992
19        66        36.138622        36.138622        36.138622
19        67        32.140317        32.140317        32.140317
19        68        42.059482        42.059482        42.059482
19        69        55.145263        55.145263        55.145263
19        70        48.764741        48.764741        48.764741
19        71        64.629715        64.629715        64.629715
19        72        54.230987        54.230987        54.230987
19        73        62.936476        62.936476        62.936476
19        74        69.202601        69.202601        69.202601
19        75        28.301943        28.301943        28.301943
19        76        39.000000        39.000000        39.000000
19        77        17.464249        17.464249        17.464249
19        78        21.095023        21.095023        21.095023
19        79        62.425956        62.425956        62.425956
19        80        73.573093        73.573093        73.573093
19        81        42.107007        42.107007        42.107007
19        82        53.235327        53.235327        53.235327
19        83        41.629317        41.629317        41.629317
19        84        26.925824        26.925824        26.925824
19        85        27.294688        27.294688        27.294688
19        86        26.172505        26.172505        26.172505
19        87        29.832868        29.832868        29.832868
19        88        37.215588        37.215588        37.215588
19        89        56.648036        56.648036        56.648036
19        90        23.000000        23.000000        23.000000
19        91        42.579338        42.579338        42.579338
19        92        37.336309        37.336309        37.336309
19        93        39.051248        39.051248        39.051248
19        94        49.979996        49.979996        49.979996
19        95        44.922155        44.922155        44.922155
19        96        34.176015        34.176015        34.176015
19        97        50.219518        50.219518        50.219518
19        98        42.059482        42.059482        42.059482
19        99        50.328918        50.328918        50.328918
19        100        34.985711        34.985711        34.985711
19        101        63.348244        63.348244        63.348244
20        1        40.049969        40.049969        40.049969
20        2        76.902536        76.902536        76.902536
20        3        68.007353        68.007353        68.007353
20        4        77.620873        77.620873        77.620873
20        5        73.375745        73.375745        73.375745
20        6        78.160092        78.160092        78.160092
```

```
20        7        69.289249        69.289249        69.289249
20        8        70.384657        70.384657        70.384657
20        9        75.026662        75.026662        75.026662
20        10        40.607881        40.607881        40.607881
20        11        43.863424        43.863424        43.863424
20        12        45.343136        45.343136        45.343136
20        13        48.795492        48.795492        48.795492
20        14        44.654227        44.654227        44.654227
20        15        50.931326        50.931326        50.931326
20        16        50.000000        50.000000        50.000000
20        17        51.613952        51.613952        51.613952
20        18        54.671748        54.671748        54.671748
20        19        5.385165        5.385165        5.385165
20        21        5.000000        5.000000        5.000000
20        22        5.385165        5.385165        5.385165
20        23        5.385165        5.385165        5.385165
20        24        6.403124        6.403124        6.403124
20        25        6.403124        6.403124        6.403124
20        26        8.602325        8.602325        8.602325
20        27        56.648036        56.648036        56.648036
20        28        58.600341        58.600341        58.600341
20        29        53.851648        53.851648        53.851648
20        30        54.120237        54.120237        54.120237
20        31        50.159745        50.159745        50.159745
20        32        52.354560        52.354560        52.354560
20        33        49.244289        49.244289        49.244289
20        34        45.541190        45.541190        45.541190
20        35        49.739320        49.739320        49.739320
20        36        79.056942        79.056942        79.056942
20        37        78.447435        78.447435        78.447435
20        38        75.584390        75.584390        75.584390
20        39        72.801099        72.801099        72.801099
20        40        72.277244        72.277244        72.277244
20        41        77.129761        77.129761        77.129761
20        42        66.940272        66.940272        66.940272
20        43        71.196910        71.196910        71.196910
20        44        76.118329        76.118329        76.118329
20        45        73.164199        73.164199        73.164199
20        46        75.286121        75.286121        75.286121
20        47        74.000000        74.000000        74.000000
20        48        53.150729        53.150729        53.150729
20        49        5.000000        5.000000        5.000000
20        50        2.000000        2.000000        2.000000
20        51        39.051248        39.051248        39.051248
20        52        16.401219        16.401219        16.401219
20        53        26.248809        26.248809        26.248809
20        54        45.650849        45.650849        45.650849
20        55        51.662365        51.662365        51.662365
20        56        51.419841        51.419841        51.419841
20        57        26.248809        26.248809        26.248809
20        58        19.209373        19.209373        19.209373
20        59        27.000000        27.000000        27.000000
20        60        33.526109        33.526109        33.526109
20        61        56.824291        56.824291        56.824291
20        62        55.081757        55.081757        55.081757
20        63        33.970576        33.970576        33.970576
20        64        25.079872        25.079872        25.079872
20        65        20.223748        20.223748        20.223748
20        66        30.805844        30.805844        30.805844
20        67        27.018512        27.018512        27.018512
20        68        38.832976        38.832976        38.832976
20        69        50.039984        50.039984        50.039984
20        70        43.416587        43.416587        43.416587
```

```
20        71        59.413803        59.413803        59.413803
20        72        50.537115        50.537115        50.537115
20        73        58.872744        58.872744        58.872744
20        74        64.031242        64.031242        64.031242
20        75        24.166092        24.166092        24.166092
20        76        37.336309        37.336309        37.336309
20        77        18.110770        18.110770        18.110770
20        78        20.248457        20.248457        20.248457
20        79        57.201399        57.201399        57.201399
20        80        68.264193        68.264193        68.264193
20        81        37.336309        37.336309        37.336309
20        82        48.507731        48.507731        48.507731
20        83        36.249138        36.249138        36.249138
20        84        21.587033        21.587033        21.587033
20        85        24.207437        24.207437        24.207437
20        86        24.698178        24.698178        24.698178
20        87        25.238859        25.238859        25.238859
20        88        33.105891        33.105891        33.105891
20        89        51.264022        51.264022        51.264022
20        90        25.495098        25.495098        25.495098
20        91        37.336309        37.336309        37.336309
20        92        32.756679        32.756679        32.756679
20        93        34.785054        34.785054        34.785054
20        94        46.097722        46.097722        46.097722
20        95        40.853396        40.853396        40.853396
20        96        30.413813        30.413813        30.413813
20        97        45.880279        45.880279        45.880279
20        98        38.832976        38.832976        38.832976
20        99        44.944410        44.944410        44.944410
20        100        29.681644        29.681644        29.681644
20        101        58.051701        58.051701        58.051701
21        1        35.057096        35.057096        35.057096
21        2        72.034714        72.034714        72.034714
21        3        63.245553        63.245553        63.245553
21        4        72.801099        72.801099        72.801099
21        5        68.622154        68.622154        68.622154
21        6        73.375745        73.375745        73.375745
21        7        64.621978        64.621978        64.621978
21        8        65.795137        65.795137        65.795137
21        9        70.384657        70.384657        70.384657
21        10        37.735925        37.735925        37.735925
21        11        40.607881        40.607881        40.607881
21        12        42.201896        42.201896        42.201896
21        13        45.343136        45.343136        45.343136
21        14        42.059482        42.059482        42.059482
21        15        47.634021        47.634021        47.634021
21        16        47.169906        47.169906        47.169906
21        17        48.877398        48.877398        48.877398
21        18        51.613952        51.613952        51.613952
21        19        10.198039        10.198039        10.198039
21        20        5.000000        5.000000        5.000000
21        22        10.198039        10.198039        10.198039
21        23        2.000000        2.000000        2.000000
21        24        10.770330        10.770330        10.770330
21        25        4.000000        4.000000        4.000000
21        26        12.206556        12.206556        12.206556
21        27        55.081757        55.081757        55.081757
21        28        56.648036        56.648036        56.648036
21        29        52.201533        52.201533        52.201533
21        30        52.000000        52.000000        52.000000
21        31        48.383882        48.383882        48.383882
21        32        50.159745        50.159745        50.159745
21        33        47.434165        47.434165        47.434165
```

```
21        34        44.147480        44.147480        44.147480
21        35        47.423623        47.423623        47.423623
21        36        74.330344        74.330344        74.330344
21        37        73.681748        73.681748        73.681748
21        38        70.837843        70.837843        70.837843
21        39        68.007353        68.007353        68.007353
21        40        67.446275        67.446275        67.446275
21        41        72.277244        72.277244        72.277244
21        42        62.096699        62.096699        62.096699
21        43        66.287254        66.287254        66.287254
21        44        71.196910        71.196910        71.196910
21        45        68.249542        68.249542        68.249542
21        46        70.519501        70.519501        70.519501
21        47        69.289249        69.289249        69.289249
21        48        50.000000        50.000000        50.000000
21        49        10.000000        10.000000        10.000000
21        50        3.000000        3.000000        3.000000
21        51        36.055513        36.055513        36.055513
21        52        13.928388        13.928388        13.928388
21        53        22.671568        22.671568        22.671568
21        54        41.340053        41.340053        41.340053
21        55        46.840154        46.840154        46.840154
21        56        46.572524        46.572524        46.572524
21        57        21.540659        21.540659        21.540659
21        58        15.620499        15.620499        15.620499
21        59        27.459060        27.459060        27.459060
21        60        32.388269        32.388269        32.388269
21        61        52.478567        52.478567        52.478567
21        62        50.089919        50.089919        50.089919
21        63        30.479501        30.479501        30.479501
21        64        23.537205        23.537205        23.537205
21        65        15.297059        15.297059        15.297059
21        66        25.961510        25.961510        25.961510
21        67        22.022716        22.022716        22.022716
21        68        34.828150        34.828150        34.828150
21        69        45.044423        45.044423        45.044423
21        70        38.600518        38.600518        38.600518
21        71        54.451814        54.451814        54.451814
21        72        46.141088        46.141088        46.141088
21        73        54.230987        54.230987        54.230987
21        74        60.207973        60.207973        60.207973
21        75        22.561028        22.561028        22.561028
21        76        38.327536        38.327536        38.327536
21        77        18.248288        18.248288        18.248288
21        78        22.472205        22.472205        22.472205
21        79        53.263496        53.263496        53.263496
21        80        64.070274        64.070274        64.070274
21        81        32.388269        32.388269        32.388269
21        82        43.566042        43.566042        43.566042
21        83        31.764760        31.764760        31.764760
21        84        16.763055        16.763055        16.763055
21        85        20.518285        20.518285        20.518285
21        86        22.472205        22.472205        22.472205
21        87        22.847319        22.847319        22.847319
21        88        31.320920        31.320920        31.320920
21        89        46.615448        46.615448        46.615448
21        90        26.925824        26.925824        26.925824
21        91        32.388269        32.388269        32.388269
21        92        27.892651        27.892651        27.892651
21        93        30.083218        30.083218        30.083218
21        94        41.593269        41.593269        41.593269
21        95        36.249138        36.249138        36.249138
21        96        26.076810        26.076810        26.076810
```

```
21        97        41.109610        41.109610        41.109610
21        98        38.118237        38.118237        38.118237
21        99        40.311289        40.311289        40.311289
21        100        25.612497        25.612497        25.612497
21        101        53.150729        53.150729        53.150729
22        1       45.000000        45.000000        45.000000
22        2       81.394103        81.394103        81.394103
22        3       72.277244        72.277244        72.277244
22        4       82.000000        82.000000        82.000000
22        5       77.620873        77.620873        77.620873
22        6       82.462113        82.462113        82.462113
22        7       73.375745        73.375745        73.375745
22        8       74.330344        74.330344        74.330344
22        9       79.056942        79.056942        79.056942
22        10        42.426407        42.426407        42.426407
22        11        46.097722        46.097722        46.097722
22        12        47.423623        47.423623        47.423623
22        13        51.224994        51.224994        51.224994
22        14        46.097722        46.097722        46.097722
22        15        53.150729        53.150729        53.150729
22        16        51.662365        51.662365        51.662365
22        17        53.150729        53.150729        53.150729
22        18        56.568542        56.568542        56.568542
22        19        4.000000        4.000000        4.000000
22        20        5.385165        5.385165        5.385165
22        21        10.198039        10.198039        10.198039
22        23        10.000000        10.000000        10.000000
22        24        2.000000        2.000000        2.000000
22        25        10.198039        10.198039        10.198039
22        26        5.000000        5.000000        5.000000
22        27        60.415230        60.415230        60.415230
22        28        62.649820        62.649820        62.649820
22        29        57.697487        57.697487        57.697487
22        30        58.309519        58.309519        58.309519
22        31        54.120237        54.120237        54.120237
22        32        56.603887        56.603887        56.603887
22        33        53.235327        53.235327        53.235327
22        34        49.244289        49.244289        49.244289
22        35        54.083269        54.083269        54.083269
22        36        84.433406        84.433406        84.433406
22        37        83.815273        83.815273        83.815273
22        38        80.956779        80.956779        80.956779
22        39        78.160092        78.160092        78.160092
22        40        77.620873        77.620873        77.620873
22        41        82.462113        82.462113        82.462113
22        42        72.277244        72.277244        72.277244
22        43        76.485293        76.485293        76.485293
22        44        81.394103        81.394103        81.394103
22        45        78.447435        78.447435        78.447435
22        46        79.555012        79.555012        79.555012
22        47        78.160092        78.160092        78.160092
22        48        55.172457        55.172457        55.172457
22        49        2.000000        2.000000        2.000000
22        50        7.280110        7.280110        7.280110
22        51        43.863424        43.863424        43.863424
22        52        21.213203        21.213203        21.213203
22        53        29.154759        29.154759        29.154759
22        54        49.244289        49.244289        49.244289
22        55        57.008771        57.008771        57.008771
22        56        55.901699        55.901699        55.901699
22        57        31.622777        31.622777        31.622777
22        58        22.360680        22.360680        22.360680
22        59        25.495098        25.495098        25.495098
```

```
22         60        33.541020        33.541020        33.541020
22         61        60.415230        60.415230        60.415230
22         62        60.207973        60.207973        60.207973
22         63        39.051248        39.051248        39.051248
22         64        29.154759        29.154759        29.154759
22         65        25.495098        25.495098        25.495098
22         66        35.355339        35.355339        35.355339
22         67        32.015621        32.015621        32.015621
22         68        44.102154        44.102154        44.102154
22         69        55.000000        55.000000        55.000000
22         70        47.853944        47.853944        47.853944
22         71        64.195015        64.195015        64.195015
22         72        55.901699        55.901699        55.901699
22         73        64.257295        64.257295        64.257295
22         74        66.850580        66.850580        66.850580
22         75        25.000000        25.000000        25.000000
22         76        35.000000        35.000000        35.000000
22         77        21.189620        21.189620        21.189620
22         78        17.117243        17.117243        17.117243
22         79        60.207973        60.207973        60.207973
22         80        71.589105        71.589105        71.589105
22         81        42.579338        42.579338        42.579338
22         82        53.758720        53.758720        53.758720
22         83        40.162171        40.162171        40.162171
22         84        26.172505        26.172505        26.172505
22         85        29.410882        29.410882        29.410882
22         86        29.206164        29.206164        29.206164
22         87        26.870058        26.870058        26.870058
22         88        33.837849        33.837849        33.837849
22         89        55.362442        55.362442        55.362442
22         90        27.000000        27.000000        27.000000
22         91        42.107007        42.107007        42.107007
22         92        38.078866        38.078866        38.078866
22         93        40.162171        40.162171        40.162171
22         94        51.478151        51.478151        51.478151
22         95        46.238512        46.238512        46.238512
22         96        35.777088        35.777088        35.777088
22         97        51.244512        51.244512        51.244512
22         98        38.275318        38.275318        38.275318
22         99        49.040799        49.040799        49.040799
22         100        33.105891        33.105891        33.105891
22         101        62.649820        62.649820        62.649820
23         1        35.000000        35.000000        35.000000
23         2        71.589105        71.589105        71.589105
23         3        62.641839        62.641839        62.641839
23         4        72.277244        72.277244        72.277244
23         5        68.007353        68.007353        68.007353
23         6        72.801099        72.801099        72.801099
23         7        63.906181        63.906181        63.906181
23         8        65.000000        65.000000        65.000000
23         9        69.641941        69.641941        69.641941
23         10        36.055513        36.055513        36.055513
23         11        39.051248        39.051248        39.051248
23         12        40.607881        40.607881        40.607881
23         13        43.863424        43.863424        43.863424
23         14        40.311289        40.311289        40.311289
23         15        46.097722        46.097722        46.097722
23         16        45.486262        45.486262        45.486262
23         17        47.169906        47.169906        47.169906
23         18        50.000000        50.000000        50.000000
23         19        10.770330        10.770330        10.770330
23         20        5.385165        5.385165        5.385165
23         21        2.000000        2.000000        2.000000
```

```
23        22        10.000000        10.000000        10.000000
23        24        10.198039        10.198039        10.198039
23        25        2.000000        2.000000        2.000000
23        26        11.180340        11.180340        11.180340
23        27        57.008771        57.008771        57.008771
23        28        58.523500        58.523500        58.523500
23        29        54.120237        54.120237        54.120237
23        30        53.851648        53.851648        53.851648
23        31        50.289164        50.289164        50.289164
23        32        52.000000        52.000000        52.000000
23        33        49.335586        49.335586        49.335586
23        34        46.097722        46.097722        46.097722
23        35        49.244289        49.244289        49.244289
23        36        75.026662        75.026662        75.026662
23        37        74.330344        74.330344        74.330344
23        38        71.512237        71.512237        71.512237
23        39        68.622154        68.622154        68.622154
23        40        68.007353        68.007353        68.007353
23        41        72.801099        72.801099        72.801099
23        42        62.641839        62.641839        62.641839
23        43        66.708320        66.708320        66.708320
23        44        71.589105        71.589105        71.589105
23        45        68.658576        68.658576        68.658576
23        46        69.921384        69.921384        69.921384
23        47        68.622154        68.622154        68.622154
23        48        48.414874        48.414874        48.414874
23        49        10.198039        10.198039        10.198039
23        50        3.605551        3.605551        3.605551
23        51        37.735925        37.735925        37.735925
23        52        15.811388        15.811388        15.811388
23        53        21.213203        21.213203        21.213203
23        54        40.311289        40.311289        40.311289
23        55        47.434165        47.434165        47.434165
23        56        46.097722        46.097722        46.097722
23        57        22.360680        22.360680        22.360680
23        58        14.142136        14.142136        14.142136
23        59        25.495098        25.495098        25.495098
23        60        30.413813        30.413813        30.413813
23        61        51.478151        51.478151        51.478151
23        62        50.249378        50.249378        50.249378
23        63        32.015621        32.015621        32.015621
23        64        25.495098        25.495098        25.495098
23        65        15.811388        15.811388        15.811388
23        66        25.495098        25.495098        25.495098
23        67        22.022716        22.022716        22.022716
23        68        36.124784        36.124784        36.124784
23        69        45.000000        45.000000        45.000000
23        70        38.078866        38.078866        38.078866
23        71        54.230987        54.230987        54.230987
23        72        47.169906        47.169906        47.169906
23        73        55.036352        55.036352        55.036352
23        74        58.898217        58.898217        58.898217
23        75        20.615528        20.615528        20.615528
23        76        36.400549        36.400549        36.400549
23        77        20.223748        20.223748        20.223748
23        78        20.808652        20.808652        20.808652
23        79        52.009614        52.009614        52.009614
23        80        62.968246        62.968246        62.968246
23        81        32.756679        32.756679        32.756679
23        82        43.931765        43.931765        43.931765
23        83        30.870698        30.870698        30.870698
23        84        16.278821        16.278821        16.278821
23        85        22.022716        22.022716        22.022716
```

```
23        86        24.351591        24.351591        24.351591
23        87        21.023796        21.023796        21.023796
23        88        29.410882        29.410882        29.410882
23        89        45.880279        45.880279        45.880279
23        90        28.792360        28.792360        28.792360
23        91        32.140317        32.140317        32.140317
23        92        28.460499        28.460499        28.460499
23        93        30.870698        30.870698        30.870698
23        94        42.544095        42.544095        42.544095
23        95        37.121422        37.121422        37.121422
23        96        27.202941        27.202941        27.202941
23        97        41.785165        41.785165        41.785165
23        98        36.124784        36.124784        36.124784
23        99        39.560081        39.560081        39.560081
23        100        24.413111        24.413111        24.413111
23        101        52.773099        52.773099        52.773099
24        1        45.044423        45.044423        45.044423
24        2        81.049368        81.049368        81.049368
24        3        71.805292        71.805292        71.805292
24        4        81.584312        81.584312        81.584312
24        5        77.129761        77.129761        77.129761
24        6        82.000000        82.000000        82.000000
24        7        72.801099        72.801099        72.801099
24        8        73.681748        73.681748        73.681748
24        9        78.447435        78.447435        78.447435
24        10        41.036569        41.036569        41.036569
24        11        44.821870        44.821870        44.821870
24        12        46.097722        46.097722        46.097722
24        13        50.000000        50.000000        50.000000
24        14        44.598206        44.598206        44.598206
24        15        51.855569        51.855569        51.855569
24        16        50.209561        50.209561        50.209561
24        17        51.662365        51.662365        51.662365
24        18        55.172457        55.172457        55.172457
24        19        6.000000        6.000000        6.000000
24        20        6.403124        6.403124        6.403124
24        21        10.770330        10.770330        10.770330
24        22        2.000000        2.000000        2.000000
24        23        10.198039        10.198039        10.198039
24        25        10.000000        10.000000        10.000000
24        26        3.000000        3.000000        3.000000
24        27        62.241465        62.241465        62.241465
24        28        64.412732        64.412732        64.412732
24        29        59.506302        59.506302        59.506302
24        30        60.033324        60.033324        60.033324
24        31        55.901699        55.901699        55.901699
24        32        58.309519        58.309519        58.309519
24        33        55.009090        55.009090        55.009090
24        34        51.078371        51.078371        51.078371
24        35        55.758407        55.758407        55.758407
24        36        85.094066        85.094066        85.094066
24        37        84.433406        84.433406        84.433406
24        38        81.596569        81.596569        81.596569
24        39        78.746428        78.746428        78.746428
24        40        78.160092        78.160092        78.160092
24        41        82.969874        82.969874        82.969874
24        42        72.801099        72.801099        72.801099
24        43        76.902536        76.902536        76.902536
24        44        81.786307        81.786307        81.786307
24        45        78.854296        78.854296        78.854296
24        46        79.075913        79.075913        79.075913
24        47        77.620873        77.620873        77.620873
24        48        53.814496        53.814496        53.814496
```

```
24        49       4.000000        4.000000        4.000000
24        50       8.062258        8.062258        8.062258
24        51       45.343136       45.343136       45.343136
24        52       22.671568       22.671568       22.671568
24        53       28.178006       28.178006       28.178006
24        54       48.466483       48.466483       48.466483
24        55       57.567352       57.567352       57.567352
24        56       55.578773       55.578773       55.578773
24        57       32.310989       32.310989       32.310989
24        58       21.540659       21.540659       21.540659
24        59       23.537205       23.537205       23.537205
24        60       31.764760       31.764760       31.764760
24        61       59.615434       59.615434       59.615434
24        62       60.406953       60.406953       60.406953
24        63       40.360872       40.360872       40.360872
24        64       30.886890       30.886890       30.886890
24        65       25.961510       25.961510       25.961510
24        66       35.128336       35.128336       35.128336
24        67       32.140317       32.140317       32.140317
24        68       45.221676       45.221676       45.221676
24        69       55.036352       55.036352       55.036352
24        70       47.518417       47.518417       47.518417
24        71       64.070274       64.070274       64.070274
24        72       56.824291       56.824291       56.824291
24        73       65.000000       65.000000       65.000000
24        74       65.734314       65.734314       65.734314
24        75       23.430749       23.430749       23.430749
24        76       33.000000       33.000000       33.000000
24        77       23.086793       23.086793       23.086793
24        78       15.132746       15.132746       15.132746
24        79       59.169249       59.169249       59.169249
24        80       70.661163       70.661163       70.661163
24        81       42.953463       42.953463       42.953463
24        82       54.129474       54.129474       54.129474
24        83       39.560081       39.560081       39.560081
24        84       26.019224       26.019224       26.019224
24        85       30.610456       30.610456       30.610456
24        86       30.805844       30.805844       30.805844
24        87       25.495098       25.495098       25.495098
24        88       32.202484       32.202484       32.202484
24        89       54.817880       54.817880       54.817880
24        90       29.000000       29.000000       29.000000
24        91       42.011903       42.011903       42.011903
24        92       38.600518       38.600518       38.600518
24        93       40.853396       40.853396       40.853396
24        94       52.325902       52.325902       52.325902
24        95       47.010637       47.010637       47.010637
24        96       36.715120       36.715120       36.715120
24        97       51.865210       51.865210       51.865210
24        98       36.400549       36.400549       36.400549
24        99       48.507731       48.507731       48.507731
24        100      32.310989       32.310989       32.310989
24        101      62.393910       62.393910       62.393910
25        1        35.057096       35.057096       35.057096
25        2        71.196910       71.196910       71.196910
25        3        62.096699       62.096699       62.096699
25        4        71.805292       71.805292       71.805292
25        5        67.446275       67.446275       67.446275
25        6        72.277244       72.277244       72.277244
25        7        63.245553       63.245553       63.245553
25        8        64.257295       64.257295       64.257295
25        9        68.949257       68.949257       68.949257
25        10       34.409301       34.409301       34.409301
```

```
25        11        37.536649        37.536649        37.536649
25        12        39.051248        39.051248        39.051248
25        13        42.426407        42.426407        42.426407
25        14        38.587563        38.587563        38.587563
25        15        44.598206        44.598206        44.598206
25        16        43.829214        43.829214        43.829214
25        17        45.486262        45.486262        45.486262
25        18        48.414874        48.414874        48.414874
25        19        11.661904        11.661904        11.661904
25        20        6.403124        6.403124        6.403124
25        21        4.000000        4.000000        4.000000
25        22        10.198039        10.198039        10.198039
25        23        2.000000        2.000000        2.000000
25        24        10.000000        10.000000        10.000000
25        26        10.440307        10.440307        10.440307
25        27        58.940648        58.940648        58.940648
25        28        60.406953        60.406953        60.406953
25        29        56.044625        56.044625        56.044625
25        30        55.713553        55.713553        55.713553
25        31        52.201533        52.201533        52.201533
25        32        53.851648        53.851648        53.851648
25        33        51.244512        51.244512        51.244512
25        34        48.052055        48.052055        48.052055
25        35        51.078371        51.078371        51.078371
25        36        75.769387        75.769387        75.769387
25        37        75.026662        75.026662        75.026662
25        38        72.235725        72.235725        72.235725
25        39        69.289249        69.289249        69.289249
25        40        68.622154        68.622154        68.622154
25        41        73.375745        73.375745        73.375745
25        42        63.245553        63.245553        63.245553
25        43        67.186308        67.186308        67.186308
25        44        72.034714        72.034714        72.034714
25        45        69.123079        69.123079        69.123079
25        46        69.375788        69.375788        69.375788
25        47        68.007353        68.007353        68.007353
25        48        46.861498        46.861498        46.861498
25        49        10.770330        10.770330        10.770330
25        50        5.000000        5.000000        5.000000
25        51        39.446166        39.446166        39.446166
25        52        17.720045        17.720045        17.720045
25        53        19.849433        19.849433        19.849433
25        54        39.357337        39.357337        39.357337
25        55        48.104054        48.104054        48.104054
25        56        45.705580        45.705580        45.705580
25        57        23.323808        23.323808        23.323808
25        58        12.806248        12.806248        12.806248
25        59        23.537205        23.537205        23.537205
25        60        28.442925        28.442925        28.442925
25        61        50.537115        50.537115        50.537115
25        62        50.487622        50.487622        50.487622
25        63        33.600595        33.600595        33.600595
25        64        27.459060        27.459060        27.459060
25        65        16.552945        16.552945        16.552945
25        66        25.179357        25.179357        25.179357
25        67        22.203603        22.203603        22.203603
25        68        37.483330        37.483330        37.483330
25        69        45.044423        45.044423        45.044423
25        70        37.656341        37.656341        37.656341
25        71        54.083269        54.083269        54.083269
25        72        48.259714        48.259714        48.259714
25        73        55.901699        55.901699        55.901699
25        74        57.628118        57.628118        57.628118
```

```
25       75       18.681542        18.681542        18.681542
25       76       34.481879        34.481879        34.481879
25       77       22.203603        22.203603        22.203603
25       78       19.209373        19.209373        19.209373
25       79       50.803543        50.803543        50.803543
25       80       61.911227        61.911227        61.911227
25       81       33.241540        33.241540        33.241540
25       82       44.384682        44.384682        44.384682
25       83       30.083218        30.083218        30.083218
25       84       16.031220        16.031220        16.031220
25       85       23.600847        23.600847        23.600847
25       86       26.248809        26.248809        26.248809
25       87       19.235384        19.235384        19.235384
25       88       27.513633        27.513633        27.513633
25       89       45.221676        45.221676        45.221676
25       90       30.675723        30.675723        30.675723
25       91       32.015621        32.015621        32.015621
25       92       29.154759        29.154759        29.154759
25       93       31.764760        31.764760        31.764760
25       94       43.566042        43.566042        43.566042
25       95       38.078866        38.078866        38.078866
25       96       28.425341        28.425341        28.425341
25       97       42.544095        42.544095        42.544095
25       98       34.132096        34.132096        34.132096
25       99       38.897301        38.897301        38.897301
25       100       23.323808        23.323808        23.323808
25       101       52.469038        52.469038        52.469038
26       1       45.276926        45.276926        45.276926
26       2       80.622577        80.622577        80.622577
26       3       71.196910        71.196910        71.196910
26       4       81.049368        81.049368        81.049368
26       5       76.485293        76.485293        76.485293
26       6       81.394103        81.394103        81.394103
26       7       72.034714        72.034714        72.034714
26       8       72.801099        72.801099        72.801099
26       9       77.620873        77.620873        77.620873
26       10       39.051248        39.051248        39.051248
26       11       43.011626        43.011626        43.011626
26       12       44.204072        44.204072        44.204072
26       13       48.259714        48.259714        48.259714
26       14       42.426407        42.426407        42.426407
26       15       50.000000        50.000000        50.000000
26       16       48.104054        48.104054        48.104054
26       17       49.497475        49.497475        49.497475
26       18       53.150729        53.150729        53.150729
26       19       9.000000        9.000000        9.000000
26       20       8.602325        8.602325        8.602325
26       21       12.206556        12.206556        12.206556
26       22       5.000000        5.000000        5.000000
26       23       11.180340        11.180340        11.180340
26       24       3.000000        3.000000        3.000000
26       25       10.440307        10.440307        10.440307
26       27       65.000000        65.000000        65.000000
26       28       67.082039        67.082039        67.082039
26       29       62.241465        62.241465        62.241465
26       30       62.649820        62.649820        62.649820
26       31       58.600341        58.600341        58.600341
26       32       60.901560        60.901560        60.901560
26       33       57.697487        57.697487        57.697487
26       34       53.851648        53.851648        53.851648
26       35       58.309519        58.309519        58.309519
26       36       86.162637        86.162637        86.162637
26       37       85.440037        85.440037        85.440037
```

```
26        38       82.637764        82.637764        82.637764
26        39       79.711982        79.711982        79.711982
26        40       79.056942        79.056942        79.056942
26        41       83.815273        83.815273        83.815273
26        42       73.681748        73.681748        73.681748
26        43       77.620873        77.620873        77.620873
26        44       82.462113        82.462113        82.462113
26        45       79.555012        79.555012        79.555012
26        46       78.447435        78.447435        78.447435
26        47       76.902536        76.902536        76.902536
26        48       51.855569        51.855569        51.855569
26        49       7.000000        7.000000        7.000000
26        50       9.899495        9.899495        9.899495
26        51       47.634021        47.634021        47.634021
26        52       25.000000        25.000000        25.000000
26        53       26.925824        26.925824        26.925824
26        54       47.434165        47.434165        47.434165
26        55       58.523500        58.523500        58.523500
26        56       55.226805        55.226805        55.226805
26        57       33.541020        33.541020        33.541020
26        58       20.615528        20.615528        20.615528
26        59       20.615528        20.615528        20.615528
26        60       29.154759        29.154759        29.154759
26        61       58.523500        58.523500        58.523500
26        62       60.827625        60.827625        60.827625
26        63       42.426407        42.426407        42.426407
26        64       33.541020        33.541020        33.541020
26        65       26.925824        26.925824        26.925824
26        66       35.000000        35.000000        35.000000
26        67       32.557641        32.557641        32.557641
26        68       47.010637        47.010637        47.010637
26        69       55.226805        55.226805        55.226805
26        70       47.169906        47.169906        47.169906
26        71       64.000000        64.000000        64.000000
26        72       58.309519        58.309519        58.309519
26        73       66.211781        66.211781        66.211781
26        74       64.140471        64.140471        64.140471
26        75       21.213203        21.213203        21.213203
26        76       30.000000        30.000000        30.000000
26        77       25.961510        25.961510        25.961510
26        78       12.165525        12.165525        12.165525
26        79       57.706152        57.706152        57.706152
26        80       69.354164        69.354164        69.354164
26        81       43.680659        43.680659        43.680659
26        82       54.817880        54.817880        54.817880
26        83       38.832976        38.832976        38.832976
26        84       26.076810        26.076810        26.076810
26        85       32.557641        32.557641        32.557641
26        86       33.286634        33.286634        33.286634
26        87       23.600847        23.600847        23.600847
26        88       29.832868        29.832868        29.832868
26        89       54.129474        54.129474        54.129474
26        90       32.000000        32.000000        32.000000
26        91       42.047592        42.047592        42.047592
26        92       39.560081        39.560081        39.560081
26        93       42.047592        42.047592        42.047592
26        94       53.712196        53.712196        53.712196
26        95       48.301139        48.301139        48.301139
26        96       38.275318        38.275318        38.275318
26        97       52.924474        52.924474        52.924474
26        98       33.615473        33.615473        33.615473
26        99       47.853944        47.853944        47.853944
26        100       31.320920        31.320920        31.320920
```

```
26        101        62.128898        62.128898        62.128898
27        1        58.523500        58.523500        58.523500
27        2        89.022469        89.022469        89.022469
27        3        85.755466        85.755466        85.755466
27        4        91.400219        91.400219        91.400219
27        5        90.138782        90.138782        90.138782
27        6        93.005376        93.005376        93.005376
27        7        89.185201        89.185201        89.185201
27        8        91.787799        91.787799        91.787799
27        9        94.339811        94.339811        94.339811
27        10        85.146932        85.146932        85.146932
27        11        85.586214        85.586214        85.586214
27        12        87.572827        87.572827        87.572827
27        13        88.283634        88.283634        88.283634
27        14        90.138782        90.138782        90.138782
27        15        91.241438        91.241438        91.241438
27        16        93.536089        93.536089        93.536089
27        17        95.524866        95.524866        95.524866
27        18        96.176920        96.176920        96.176920
27        19        56.797887        56.797887        56.797887
27        20        56.648036        56.648036        56.648036
27        21        55.081757        55.081757        55.081757
27        22        60.415230        60.415230        60.415230
27        23        57.008771        57.008771        57.008771
27        24        62.241465        62.241465        62.241465
27        25        58.940648        58.940648        58.940648
27        26        65.000000        65.000000        65.000000
27        28        5.000000        5.000000        5.000000
27        29        3.000000        3.000000        3.000000
27        30        7.071068        7.071068        7.071068
27        31        7.000000        7.000000        7.000000
27        32        8.602325        8.602325        8.602325
27        33        8.000000        8.000000        8.000000
27        34        11.180340        11.180340        11.180340
27        35        11.180340        11.180340        11.180340
27        36        61.717096        61.717096        61.717096
27        37        62.649820        62.649820        62.649820
27        38        60.033324        60.033324        60.033324
27        39        59.908263        59.908263        59.908263
27        40        61.032778        61.032778        61.032778
27        41        65.192024        65.192024        65.192024
27        42        58.258047        58.258047        58.258047
27        43        64.031242        64.031242        64.031242
27        44        68.007353        68.007353        68.007353
27        45        65.604878        65.604878        65.604878
27        46        91.263355        91.263355        91.263355
27        47        91.809586        91.809586        91.809586
27        48        94.201911        94.201911        94.201911
27        49        58.600341        58.600341        58.600341
27        50        55.973208        55.973208        55.973208
27        51        23.537205        23.537205        23.537205
27        52        41.231056        41.231056        41.231056
27        53        70.000000        70.000000        70.000000
27        54        77.620873        77.620873        77.620873
27        55        50.000000        50.000000        50.000000
27        56        71.589105        71.589105        71.589105
27        57        45.276926        45.276926        45.276926
27        58        65.192024        65.192024        65.192024
27        59        82.462113        82.462113        82.462113
27        60        85.586214        85.586214        85.586214
27        61        85.440037        85.440037        85.440037
27        62        61.032778        61.032778        61.032778
27        63        30.413813        30.413813        30.413813
```

```
27        64        31.622777        31.622777        31.622777
27        65        50.000000        50.000000        50.000000
27        66        60.827625        60.827625        60.827625
27        67        54.451814        54.451814        54.451814
27        68        33.241540        33.241540        33.241540
27        69        62.649820        62.649820        62.649820
27        70        67.675697        67.675697        67.675697
27        71        71.561163        71.561163        71.561163
27        72        39.051248        39.051248        39.051248
27        73        47.423623        47.423623        47.423623
27        74        97.718985        97.718985        97.718985
27        75        75.663730        75.663730        75.663730
27        76        93.407708        93.407708        93.407708
27        77        39.357337        39.357337        39.357337
27        78        76.896034        76.896034        76.896034
27        79        90.801982        90.801982        90.801982
27        80        96.772930        96.772930        96.772930
27        81        50.921508        50.921508        50.921508
27        82        53.851648        53.851648        53.851648
27        83        69.231496        69.231496        69.231496
27        84        58.008620        58.008620        58.008620
27        85        38.013156        38.013156        38.013156
27        86        32.756679        32.756679        32.756679
27        87        74.242845        74.242845        74.242845
27        88        83.216585        83.216585        83.216585
27        89        76.321688        76.321688        76.321688
27        90        37.536649        37.536649        37.536649
27        91        60.440053        60.440053        60.440053
27        92        47.539457        47.539457        47.539457
27        93        43.965896        43.965896        43.965896
27        94        40.496913        40.496913        40.496913
27        95        42.047592        42.047592        42.047592
27        96        39.623226        39.623226        39.623226
27        97        46.647615        46.647615        46.647615
27        98        91.787799        91.787799        91.787799
27        99        72.422372        72.422372        72.422372
27        100        69.180922        69.180922        69.180922
27        101        73.925638        73.925638        73.925638
28        1        57.008771        57.008771        57.008771
28        2        86.023253        86.023253        86.023253
28        3        83.240615        83.240615        83.240615
28        4        88.481637        88.481637        88.481637
28        5        87.464278        87.464278        87.464278
28        6        90.138782        90.138782        90.138782
28        7        86.769810        86.769810        86.769810
28        8        89.442719        89.442719        89.442719
28        9        91.787799        91.787799        91.787799
28        10        85.000000        85.000000        85.000000
28        11        85.146932        85.146932        85.146932
28        12        87.143560        87.143560        87.143560
28        13        87.572827        87.572827        87.572827
28        14        90.000000        90.000000        90.000000
28        15        90.553851        90.553851        90.553851
28        16        93.134312        93.134312        93.134312
28        17        95.131488        95.131488        95.131488
28        18        95.524866        95.524866        95.524866
28        19        59.169249        59.169249        59.169249
28        20        58.600341        58.600341        58.600341
28        21        56.648036        56.648036        56.648036
28        22        62.649820        62.649820        62.649820
28        23        58.523500        58.523500        58.523500
28        24        64.412732        64.412732        64.412732
28        25        60.406953        60.406953        60.406953
```

```
28          26          67.082039         67.082039         67.082039
28          27          5.000000         5.000000         5.000000
28          29          5.830952         5.830952         5.830952
28          30          5.000000         5.000000         5.000000
28          31          8.602325         8.602325         8.602325
28          32          7.000000         7.000000         7.000000
28          33          9.433981         9.433981         9.433981
28          34          14.142136         14.142136         14.142136
28          35          10.000000         10.000000         10.000000
28          36          57.306195         57.306195         57.306195
28          37          58.309519         58.309519         58.309519
28          38          55.758407         55.758407         55.758407
28          39          55.803226         55.803226         55.803226
28          40          57.008771         57.008771         57.008771
28          41          61.032778         61.032778         61.032778
28          42          54.488531         54.488531         54.488531
28          43          60.207973         60.207973         60.207973
28          44          64.031242         64.031242         64.031242
28          45          61.717096         61.717096         61.717096
28          46          88.509886         88.509886         88.509886
28          47          89.185201         89.185201         89.185201
28          48          93.536089         93.536089         93.536089
28          49          60.901560         60.901560         60.901560
28          50          57.775427         57.775427         57.775427
28          51          23.000000         23.000000         23.000000
28          52          42.720019         42.720019         42.720019
28          53          70.178344         70.178344         70.178344
28          54          76.485293         76.485293         76.485293
28          55          47.169906         47.169906         47.169906
28          56          69.641941         69.641941         69.641941
28          57          45.000000         45.000000         45.000000
28          58          65.764732         65.764732         65.764732
28          59          83.815273         83.815273         83.815273
28          60          86.313383         86.313383         86.313383
28          61          83.815273         83.815273         83.815273
28          62          58.309519         58.309519         58.309519
28          63          30.000000         30.000000         30.000000
28          64          33.541020         33.541020         33.541020
28          65          50.249378         50.249378         50.249378
28          66          60.207973         60.207973         60.207973
28          67          54.037024         54.037024         54.037024
28          68          31.780497         31.780497         31.780497
28          69          60.415230         60.415230         60.415230
28          70          66.219333         66.219333         66.219333
28          71          68.963759         68.963759         68.963759
28          72          36.055513         36.055513         36.055513
28          73          43.863424         43.863424         43.863424
28          74          96.301610         96.301610         96.301610
28          75          76.485293         76.485293         76.485293
28          76          94.868330         94.868330         94.868330
28          77          41.880783         41.880783         41.880783
28          78          78.790862         78.790862         78.790862
28          79          89.498603         89.498603         89.498603
28          80          94.921020         94.921020         94.921020
28          81          49.477268         49.477268         49.477268
28          82          51.429563         51.429563         51.429563
28          83          68.468971         68.468971         68.468971
28          84          58.137767         58.137767         58.137767
28          85          38.470768         38.470768         38.470768
28          86          34.176015         34.176015         34.176015
28          87          74.813100         74.813100         74.813100
28          88          83.725743         83.725743         83.725743
28          89          74.632433         74.632433         74.632433
```

```
28       90       41.036569       41.036569       41.036569
28       91       59.228372       59.228372       59.228372
28       92       46.529560       46.529560       46.529560
28       93       42.755117       42.755117       42.755117
28       94       38.013156       38.013156       38.013156
28       95       40.162171       40.162171       40.162171
28       96       39.051248       39.051248       39.051248
28       97       44.283180       44.283180       44.283180
28       98       92.574294       92.574294       92.574294
28       99       71.063352       71.063352       71.063352
28       100       69.000000       69.000000       69.000000
28       101       71.554175       71.554175       71.554175
29       1       55.713553       55.713553       55.713553
29       2       86.683332       86.683332       86.683332
29       3       83.216585       83.216585       83.216585
29       4       89.022469       89.022469       89.022469
29       5       87.658428       87.658428       87.658428
29       6       90.603532       90.603532       90.603532
29       7       86.608314       86.608314       86.608314
29       8       89.185201       89.185201       89.185201
29       9       91.809586       91.809586       91.809586
29       10       82.152298       82.152298       82.152298
29       11       82.607506       82.607506       82.607506
29       12       84.593144       84.593144       84.593144
29       13       85.328776       85.328776       85.328776
29       14       87.143560       87.143560       87.143560
29       15       88.283634       88.283634       88.283634
29       16       90.553851       90.553851       90.553851
29       17       92.541882       92.541882       92.541882
29       18       93.214806       93.214806       93.214806
29       19       54.120237       54.120237       54.120237
29       20       53.851648       53.851648       53.851648
29       21       52.201533       52.201533       52.201533
29       22       57.697487       57.697487       57.697487
29       23       54.120237       54.120237       54.120237
29       24       59.506302       59.506302       59.506302
29       25       56.044625       56.044625       56.044625
29       26       62.241465       62.241465       62.241465
29       27       3.000000       3.000000       3.000000
29       28       5.830952       5.830952       5.830952
29       30       5.385165       5.385165       5.385165
29       31       4.000000       4.000000       4.000000
29       32       6.403124       6.403124       6.403124
29       33       5.000000       5.000000       5.000000
29       34       8.602325       8.602325       8.602325
29       35       8.602325       8.602325       8.602325
29       36       60.415230       60.415230       60.415230
29       37       61.269895       61.269895       61.269895
29       38       58.591808       58.591808       58.591808
29       39       58.309519       58.309519       58.309519
29       40       59.363288       59.363288       59.363288
29       41       63.631753       63.631753       63.631753
29       42       56.400355       56.400355       56.400355
29       43       62.201286       62.201286       62.201286
29       44       66.287254       66.287254       66.287254
29       45       63.820060       63.820060       63.820060
29       46       88.814413       88.814413       88.814413
29       47       89.308454       89.308454       89.308454
29       48       91.241438       91.241438       91.241438
29       49       55.901699       55.901699       55.901699
29       50       53.141321       53.141321       53.141321
29       51       20.615528       20.615528       20.615528
29       52       38.327536       38.327536       38.327536
```

```
29        53        67.000000        67.000000        67.000000
29        54        74.726167        74.726167        74.726167
29        55        47.634021        47.634021        47.634021
29        56        68.876701        68.876701        68.876701
29        57        42.296572        42.296572        42.296572
29        58        62.201286        62.201286        62.201286
29        59        79.555012        79.555012        79.555012
29        60        82.607506        82.607506        82.607506
29        61        82.637764        82.637764        82.637764
29        62        58.600341        58.600341        58.600341
29        63        27.459060        27.459060        27.459060
29        64        28.792360        28.792360        28.792360
29        65        47.000000        47.000000        47.000000
29        66        57.870545        57.870545        57.870545
29        67        51.478151        51.478151        51.478151
29        68        30.463092        30.463092        30.463092
29        69        60.033324        60.033324        60.033324
29        70        64.845971        64.845971        64.845971
29        71        69.065187        69.065187        69.065187
29        72        36.796739        36.796739        36.796739
29        73        45.453273        45.453273        45.453273
29        74        94.868330        94.868330        94.868330
29        75        72.691127        72.691127        72.691127
29        76        90.520716        90.520716        90.520716
29        77        36.715120        36.715120        36.715120
29        78        74.094534        74.094534        74.094534
29        79        87.931792        87.931792        87.931792
29        80        94.021274        94.021274        94.021274
29        81        48.104054        48.104054        48.104054
29        82        51.312766        51.312766        51.312766
29        83        66.287254        66.287254        66.287254
29        84        55.009090        55.009090        55.009090
29        85        35.014283        35.014283        35.014283
29        86        29.832868        29.832868        29.832868
29        87        71.253070        71.253070        71.253070
29        88        80.224684        80.224684        80.224684
29        89        73.539105        73.539105        73.539105
29        90        35.355339        35.355339        35.355339
29        91        57.567352        57.567352        57.567352
29        92        44.643029        44.643029        44.643029
29        93        41.109610        41.109610        41.109610
29        94        38.013156        38.013156        38.013156
29        95        39.357337        39.357337        39.357337
29        96        36.674242        36.674242        36.674242
29        97        44.102154        44.102154        44.102154
29        98        88.814413        88.814413        88.814413
29        99        69.570109        69.570109        69.570109
29        100        66.189123        66.189123        66.189123
29        101        71.344236        71.344236        71.344236
30        1        52.201533        52.201533        52.201533
30        2        82.006097        82.006097        82.006097
30        3        78.892332        78.892332        78.892332
30        4        84.403791        84.403791        84.403791
30        5        83.216585        83.216585        83.216585
30        6        86.023253        86.023253        86.023253
30        7        82.365041        82.365041        82.365041
30        8        85.000000        85.000000        85.000000
30        9        87.464278        87.464278        87.464278
30        10        80.000000        80.000000        80.000000
30        11        80.156098        80.156098        80.156098
30        12        82.152298        82.152298        82.152298
30        13        82.607506        82.607506        82.607506
30        14        85.000000        85.000000        85.000000
```

```
30        15        85.586214        85.586214        85.586214
30        16        88.141931        88.141931        88.141931
30        17        90.138782        90.138782        90.138782
30        18        90.553851        90.553851        90.553851
30        19        54.918121        54.918121        54.918121
30        20        54.120237        54.120237        54.120237
30        21        52.000000        52.000000        52.000000
30        22        58.309519        58.309519        58.309519
30        23        53.851648        53.851648        53.851648
30        24        60.033324        60.033324        60.033324
30        25        55.713553        55.713553        55.713553
30        26        62.649820        62.649820        62.649820
30        27        7.071068        7.071068        7.071068
30        28        5.000000        5.000000        5.000000
30        29        5.385165        5.385165        5.385165
30        31        5.385165        5.385165        5.385165
30        32        2.000000        2.000000        2.000000
30        33        5.830952        5.830952        5.830952
30        34        11.180340        11.180340        11.180340
30        35        5.000000        5.000000        5.000000
30        36        55.036352        55.036352        55.036352
30        37        55.901699        55.901699        55.901699
30        38        53.235327        53.235327        53.235327
30        39        53.000000        53.000000        53.000000
30        40        54.083269        54.083269        54.083269
30        41        58.309519        58.309519        58.309519
30        42        51.224994        51.224994        51.224994
30        43        57.008771        57.008771        57.008771
30        44        61.032778        61.032778        61.032778
30        45        58.600341        58.600341        58.600341
30        46        84.314886        84.314886        84.314886
30        47        84.905830        84.905830        84.905830
30        48        88.566359        88.566359        88.566359
30        49        56.603887        56.603887        56.603887
30        50        53.225934        53.225934        53.225934
30        51        18.000000        18.000000        18.000000
30        52        38.078866        38.078866        38.078866
30        53        65.192024        65.192024        65.192024
30        54        71.589105        71.589105        71.589105
30        55        43.011626        43.011626        43.011626
30        56        65.000000        65.000000        65.000000
30        57        40.000000        40.000000        40.000000
30        58        60.827625        60.827625        60.827625
30        59        79.056942        79.056942        79.056942
30        60        81.394103        81.394103        81.394103
30        61        79.056942        79.056942        79.056942
30        62        54.083269        54.083269        54.083269
30        63        25.000000        25.000000        25.000000
30        64        29.154759        29.154759        29.154759
30        65        45.276926        45.276926        45.276926
30        66        55.226805        55.226805        55.226805
30        67        49.040799        49.040799        49.040799
30        68        26.925824        26.925824        26.925824
30        69        55.901699        55.901699        55.901699
30        70        61.400326        61.400326        61.400326
30        71        64.660653        64.660653        64.660653
30        72        32.015621        32.015621        32.015621
30        73        40.360872        40.360872        40.360872
30        74        91.482239        91.482239        91.482239
30        75        71.589105        71.589105        71.589105
30        76        90.138782        90.138782        90.138782
30        77        37.802116        37.802116        37.802116
30        78        74.249579        74.249579        74.249579
```

```
30        79        84.646323        84.646323        84.646323
30        80        90.249654        90.249654        90.249654
30        81        44.643029        44.643029        44.643029
30        82        47.010637        47.010637        47.010637
30        83        63.505905        63.505905        63.505905
30        84        53.150729        53.150729        53.150729
30        85        33.541020        33.541020        33.541020
30        86        29.546573        29.546573        29.546573
30        87        69.871310        69.871310        69.871310
30        88        78.771822        78.771822        78.771822
30        89        69.892775        69.892775        69.892775
30        90        37.802116        37.802116        37.802116
30        91        54.341513        54.341513        54.341513
30        92        41.593269        41.593269        41.593269
30        93        37.854986        37.854986        37.854986
30        94        33.615473        33.615473        33.615473
30        95        35.468296        35.468296        35.468296
30        96        34.058773        34.058773        34.058773
30        97        39.824616        39.824616        39.824616
30        98        87.664132        87.664132        87.664132
30        99        66.219333        66.219333        66.219333
30        100        64.000000        64.000000        64.000000
30        101        67.119297        67.119297        67.119297
31        1        52.000000        52.000000        52.000000
31        2        83.630138        83.630138        83.630138
31        3        79.881162        79.881162        79.881162
31        4        85.912746        85.912746        85.912746
31        5        84.403791        84.403791        84.403791
31        6        87.458562        87.458562        87.458562
31        7        83.216585        83.216585        83.216585
31        8        85.755466        85.755466        85.755466
31        9        88.481637        88.481637        88.481637
31        10        78.160092        78.160092        78.160092
31        11        78.638413        78.638413        78.638413
31        12        80.622577        80.622577        80.622577
31        13        81.394103        81.394103        81.394103
31        14        83.150466        83.150466        83.150466
31        15        84.344532        84.344532        84.344532
31        16        86.579443        86.579443        86.579443
31        17        88.566359        88.566359        88.566359
31        18        89.269256        89.269256        89.269256
31        19        50.606324        50.606324        50.606324
31        20        50.159745        50.159745        50.159745
31        21        48.383882        48.383882        48.383882
31        22        54.120237        54.120237        54.120237
31        23        50.289164        50.289164        50.289164
31        24        55.901699        55.901699        55.901699
31        25        52.201533        52.201533        52.201533
31        26        58.600341        58.600341        58.600341
31        27        7.000000        7.000000        7.000000
31        28        8.602325        8.602325        8.602325
31        29        4.000000        4.000000        4.000000
31        30        5.385165        5.385165        5.385165
31        32        5.000000        5.000000        5.000000
31        33        1.000000        1.000000        1.000000
31        34        5.830952        5.830952        5.830952
31        35        5.830952        5.830952        5.830952
31        36        58.872744        58.872744        58.872744
31        37        59.615434        59.615434        59.615434
31        38        56.859476        56.859476        56.859476
31        39        56.356011        56.356011        56.356011
31        40        57.306195        57.306195        57.306195
31        41        61.717096        61.717096        61.717096
```

```
31        42       54.083269       54.083269       54.083269
31        43       59.908263       59.908263       59.908263
31        44       64.140471       64.140471       64.140471
31        45       61.587336       61.587336       61.587336
31        46       85.603738       85.603738       85.603738
31        47       86.023253       86.023253       86.023253
31        48       87.298339       87.298339       87.298339
31        49       52.354560       52.354560       52.354560
31        50       49.396356       49.396356       49.396356
31        51       16.763055       16.763055       16.763055
31        52       34.481879       34.481879       34.481879
31        53       63.000000       63.000000       63.000000
31        54       70.880181       70.880181       70.880181
31        55       44.598206       44.598206       44.598206
31        56       65.299311       65.299311       65.299311
31        57       38.327536       38.327536       38.327536
31        58       58.215118       58.215118       58.215118
31        59       75.690158       75.690158       75.690158
31        60       78.638413       78.638413       78.638413
31        61       78.924014       78.924014       78.924014
31        62       55.443665       55.443665       55.443665
31        63       23.537205       23.537205       23.537205
31        64       25.079872       25.079872       25.079872
31        65       43.000000       43.000000       43.000000
31        66       53.935146       53.935146       53.935146
31        67       47.518417       47.518417       47.518417
31        68       26.832816       26.832816       26.832816
31        69       56.603887       56.603887       56.603887
31        70       61.098281       61.098281       61.098281
31        71       65.802736       65.802736       65.802736
31        72       33.970576       33.970576       33.970576
31        73       43.011626       43.011626       43.011626
31        74       91.082380       91.082380       91.082380
31        75       68.731361       68.731361       68.731361
31        76       86.683332       86.683332       86.683332
31        77       33.286634       33.286634       33.286634
31        78       70.384657       70.384657       70.384657
31        79       84.118963       84.118963       84.118963
31        80       90.376988       90.376988       90.376988
31        81       44.384682       44.384682       44.384682
31        82       48.010416       48.010416       48.010416
31        83       62.369865       62.369865       62.369865
31        84       51.009803       51.009803       51.009803
31        85       31.016125       31.016125       31.016125
31        86       25.961510       25.961510       25.961510
31        87       67.268120       67.268120       67.268120
31        88       76.236474       76.236474       76.236474
31        89       69.856997       69.856997       69.856997
31        90       32.649655       32.649655       32.649655
31        91       53.758720       53.758720       53.758720
31        92       40.804412       40.804412       40.804412
31        93       37.336309       37.336309       37.336309
31        94       34.828150       34.828150       34.828150
31        95       35.846897       35.846897       35.846897
31        96       32.756679       32.756679       32.756679
31        97       40.804412       40.804412       40.804412
31        98       84.852814       84.852814       84.852814
31        99       65.787537       65.787537       65.787537
31        100       62.201286       62.201286       62.201286
31        101       67.955868       67.955868       67.955868
32        1       50.289164       50.289164       50.289164
32        2       80.430094       80.430094       80.430094
32        3       77.175126       77.175126       77.175126
```

```
32        4        82.800966        82.800966        82.800966
32        5        81.541401        81.541401        81.541401
32        6        84.403791        84.403791        84.403791
32        7        80.622577        80.622577        80.622577
32        8        83.240615        83.240615        83.240615
32        9        85.755466        85.755466        85.755466
32        10        78.000000        78.000000        78.000000
32        11        78.160092        78.160092        78.160092
32        12        80.156098        80.156098        80.156098
32        13        80.622577        80.622577        80.622577
32        14        83.000000        83.000000        83.000000
32        15        83.600239        83.600239        83.600239
32        16        86.145226        86.145226        86.145226
32        17        88.141931        88.141931        88.141931
32        18        88.566359        88.566359        88.566359
32        19        53.254108        53.254108        53.254108
32        20        52.354560        52.354560        52.354560
32        21        50.159745        50.159745        50.159745
32        22        56.603887        56.603887        56.603887
32        23        52.000000        52.000000        52.000000
32        24        58.309519        58.309519        58.309519
32        25        53.851648        53.851648        53.851648
32        26        60.901560        60.901560        60.901560
32        27        8.602325        8.602325        8.602325
32        28        7.000000        7.000000        7.000000
32        29        6.403124        6.403124        6.403124
32        30        2.000000        2.000000        2.000000
32        31        5.000000        5.000000        5.000000
32        33        5.099020        5.099020        5.099020
32        34        10.440307        10.440307        10.440307
32        35        3.000000        3.000000        3.000000
32        36        54.230987        54.230987        54.230987
32        37        55.036352        55.036352        55.036352
32        38        52.325902        52.325902        52.325902
32        39        51.971146        51.971146        51.971146
32        40        53.000000        53.000000        53.000000
32        41        57.306195        57.306195        57.306195
32        42        50.000000        50.000000        50.000000
32        43        55.803226        55.803226        55.803226
32        44        59.908263        59.908263        59.908263
32        45        57.428216        57.428216        57.428216
32        46        82.661962        82.661962        82.661962
32        47        83.216585        83.216585        83.216585
32        48        86.579443        86.579443        86.579443
32        49        54.918121        54.918121        54.918121
32        50        51.429563        51.429563        51.429563
32        51        16.000000        16.000000        16.000000
32        52        36.249138        36.249138        36.249138
32        53        63.198101        63.198101        63.198101
32        54        69.634761        69.634761        69.634761
32        55        41.400483        41.400483        41.400483
32        56        63.158531        63.158531        63.158531
32        57        38.000000        38.000000        38.000000
32        58        58.855756        58.855756        58.855756
32        59        77.162167        77.162167        77.162167
32        60        79.429214        79.429214        79.429214
32        61        77.162167        77.162167        77.162167
32        62        52.430907        52.430907        52.430907
32        63        23.000000        23.000000        23.000000
32        64        27.459060        27.459060        27.459060
32        65        43.289722        43.289722        43.289722
32        66        53.235327        53.235327        53.235327
32        67        47.042534        47.042534        47.042534
```

```
32        68        25.000000        25.000000        25.000000
32        69        54.120237        54.120237        54.120237
32        70        59.481089        59.481089        59.481089
32        71        62.968246        62.968246        62.968246
32        72        30.479501        30.479501        30.479501
32        73        39.051248        39.051248        39.051248
32        74        89.560036        89.560036        89.560036
32        75        69.634761        69.634761        69.634761
32        76        88.255311        88.255311        88.255311
32        77        36.235342        36.235342        36.235342
32        78        72.449983        72.449983        72.449983
32        79        82.710338        82.710338        82.710338
32        80        88.391176        88.391176        88.391176
32        81        42.720019        42.720019        42.720019
32        82        45.276926        45.276926        45.276926
32        83        61.522354        61.522354        61.522354
32        84        51.156622        51.156622        51.156622
32        85        31.575307        31.575307        31.575307
32        86        27.730849        27.730849        27.730849
32        87        67.896981        67.896981        67.896981
32        88        76.791927        76.791927        76.791927
32        89        68.007353        68.007353        68.007353
32        90        36.619667        36.619667        36.619667
32        91        52.392748        52.392748        52.392748
32        92        39.623226        39.623226        39.623226
32        93        35.902646        35.902646        35.902646
32        94        31.906112        31.906112        31.906112
32        95        33.615473        33.615473        33.615473
32        96        32.062439        32.062439        32.062439
32        97        38.078866        38.078866        38.078866
32        98        85.702975        85.702975        85.702975
32        99        64.288413        64.288413        64.288413
32        100        62.000000        62.000000        62.000000
32        101        65.368188        65.368188        65.368188
33        1        51.078371        51.078371        51.078371
33        2        82.879430        82.879430        82.879430
33        3        79.056942        79.056942        79.056942
33        4        85.146932        85.146932        85.146932
33        5        83.600239        83.600239        83.600239
33        6        86.683332        86.683332        86.683332
33        7        82.377181        82.377181        82.377181
33        8        84.905830        84.905830        84.905830
33        9        87.658428        87.658428        87.658428
33        10        77.162167        77.162167        77.162167
33        11        77.646635        77.646635        77.646635
33        12        79.630396        79.630396        79.630396
33        13        80.411442        80.411442        80.411442
33        14        82.152298        82.152298        82.152298
33        15        83.360662        83.360662        83.360662
33        16        85.586214        85.586214        85.586214
33        17        87.572827        87.572827        87.572827
33        18        88.283634        88.283634        88.283634
33        19        49.739320        49.739320        49.739320
33        20        49.244289        49.244289        49.244289
33        21        47.434165        47.434165        47.434165
33        22        53.235327        53.235327        53.235327
33        23        49.335586        49.335586        49.335586
33        24        55.009090        55.009090        55.009090
33        25        51.244512        51.244512        51.244512
33        26        57.697487        57.697487        57.697487
33        27        8.000000        8.000000        8.000000
33        28        9.433981        9.433981        9.433981
33        29        5.000000        5.000000        5.000000
```

```
33        30        5.830952        5.830952        5.830952
33        31        1.000000        1.000000        1.000000
33        32        5.099020        5.099020        5.099020
33        34        5.385165        5.385165        5.385165
33        35        5.385165        5.385165        5.385165
33        36        58.523500        58.523500        58.523500
33        37        59.236813        59.236813        59.236813
33        38        56.462377        56.462377        56.462377
33        39        55.901699        55.901699        55.901699
33        40        56.824291        56.824291        56.824291
33        41        61.269895        61.269895        61.269895
33        42        53.535035        53.535035        53.535035
33        43        59.363288        59.363288        59.363288
33        44        63.631753        63.631753        63.631753
33        45        61.057350        61.057350        61.057350
33        46        84.811556        84.811556        84.811556
33        47        85.211502        85.211502        85.211502
33        48        86.313383        86.313383        86.313383
33        49        51.478151        51.478151        51.478151
33        50        48.466483        48.466483        48.466483
33        51        15.811388        15.811388        15.811388
33        52        33.526109        33.526109        33.526109
33        53        62.000000        62.000000        62.000000
33        54        69.921384        69.921384        69.921384
33        55        43.863424        43.863424        43.863424
33        56        64.412732        64.412732        64.412732
33        57        37.336309        37.336309        37.336309
33        58        57.218878        57.218878        57.218878
33        59        74.726167        74.726167        74.726167
33        60        77.646635        77.646635        77.646635
33        61        78.000000        78.000000        78.000000
33        62        54.671748        54.671748        54.671748
33        63        22.561028        22.561028        22.561028
33        64        24.166092        24.166092        24.166092
33        65        42.000000        42.000000        42.000000
33        66        52.952809        52.952809        52.952809
33        67        46.529560        46.529560        46.529560
33        68        25.942244        25.942244        25.942244
33        69        55.758407        55.758407        55.758407
33        70        60.166436        60.166436        60.166436
33        71        65.000000        65.000000        65.000000
33        72        33.301652        33.301652        33.301652
33        73        42.438190        42.438190        42.438190
33        74        90.138782        90.138782        90.138782
33        75        67.742158        67.742158        67.742158
33        76        85.726309        85.726309        85.726309
33        77        32.449961        32.449961        32.449961
33        78        69.462220        69.462220        69.462220
33        79        83.168504        83.168504        83.168504
33        80        89.470666        89.470666        89.470666
33        81        43.462628        43.462628        43.462628
33        82        47.201695        47.201695        47.201695
33        83        61.392182        61.392182        61.392182
33        84        50.009999        50.009999        50.009999
33        85        30.016662        30.016662        30.016662
33        86        25.000000        25.000000        25.000000
33        87        66.272166        66.272166        66.272166
33        88        75.239617        75.239617        75.239617
33        89        68.942005        68.942005        68.942005
33        90        32.015621        32.015621        32.015621
33        91        52.810984        52.810984        52.810984
33        92        39.849718        39.849718        39.849718
33        93        36.400549        36.400549        36.400549
```

```
33        94        34.058773        34.058773        34.058773
33        95        34.985711        34.985711        34.985711
33        96        31.780497        31.780497        31.780497
33        97        40.000000        40.000000        40.000000
33        98        83.862983        83.862983        83.862983
33        99        64.845971        64.845971        64.845971
33        100        61.204575        61.204575        61.204575
33        101        67.119297        67.119297        67.119297
34        1        51.478151        51.478151        51.478151
34        2        84.852814        84.852814        84.852814
34        3        80.430094        80.430094        80.430094
34        4        87.000000        87.000000        87.000000
34        5        85.146932        85.146932        85.146932
34        6        88.459030        88.459030        88.459030
34        7        83.600239        83.600239        83.600239
34        8        86.023253        86.023253        86.023253
34        9        89.022469        89.022469        89.022469
34        10        75.663730        75.663730        75.663730
34        11        76.485293        76.485293        76.485293
34        12        78.447435        78.447435        78.447435
34        13        79.555012        79.555012        79.555012
34        14        80.622577        80.622577        80.622577
34        15        82.462113        82.462113        82.462113
34        16        84.344532        84.344532        84.344532
34        17        86.313383        86.313383        86.313383
34        18        87.321246        87.321246        87.321246
34        19        45.617979        45.617979        45.617979
34        20        45.541190        45.541190        45.541190
34        21        44.147480        44.147480        44.147480
34        22        49.244289        49.244289        49.244289
34        23        46.097722        46.097722        46.097722
34        24        51.078371        51.078371        51.078371
34        25        48.052055        48.052055        48.052055
34        26        53.851648        53.851648        53.851648
34        27        11.180340        11.180340        11.180340
34        28        14.142136        14.142136        14.142136
34        29        8.602325        8.602325        8.602325
34        30        11.180340        11.180340        11.180340
34        31        5.830952        5.830952        5.830952
34        32        10.440307        10.440307        10.440307
34        33        5.385165        5.385165        5.385165
34        35        10.000000        10.000000        10.000000
34        36        62.641839        62.641839        62.641839
34        37        63.245553        63.245553        63.245553
34        38        60.406953        60.406953        60.406953
34        39        59.615434        59.615434        59.615434
34        40        60.415230        60.415230        60.415230
34        41        65.000000        65.000000        65.000000
34        42        56.824291        56.824291        56.824291
34        43        62.649820        62.649820        62.649820
34        44        67.082039        67.082039        67.082039
34        45        64.412732        64.412732        64.412732
34        46        86.452299        86.452299        86.452299
34        47        86.683332        86.683332        86.683332
34        48        85.375641        85.375641        85.375641
34        49        47.423623        47.423623        47.423623
34        50        44.922155        44.922155        44.922155
34        51        16.401219        16.401219        16.401219
34        52        30.413813        30.413813        30.413813
34        53        60.207973        60.207973        60.207973
34        54        69.641941        69.641941        69.641941
34        55        46.097722        46.097722        46.097722
34        56        65.192024        65.192024        65.192024
```

```
34         57         36.400549        36.400549        36.400549
34         58         55.000000        55.000000        55.000000
34         59         71.589105        71.589105        71.589105
34         60         75.166482        75.166482        75.166482
34         61         78.262379        78.262379        78.262379
34         62         56.568542        56.568542        56.568542
34         63         22.360680        22.360680        22.360680
34         64         20.615528        20.615528        20.615528
34         65         40.311289        40.311289        40.311289
34         66         52.201533        52.201533        52.201533
34         67         45.607017        45.607017        45.607017
34         68         27.018512        27.018512        27.018512
34         69         57.008771        57.008771        57.008771
34         70         60.373835        60.373835        60.373835
34         71         66.603303        66.603303        66.603303
34         72         36.055513        36.055513        36.055513
34         73         45.650849        45.650849        45.650849
34         74         90.077744        90.077744        90.077744
34         75         65.192024        65.192024        65.192024
34         76         82.462113        82.462113        82.462113
34         77         28.178006        28.178006        28.178006
34         78         65.787537        65.787537        65.787537
34         79         83.006024        83.006024        83.006024
34         80         89.944427        89.944427        89.944427
34         81         43.908997        43.908997        43.908997
34         82         48.836462        48.836462        48.836462
34         83         60.728906        60.728906        60.728906
34         84         48.373546        48.373546        48.373546
34         85         28.284271        28.284271        28.284271
34         86         22.090722        22.090722        22.090722
34         87         64.007812        64.007812        64.007812
34         88         73.006849        73.006849        73.006849
34         89         69.354164        69.354164        69.354164
34         90         26.907248        26.907248        26.907248
34         91         52.801515        52.801515        52.801515
34         92         39.812058        39.812058        39.812058
34         93         36.715120        36.715120        36.715120
34         94         36.124784        36.124784        36.124784
34         95         36.235342        36.235342        36.235342
34         96         31.384710        31.384710        31.384710
34         97         41.725292        41.725292        41.725292
34         98         81.301906        81.301906        81.301906
34         99         64.884513        64.884513        64.884513
34         100         59.841457        59.841457        59.841457
34         101         68.410526        68.410526        68.410526
35         1         47.434165        47.434165        47.434165
35         2         78.102497        78.102497        78.102497
35         3         74.625733        74.625733        74.625733
35         4         80.430094        80.430094        80.430094
35         5         79.056942        79.056942        79.056942
35         6         82.006097        82.006097        82.006097
35         7         78.032045        78.032045        78.032045
35         8         80.622577        80.622577        80.622577
35         9         83.216585        83.216585        83.216585
35         10         75.000000        75.000000        75.000000
35         11         75.166482        75.166482        75.166482
35         12         77.162167        77.162167        77.162167
35         13         77.646635        77.646635        77.646635
35         14         80.000000        80.000000        80.000000
35         15         80.622577        80.622577        80.622577
35         16         83.150466        83.150466        83.150466
35         17         85.146932        85.146932        85.146932
35         18         85.586214        85.586214        85.586214
```

```
35        19        50.803543        50.803543        50.803543
35        20        49.739320        49.739320        49.739320
35        21        47.423623        47.423623        47.423623
35        22        54.083269        54.083269        54.083269
35        23        49.244289        49.244289        49.244289
35        24        55.758407        55.758407        55.758407
35        25        51.078371        51.078371        51.078371
35        26        58.309519        58.309519        58.309519
35        27        11.180340        11.180340        11.180340
35        28        10.000000        10.000000        10.000000
35        29        8.602325        8.602325        8.602325
35        30        5.000000        5.000000        5.000000
35        31        5.830952        5.830952        5.830952
35        32        3.000000        3.000000        3.000000
35        33        5.385165        5.385165        5.385165
35        34        10.000000        10.000000        10.000000
35        36        53.141321        53.141321        53.141321
35        37        53.851648        53.851648        53.851648
35        38        51.078371        51.078371        51.078371
35        39        50.537115        50.537115        50.537115
35        40        51.478151        51.478151        51.478151
35        41        55.901699        55.901699        55.901699
35        42        48.259714        48.259714        48.259714
35        43        54.083269        54.083269        54.083269
35        44        58.309519        58.309519        58.309519
35        45        55.758407        55.758407        55.758407
35        46        80.212219        80.212219        80.212219
35        47        80.709355        80.709355        80.709355
35        48        83.600239        83.600239        83.600239
35        49        52.430907        52.430907        52.430907
35        50        48.764741        48.764741        48.764741
35        51        13.000000        13.000000        13.000000
35        52        33.541020        33.541020        33.541020
35        53        60.207973        60.207973        60.207973
35        54        66.708320        66.708320        66.708320
35        55        39.051248        39.051248        39.051248
35        56        60.415230        60.415230        60.415230
35        57        35.000000        35.000000        35.000000
35        58        55.901699        55.901699        55.901699
35        59        74.330344        74.330344        74.330344
35        60        76.485293        76.485293        76.485293
35        61        74.330344        74.330344        74.330344
35        62        50.000000        50.000000        50.000000
35        63        20.000000        20.000000        20.000000
35        64        25.000000        25.000000        25.000000
35        65        40.311289        40.311289        40.311289
35        66        50.249378        50.249378        50.249378
35        67        44.045431        44.045431        44.045431
35        68        22.135944        22.135944        22.135944
35        69        51.478151        51.478151        51.478151
35        70        56.612719        56.612719        56.612719
35        71        60.464866        60.464866        60.464866
35        72        28.284271        28.284271        28.284271
35        73        37.202150        37.202150        37.202150
35        74        86.683332        86.683332        86.683332
35        75        66.708320        66.708320        66.708320
35        76        85.440037        85.440037        85.440037
35        77        33.970576        33.970576        33.970576
35        78        69.771054        69.771054        69.771054
35        79        79.812280        79.812280        79.812280
35        80        85.615419        85.615419        85.615419
35        81        39.849718        39.849718        39.849718
35        82        42.720019        42.720019        42.720019
```

```
35        83        58.549125        58.549125        58.549125
35        84        48.166378        48.166378        48.166378
35        85        28.635642        28.635642        28.635642
35        86        25.059928        25.059928        25.059928
35        87        64.938432        64.938432        64.938432
35        88        73.824115        73.824115        73.824115
35        89        65.192024        65.192024        65.192024
35        90        34.985711        34.985711        34.985711
35        91        49.477268        49.477268        49.477268
35        92        36.674242        36.674242        36.674242
35        93        32.984845        32.984845        32.984845
35        94        29.410882        29.410882        29.410882
35        95        30.870698        30.870698        30.870698
35        96        29.068884        29.068884        29.068884
35        97        35.510562        35.510562        35.510562
35        98        82.764727        82.764727        82.764727
35        99        61.400326        61.400326        61.400326
35        100        59.000000        59.000000        59.000000
35        101        62.769419        62.769419        62.769419
36        1        44.204072        44.204072        44.204072
36        2        42.000000        42.000000        42.000000
36        3        46.097722        46.097722        46.097722
36        4        45.000000        45.000000        45.000000
36        5        47.265209        47.265209        47.265209
36        6        47.000000        47.000000        47.000000
36        7        50.009999        50.009999        50.009999
36        8        52.952809        52.952809        52.952809
36        9        52.239832        52.239832        52.239832
36        10        75.822160        75.822160        75.822160
36        11        72.622311        72.622311        72.622311
36        12        74.202426        74.202426        74.202426
36        13        71.281134        71.281134        71.281134
36        14        79.649231        79.649231        79.649231
36        15        73.783467        73.783467        73.783467
36        16        79.056942        79.056942        79.056942
36        17        80.709355        80.709355        80.709355
36        18        78.032045        78.032045        78.032045
36        19        83.240615        83.240615        83.240615
36        20        79.056942        79.056942        79.056942
36        21        74.330344        74.330344        74.330344
36        22        84.433406        84.433406        84.433406
36        23        75.026662        75.026662        75.026662
36        24        85.094066        85.094066        85.094066
36        25        75.769387        75.769387        75.769387
36        26        86.162637        86.162637        86.162637
36        27        61.717096        61.717096        61.717096
36        28        57.306195        57.306195        57.306195
36        29        60.415230        60.415230        60.415230
36        30        55.036352        55.036352        55.036352
36        31        58.872744        58.872744        58.872744
36        32        54.230987        54.230987        54.230987
36        33        58.523500        58.523500        58.523500
36        34        62.641839        62.641839        62.641839
36        35        53.141321        53.141321        53.141321
36        37        2.000000        2.000000        2.000000
36        38        3.605551        3.605551        3.605551
36        39        7.071068        7.071068        7.071068
36        40        8.602325        8.602325        8.602325
36        41        7.000000        7.000000        7.000000
36        42        13.453624        13.453624        13.453624
36        43        13.000000        13.000000        13.000000
36        44        12.000000        12.000000        12.000000
36        45        12.369317        12.369317        12.369317
```

```
36        46       47.095647       47.095647       47.095647
36        47       49.254441       49.254441       49.254441
36        48       76.321688       76.321688       76.321688
36        49       83.815273       83.815273       83.815273
36        50       77.162167       77.162167       77.162167
36        51       50.249378       50.249378       50.249378
36        52       66.098411       66.098411       66.098411
36        53       69.202601       69.202601       69.202601
36        54       58.600341       58.600341       58.600341
36        55       27.730849       27.730849       27.730849
36        56       44.654227       44.654227       44.654227
36        57       52.810984       52.810984       52.810984
36        58       70.491134       70.491134       70.491134
36        59       91.263355       91.263355       91.263355
36        60       86.452299       86.452299       86.452299
36        61       57.697487       57.697487       57.697487
36        62       29.732137       29.732137       29.732137
36        63       50.039984       50.039984       50.039984
36        64       65.030762       65.030762       65.030762
36        65       59.236813       59.236813       59.236813
36        66       55.217751       55.217751       55.217751
36        67       54.589376       54.589376       54.589376
36        68       43.104524       43.104524       43.104524
36        69       36.796739       36.796739       36.796739
36        70       48.836462       48.836462       48.836462
36        71       35.777088       35.777088       35.777088
36        72       30.066593       30.066593       30.066593
36        73       20.396078       20.396078       20.396078
36        74       69.641941       69.641941       69.641941
36        75       80.212219       80.212219       80.212219
36        76       101.212647       101.212647       101.212647
36        77       73.334848       73.334848       73.334848
36        78       93.059121       93.059121       93.059121
36        79       65.741920       65.741920       65.741920
36        80       63.324561       63.324561       63.324561
36        81       42.941821       42.941821       42.941821
36        82       32.449961       32.449961       32.449961
36        83       58.000000       58.000000       58.000000
36        84       61.773781       61.773781       61.773781
36        85       56.885851       56.885851       56.885851
36        86       62.128898       62.128898       62.128898
36        87       76.400262       76.400262       76.400262
36        88       82.134037       82.134037       82.134037
36        89       50.774009       50.774009       50.774009
36        90       80.000000       80.000000       80.000000
36        91       48.414874       48.414874       48.414874
36        92       46.615448       46.615448       46.615448
36        93       44.271887       44.271887       44.271887
36        94       33.541020       33.541020       33.541020
36        95       38.327536       38.327536       38.327536
36        96       49.244289       49.244289       49.244289
36        97       33.241540       33.241540       33.241540
36        98       91.967386       91.967386       91.967386
36        99       52.630789       52.630789       52.630789
36        100       64.660653       64.660653       64.660653
36        101       40.249224       40.249224       40.249224
37        1       43.011626       43.011626       43.011626
37        2       40.000000       40.000000       40.000000
37        3       44.147480       44.147480       44.147480
37        4       43.000000       43.000000       43.000000
37        5       45.276926       45.276926       45.276926
37        6       45.000000       45.000000       45.000000
37        7       48.052055       48.052055       48.052055
```

```
37        8       50.990195       50.990195       50.990195
37        9       50.249378       50.249378       50.249378
37        10       74.330344       74.330344       74.330344
37        11       71.063352       71.063352       71.063352
37        12       72.622311       72.622311       72.622311
37        13       69.634761       69.634761       69.634761
37        14       78.102497       78.102497       78.102497
37        15       72.111026       72.111026       72.111026
37        16       77.420927       77.420927       77.420927
37        17       79.056942       79.056942       79.056942
37        18       76.321688       76.321688       76.321688
37        19       82.710338       82.710338       82.710338
37        20       78.447435       78.447435       78.447435
37        21       73.681748       73.681748       73.681748
37        22       83.815273       83.815273       83.815273
37        23       74.330344       74.330344       74.330344
37        24       84.433406       84.433406       84.433406
37        25       75.026662       75.026662       75.026662
37        26       85.440037       85.440037       85.440037
37        27       62.649820       62.649820       62.649820
37        28       58.309519       58.309519       58.309519
37        29       61.269895       61.269895       61.269895
37        30       55.901699       55.901699       55.901699
37        31       59.615434       59.615434       59.615434
37        32       55.036352       55.036352       55.036352
37        33       59.236813       59.236813       59.236813
37        34       63.245553       63.245553       63.245553
37        35       53.851648       53.851648       53.851648
37        36       2.000000       2.000000       2.000000
37        38       3.000000       3.000000       3.000000
37        39       5.830952       5.830952       5.830952
37        40       7.071068       7.071068       7.071068
37        41       5.000000       5.000000       5.000000
37        42       12.206556       12.206556       12.206556
37        43       11.180340       11.180340       11.180340
37        44       10.000000       10.000000       10.000000
37        45       10.440307       10.440307       10.440307
37        46       45.099889       45.099889       45.099889
37        47       47.265209       47.265209       47.265209
37        48       74.625733       74.625733       74.625733
37        49       83.240615       83.240615       83.240615
37        50       76.537572       76.537572       76.537572
37        51       50.487622       50.487622       50.487622
37        52       65.764732       65.764732       65.764732
37        53       68.007353       68.007353       68.007353
37        54       57.008771       57.008771       57.008771
37        55       26.925824       26.925824       26.925824
37        56       43.011626       43.011626       43.011626
37        57       52.201533       52.201533       52.201533
37        58       69.462220       69.462220       69.462220
37        59       90.138782       90.138782       90.138782
37        60       85.146932       85.146932       85.146932
37        61       55.901699       55.901699       55.901699
37        62       28.284271       28.284271       28.284271
37        63       50.000000       50.000000       50.000000
37        64       65.000000       65.000000       65.000000
37        65       58.523500       58.523500       58.523500
37        66       54.083269       54.083269       54.083269
37        67       53.665631       53.665631       53.665631
37        68       43.011626       43.011626       43.011626
37        69       35.355339       35.355339       35.355339
37        70       47.381431       47.381431       47.381431
37        71       34.000000       34.000000       34.000000
```

```
37         72        30.000000        30.000000        30.000000
37         73        20.099751        20.099751        20.099751
37         74        67.779053        67.779053        67.779053
37         75        79.056942        79.056942        79.056942
37         76        100.000000        100.000000        100.000000
37         77        73.171033        73.171033        73.171033
37         78        92.130342        92.130342        92.130342
37         79        63.953108        63.953108        63.953108
37         80        61.400326        61.400326        61.400326
37         81        42.047592        42.047592        42.047592
37         82        31.384710        31.384710        31.384710
37         83        56.639209        56.639209        56.639209
37         84        60.827625        60.827625        60.827625
37         85        56.568542        56.568542        56.568542
37         86        62.032250        62.032250        62.032250
37         87        75.213031        75.213031        75.213031
37         88        80.808415        80.808415        80.808415
37         89        49.091751        49.091751        49.091751
37         90        80.024996        80.024996        80.024996
37         91        47.201695        47.201695        47.201695
37         92        45.880279        45.880279        45.880279
37         93        43.680659        43.680659        43.680659
37         94        33.241540        33.241540        33.241540
37         95        37.854986        37.854986        37.854986
37         96        48.836462        48.836462        48.836462
37         97        32.572995        32.572995        32.572995
37         98        90.609050        90.609050        90.609050
37         99        51.088159        51.088159        51.088159
37         100        63.411355        63.411355        63.411355
37         101        38.470768        38.470768        38.470768
38         1        40.607881        40.607881        40.607881
38         2        40.112342        40.112342        40.112342
38         3        43.566042        43.566042        43.566042
38         4        43.104524        43.104524        43.104524
38         5        45.044423        45.044423        45.044423
38         6        45.099889        45.099889        45.099889
38         7        47.518417        47.518417        47.518417
38         8        50.487622        50.487622        50.487622
38         9        50.039984        50.039984        50.039984
38         10        72.346389        72.346389        72.346389
38         11        69.202601        69.202601        69.202601
38         12        70.802542        70.802542        70.802542
38         13        67.955868        67.955868        67.955868
38         14        76.216796        76.216796        76.216796
38         15        70.491134        70.491134        70.491134
38         16        75.716577        75.716577        75.716577
38         17        77.388630        77.388630        77.388630
38         18        74.793048        74.793048        74.793048
38         19        79.812280        79.812280        79.812280
38         20        75.584390        75.584390        75.584390
38         21        70.837843        70.837843        70.837843
38         22        80.956779        80.956779        80.956779
38         23        71.512237        71.512237        71.512237
38         24        81.596569        81.596569        81.596569
38         25        72.235725        72.235725        72.235725
38         26        82.637764        82.637764        82.637764
38         27        60.033324        60.033324        60.033324
38         28        55.758407        55.758407        55.758407
38         29        58.591808        58.591808        58.591808
38         30        53.235327        53.235327        53.235327
38         31        56.859476        56.859476        56.859476
38         32        52.325902        52.325902        52.325902
38         33        56.462377        56.462377        56.462377
```

```
38         34         60.406953         60.406953         60.406953
38         35         51.078371         51.078371         51.078371
38         36         3.605551        3.605551        3.605551
38         37         3.000000        3.000000        3.000000
38         39         3.605551        3.605551        3.605551
38         40         5.385165        5.385165        5.385165
38         41         5.830952        5.830952        5.830952
38         42         9.899495        9.899495        9.899495
38         43         10.198039         10.198039         10.198039
38         44         10.440307         10.440307         10.440307
38         45         10.000000         10.000000         10.000000
38         46         45.000000         45.000000         45.000000
38         47         47.042534         47.042534         47.042534
38         48         73.061618         73.061618         73.061618
38         49         80.361682         80.361682         80.361682
38         50         73.681748         73.681748         73.681748
38         51         47.518417         47.518417         47.518417
38         52         62.801274         62.801274         62.801274
38         53         65.604878         65.604878         65.604878
38         54         55.217751         55.217751         55.217751
38         55         24.166092         24.166092         24.166092
38         56         41.340053         41.340053         41.340053
38         57         49.335586         49.335586         49.335586
38         58         66.887966         66.887966         66.887966
38         59         87.658428         87.658428         87.658428
38         60         82.879430         82.879430         82.879430
38         61         54.626001         54.626001         54.626001
38         62         26.248809         26.248809         26.248809
38         63         47.000000         47.000000         47.000000
38         64         62.000000         62.000000         62.000000
38         65         55.713553         55.713553         55.713553
38         66         51.613952         51.613952         51.613952
38         67         51.000000         51.000000         51.000000
38         68         40.012498         40.012498         40.012498
38         69         33.301652         33.301652         33.301652
38         70         45.343136         45.343136         45.343136
38         71         32.695565         32.695565         32.695565
38         72         27.000000         27.000000         27.000000
38         73         17.117243         17.117243         17.117243
38         74         66.730802         66.730802         66.730802
38         75         76.609399         76.609399         76.609399
38         76         97.616597         97.616597         97.616597
38         77         70.178344         70.178344         70.178344
38         78         89.470666         89.470666         89.470666
38         79         62.649820         62.649820         62.649820
38         80         60.638272         60.638272         60.638272
38         81         39.357337         39.357337         39.357337
38         82         28.844410         28.844410         28.844410
38         83         54.451814         54.451814         54.451814
38         84         58.180753         58.180753         58.180753
38         85         53.600373         53.600373         53.600373
38         86         59.033889         59.033889         59.033889
38         87         72.801099         72.801099         72.801099
38         88         78.568442         78.568442         78.568442
38         89         47.507894         47.507894         47.507894
38         90         77.025970         77.025970         77.025970
38         91         44.821870         44.821870         44.821870
38         92         43.081318         43.081318         43.081318
38         93         40.804412         40.804412         40.804412
38         94         30.265492         30.265492         30.265492
38         95         34.928498         34.928498         34.928498
38         96         45.891176         45.891176         45.891176
38         97         29.732137         29.732137         29.732137
```

```
38        98        88.413800        88.413800        88.413800
38        99        49.203658        49.203658        49.203658
38        100        61.073726        61.073726        61.073726
38        101        37.161808        37.161808        37.161808
39        1        37.202150        37.202150        37.202150
39        2        37.336309        37.336309        37.336309
39        3        40.311289        40.311289        40.311289
39        4        40.311289        40.311289        40.311289
39        5        42.000000        42.000000        42.000000
39        6        42.296572        42.296572        42.296572
39        7        44.283180        44.283180        44.283180
39        8        47.265209        47.265209        47.265209
39        9        47.000000        47.000000        47.000000
39        10        68.767725        68.767725        68.767725
39        11        65.604878        65.604878        65.604878
39        12        67.201190        67.201190        67.201190
39        13        64.350602        64.350602        64.350602
39        14        72.622311        72.622311        72.622311
39        15        66.887966        66.887966        66.887966
39        16        72.111026        72.111026        72.111026
39        17        73.783467        73.783467        73.783467
39        18        71.196910        71.196910        71.196910
39        19        77.129761        77.129761        77.129761
39        20        72.801099        72.801099        72.801099
39        21        68.007353        68.007353        68.007353
39        22        78.160092        78.160092        78.160092
39        23        68.622154        68.622154        68.622154
39        24        78.746428        78.746428        78.746428
39        25        69.289249        69.289249        69.289249
39        26        79.711982        79.711982        79.711982
39        27        59.908263        59.908263        59.908263
39        28        55.803226        55.803226        55.803226
39        29        58.309519        58.309519        58.309519
39        30        53.000000        53.000000        53.000000
39        31        56.356011        56.356011        56.356011
39        32        51.971146        51.971146        51.971146
39        33        55.901699        55.901699        55.901699
39        34        59.615434        59.615434        59.615434
39        35        50.537115        50.537115        50.537115
39        36        7.071068        7.071068        7.071068
39        37        5.830952        5.830952        5.830952
39        38        3.605551        3.605551        3.605551
39        40        2.000000        2.000000        2.000000
39        41        5.385165        5.385165        5.385165
39        42        6.403124        6.403124        6.403124
39        43        7.000000        7.000000        7.000000
39        44        8.602325        8.602325        8.602325
39        45        7.280110        7.280110        7.280110
39        46        42.047592        42.047592        42.047592
39        47        44.000000        44.000000        44.000000
39        48        69.462220        69.462220        69.462220
39        49        77.620873        77.620873        77.620873
39        50        70.880181        70.880181        70.880181
39        51        46.097722        46.097722        46.097722
39        52        60.406953        60.406953        60.406953
39        53        62.201286        62.201286        62.201286
39        54        51.613952        51.613952        51.613952
39        55        21.189620        21.189620        21.189620
39        56        37.735925        37.735925        37.735925
39        57        46.572524        46.572524        46.572524
39        58        63.631753        63.631753        63.631753
39        59        84.314886        84.314886        84.314886
39        60        79.397733        79.397733        79.397733
```

```
39         61        51.078371        51.078371        51.078371
39         62        22.671568        22.671568        22.671568
39         63        45.099889        45.099889        45.099889
39         64        60.074953        60.074953        60.074953
39         65        52.810984        52.810984        52.810984
39         66        48.259714        48.259714        48.259714
39         67        47.853944        47.853944        47.853944
39         68        38.052595        38.052595        38.052595
39         69        29.732137        29.732137        29.732137
39         70        41.773197        41.773197        41.773197
39         71        29.154759        29.154759        29.154759
39         72        25.179357        25.179357        25.179357
39         73        15.033296        15.033296        15.033296
39         74        63.245553        63.245553        63.245553
39         75        73.239334        73.239334        73.239334
39         76        94.201911        94.201911        94.201911
39         77        68.029405        68.029405        68.029405
39         78        86.313383        86.313383        86.313383
39         79        59.093147        59.093147        59.093147
39         80        57.271284        57.271284        57.271284
39         81        36.249138        36.249138        36.249138
39         82        25.553865        25.553865        25.553865
39         83        50.931326        50.931326        50.931326
39         84        55.009090        55.009090        55.009090
39         85        51.244512        51.244512        51.244512
39         86        57.008771        57.008771        57.008771
39         87        69.404611        69.404611        69.404611
39         88        75.073298        75.073298        75.073298
39         89        43.908997        43.908997        43.908997
39         90        75.166482        75.166482        75.166482
39         91        41.400483        41.400483        41.400483
39         92        40.162171        40.162171        40.162171
39         93        38.078866        38.078866        38.078866
39         94        28.017851        28.017851        28.017851
39         95        32.388269        32.388269        32.388269
39         96        43.416587        43.416587        43.416587
39         97        26.925824        26.925824        26.925824
39         98        84.899941        84.899941        84.899941
39         99        45.607017        45.607017        45.607017
39         100        57.628118        57.628118        57.628118
39         101        33.615473        33.615473        33.615473
40         1        36.055513        36.055513        36.055513
40         2        35.355339        35.355339        35.355339
40         3        38.327536        38.327536        38.327536
40         4        38.327536        38.327536        38.327536
40         5        40.000000        40.000000        40.000000
40         6        40.311289        40.311289        40.311289
40         7        42.296572        42.296572        42.296572
40         8        45.276926        45.276926        45.276926
40         9        45.000000        45.000000        45.000000
40         10        67.268120        67.268120        67.268120
40         11        64.031242        64.031242        64.031242
40         12        65.604878        65.604878        65.604878
40         13        62.681736        62.681736        62.681736
40         14        71.063352        71.063352        71.063352
40         15        65.192024        65.192024        65.192024
40         16        70.455660        70.455660        70.455660
40         17        72.111026        72.111026        72.111026
40         18        69.462220        69.462220        69.462220
40         19        76.687678        76.687678        76.687678
40         20        72.277244        72.277244        72.277244
40         21        67.446275        67.446275        67.446275
40         22        77.620873        77.620873        77.620873
```

```
40      23      68.007353       68.007353       68.007353
40      24      78.160092       78.160092       78.160092
40      25      68.622154       68.622154       68.622154
40      26      79.056942       79.056942       79.056942
40      27      61.032778       61.032778       61.032778
40      28      57.008771       57.008771       57.008771
40      29      59.363288       59.363288       59.363288
40      30      54.083269       54.083269       54.083269
40      31      57.306195       57.306195       57.306195
40      32      53.000000       53.000000       53.000000
40      33      56.824291       56.824291       56.824291
40      34      60.415230       60.415230       60.415230
40      35      51.478151       51.478151       51.478151
40      36      8.602325        8.602325        8.602325
40      37      7.071068        7.071068        7.071068
40      38      5.385165        5.385165        5.385165
40      39      2.000000        2.000000        2.000000
40      41      5.000000        5.000000        5.000000
40      42      5.385165        5.385165        5.385165
40      43      5.000000        5.000000        5.000000
40      44      7.071068        7.071068        7.071068
40      45      5.385165        5.385165        5.385165
40      46      40.049969       40.049969       40.049969
40      47      42.000000       42.000000       42.000000
40      48      67.742158       67.742158       67.742158
40      49      77.129761       77.129761       77.129761
40      50      70.342022       70.342022       70.342022
40      51      46.572524       46.572524       46.572524
40      52      60.207973       60.207973       60.207973
40      53      61.032778       61.032778       61.032778
40      54      50.000000       50.000000       50.000000
40      55      20.615528       20.615528       20.615528
40      56      36.055513       36.055513       36.055513
40      57      46.097722       46.097722       46.097722
40      58      62.649820       62.649820       62.649820
40      59      83.216585       83.216585       83.216585
40      60      78.102497       78.102497       78.102497
40      61      49.244289       49.244289       49.244289
40      62      21.213203       21.213203       21.213203
40      63      45.276926       45.276926       45.276926
40      64      60.207973       60.207973       60.207973
40      65      52.201533       52.201533       52.201533
40      66      47.169906       47.169906       47.169906
40      67      47.010637       47.010637       47.010637
40      68      38.209946       38.209946       38.209946
40      69      28.284271       28.284271       28.284271
40      70      40.311289       40.311289       40.311289
40      71      27.313001       27.313001       27.313001
40      72      25.495098       25.495098       25.495098
40      73      15.297059       15.297059       15.297059
40      74      61.351447       61.351447       61.351447
40      75      72.111026       72.111026       72.111026
40      76      93.005376       93.005376       93.005376
40      77      68.000000       68.000000       68.000000
40      78      85.428333       85.428333       85.428333
40      79      57.271284       57.271284       57.271284
40      80      55.317267       55.317267       55.317267
40      81      35.468296       35.468296       35.468296
40      82      24.596748       24.596748       24.596748
40      83      49.578221       49.578221       49.578221
40      84      54.129474       54.129474       54.129474
40      85      51.088159       51.088159       51.088159
40      86      57.078893       57.078893       57.078893
```

```
40        87        68.242216        68.242216        68.242216
40        88        73.756356        73.756356        73.756356
40        89        42.190046        42.190046        42.190046
40        90        75.325958        75.325958        75.325958
40        91        40.224371        40.224371        40.224371
40        92        39.560081        39.560081        39.560081
40        93        37.656341        37.656341        37.656341
40        94        28.017851        28.017851        28.017851
40        95        32.140317        32.140317        32.140317
40        96        43.185646        43.185646        43.185646
40        97        26.476405        26.476405        26.476405
40        98        83.546394        83.546394        83.546394
40        99        44.045431        44.045431        44.045431
40        100        56.400355        56.400355        56.400355
40        101        31.780497        31.780497        31.780497
41        1        40.311289        40.311289        40.311289
41        2        35.000000        35.000000        35.000000
41        3        39.293765        39.293765        39.293765
41        4        38.000000        38.000000        38.000000
41        5        40.311289        40.311289        40.311289
41        6        40.000000        40.000000        40.000000
41        7        43.174066        43.174066        43.174066
41        8        46.097722        46.097722        46.097722
41        9        45.276926        45.276926        45.276926
41        10        70.710678        70.710678        70.710678
41        11        67.268120        67.268120        67.268120
41        12        68.767725        68.767725        68.767725
41        13        65.604878        65.604878        65.604878
41        14        74.330344        74.330344        74.330344
41        15        68.007353        68.007353        68.007353
41        16        73.409809        73.409809        73.409809
41        17        75.000000        75.000000        75.000000
41        18        72.111026        72.111026        72.111026
41        19        81.584312        81.584312        81.584312
41        20        77.129761        77.129761        77.129761
41        21        72.277244        72.277244        72.277244
41        22        82.462113        82.462113        82.462113
41        23        72.801099        72.801099        72.801099
41        24        82.969874        82.969874        82.969874
41        25        73.375745        73.375745        73.375745
41        26        83.815273        83.815273        83.815273
41        27        65.192024        65.192024        65.192024
41        28        61.032778        61.032778        61.032778
41        29        63.631753        63.631753        63.631753
41        30        58.309519        58.309519        58.309519
41        31        61.717096        61.717096        61.717096
41        32        57.306195        57.306195        57.306195
41        33        61.269895        61.269895        61.269895
41        34        65.000000        65.000000        65.000000
41        35        55.901699        55.901699        55.901699
41        36        7.000000        7.000000        7.000000
41        37        5.000000        5.000000        5.000000
41        38        5.830952        5.830952        5.830952
41        39        5.385165        5.385165        5.385165
41        40        5.000000        5.000000        5.000000
41        42        10.198039        10.198039        10.198039
41        43        7.071068        7.071068        7.071068
41        44        5.000000        5.000000        5.000000
41        45        5.830952        5.830952        5.830952
41        46        40.112342        40.112342        40.112342
41        47        42.296572        42.296572        42.296572
41        48        70.455660        70.455660        70.455660
41        49        82.000000        82.000000        82.000000
```

```
41         50         75.186435        75.186435        75.186435
41         51         51.419841        51.419841        51.419841
41         52         65.192024        65.192024        65.192024
41         53         65.192024        65.192024        65.192024
41         54         53.150729        53.150729        53.150729
41         55         25.495098        25.495098        25.495098
41         56         39.051248        39.051248        39.051248
41         57         50.990195        50.990195        50.990195
41         58         67.082039        67.082039        67.082039
41         59         87.464278        87.464278        87.464278
41         60         82.006097        82.006097        82.006097
41         61         51.478151        51.478151        51.478151
41         62         25.000000        25.000000        25.000000
41         63         50.249378        50.249378        50.249378
41         64         65.192024        65.192024        65.192024
41         65         57.008771        57.008771        57.008771
41         66         51.478151        51.478151        51.478151
41         67         51.623638        51.623638        51.623638
41         68         43.185646        43.185646        43.185646
41         69         32.015621        32.015621        32.015621
41         70         43.931765        43.931765        43.931765
41         71         29.681644        29.681644        29.681644
41         72         30.413813        30.413813        30.413813
41         73         20.223748        20.223748        20.223748
41         74         63.158531        63.158531        63.158531
41         75         76.321688        76.321688        76.321688
41         76         97.082439        97.082439        97.082439
41         77         73.000000        73.000000        73.000000
41         78         89.961103        89.961103        89.961103
41         79         59.539903        59.539903        59.539903
41         80         56.612719        56.612719        56.612719
41         81         40.162171        40.162171        40.162171
41         82         29.154759        29.154759        29.154759
41         83         53.413481        53.413481        53.413481
41         84         58.694122        58.694122        58.694122
41         85         56.080300        56.080300        56.080300
41         86         62.072538        62.072538        62.072538
41         87         72.401657        72.401657        72.401657
41         88         77.620873        77.620873        77.620873
41         89         45.000000        45.000000        45.000000
41         90         80.305666        80.305666        80.305666
41         91         44.418465        44.418465        44.418465
41         92         44.384682        44.384682        44.384682
41         93         42.579338        42.579338        42.579338
41         94         33.015148        33.015148        33.015148
41         95         37.121422        37.121422        37.121422
41         96         48.166378        48.166378        48.166378
41         97         31.400637        31.400637        31.400637
41         98         87.321246        87.321246        87.321246
41         99         47.381431        47.381431        47.381431
41         100         60.464866        60.464866        60.464866
41         101         34.132096        34.132096        34.132096
42         1         30.805844        30.805844        30.805844
42         2         34.481879        34.481879        34.481879
42         3         36.000000        36.000000        36.000000
42         4         37.363083        37.363083        37.363083
42         5         38.327536        38.327536        38.327536
42         6         39.293765        39.293765        39.293765
42         7         40.000000        40.000000        40.000000
42         8         43.000000        43.000000        43.000000
42         9         43.289722        43.289722        43.289722
42         10         62.481997        62.481997        62.481997
42         11         59.405387        59.405387        59.405387
```

```
42          12          61.032778          61.032778          61.032778
42          13          58.309519          58.309519          58.309519
42          14          66.400301          66.400301          66.400301
42          15          60.901560          60.901560          60.901560
42          16          66.037868          66.037868          66.037868
42          17          67.742158          67.742158          67.742158
42          18          65.299311          65.299311          65.299311
42          19          71.386273          71.386273          71.386273
42          20          66.940272          66.940272          66.940272
42          21          62.096699          62.096699          62.096699
42          22          72.277244          72.277244          72.277244
42          23          62.641839          62.641839          62.641839
42          24          72.801099          72.801099          72.801099
42          25          63.245553          63.245553          63.245553
42          26          73.681748          73.681748          73.681748
42          27          58.258047          58.258047          58.258047
42          28          54.488531          54.488531          54.488531
42          29          56.400355          56.400355          56.400355
42          30          51.224994          51.224994          51.224994
42          31          54.083269          54.083269          54.083269
42          32          50.000000          50.000000          50.000000
42          33          53.535035          53.535035          53.535035
42          34          56.824291          56.824291          56.824291
42          35          48.259714          48.259714          48.259714
42          36          13.453624          13.453624          13.453624
42          37          12.206556          12.206556          12.206556
42          38          9.899495          9.899495          9.899495
42          39          6.403124          6.403124          6.403124
42          40          5.385165          5.385165          5.385165
42          41          10.198039          10.198039          10.198039
42          43          5.830952          5.830952          5.830952
42          44          10.440307          10.440307          10.440307
42          45          7.615773          7.615773          7.615773
42          46          38.639358          38.639358          38.639358
42          47          40.311289          40.311289          40.311289
42          48          63.529521          63.529521          63.529521
42          49          71.805292          71.805292          71.805292
42          50          65.000000          65.000000          65.000000
42          51          42.379240          42.379240          42.379240
42          52          55.081757          55.081757          55.081757
42          53          55.803226          55.803226          55.803226
42          54          45.486262          45.486262          45.486262
42          55          15.297059          15.297059          15.297059
42          56          31.764760          31.764760          31.764760
42          57          40.792156          40.792156          40.792156
42          58          57.306195          57.306195          57.306195
42          59          77.935871          77.935871          77.935871
42          60          73.000000          73.000000          73.000000
42          61          45.541190          45.541190          45.541190
42          62          16.401219          16.401219          16.401219
42          63          40.607881          40.607881          40.607881
42          64          55.443665          55.443665          55.443665
42          65          46.840154          46.840154          46.840154
42          66          41.880783          41.880783          41.880783
42          67          41.629317          41.629317          41.629317
42          68          33.541020          33.541020          33.541020
42          69          23.430749          23.430749          23.430749
42          70          35.468296          35.468296          35.468296
42          71          23.769729          23.769729          23.769729
42          72          21.189620          21.189620          21.189620
42          73          11.180340          11.180340          11.180340
42          74          57.974132          57.974132          57.974132
42          75          66.850580          66.850580          66.850580
42          76          87.800911          87.800911          87.800911
42          77          63.031738          63.031738          63.031738
42          78          80.056230          80.056230          80.056230
42          79          53.488316          53.488316          53.488316
42          80          52.469038          52.469038          52.469038
42          81          30.083218          30.083218          30.083218
42          82          19.235384          19.235384          19.235384
```

See Also

- http://en.wikipedia.org/wiki/Vehicle_routing_problem

Indices and tables

- genindex
- search

Graph Operations

Transformation - Family of functions - Maps a given graph to a new form.

- *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.

6.2.7 pgr_lineGraph - Experimental

`pgr_lineGraph` — Transforms a given graph into its corresponding edge-based graph.

[28]

Fig. 6.25: Boost Graph Inside

Warning: Experimental functions

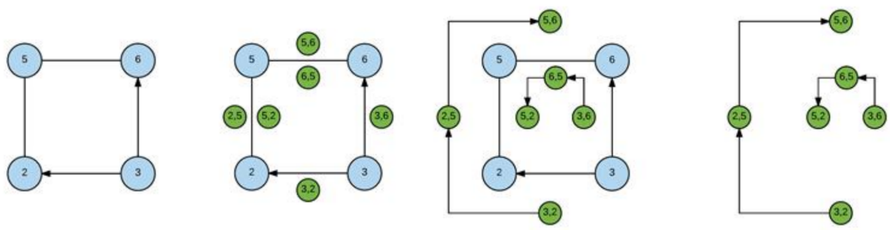
- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

Given a graph G, its line graph L(G) is a graph such that:-

- each vertex of L(G) represents an edge of G
- two vertices of L(G) are adjacent if and only if their corresponding edges share a common endpoint in G.

The following figures show a graph (left, with blue vertices) and its Line Graph (right, with green vertices).



Signature Summary

```
pgr_lineGraph(edges_sql, directed)
RETURNS SET OF (seq, source, target, cost, reverse_cost)
    OR EMPTY SET
```

Signatures

Minimal signature

```
pgr_lineGraph(edges_sql)
RETURNS SET OF (seq, source, target, cost, reverse_cost) or EMPTY SET
```

The minimal signature is for a **directed** graph:

Example

```
SELECT * FROM pgr_lineGraph(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table'
);
 seq | source | target | cost | reverse_cost
-----+--------+--------+------+--------------
   1 |    -18 |     18 |    1 |            1
   2 |    -17 |     17 |    1 |            1
   3 |    -16 |     -3 |    1 |           -1
   4 |    -16 |     16 |    1 |            1
   5 |    -15 |     -9 |    1 |            1
   6 |    -15 |     15 |    1 |            1
   7 |    -14 |    -10 |    1 |            1
   8 |    -14 |     12 |    1 |           -1
   9 |    -14 |     14 |    1 |            1
  10 |    -10 |     -7 |    1 |            1
  11 |    -10 |     -4 |    1 |            1
  12 |    -10 |      8 |    1 |            1
  13 |    -10 |     10 |    1 |            1
  14 |     -9 |     -8 |    1 |            1
  15 |     -9 |      9 |    1 |            1
  16 |     -9 |     11 |    1 |           -1
  17 |     -8 |     -7 |    1 |            1
  18 |     -8 |     -4 |    1 |            1
  19 |     -8 |      8 |    1 |            1
  20 |     -7 |     -6 |    1 |            1
  21 |     -6 |      6 |    1 |            1
  22 |     -4 |     -1 |    1 |            1
  23 |     -4 |      4 |    1 |            1
  24 |     -3 |     -2 |    1 |           -1
  25 |     -3 |      5 |    1 |           -1
  26 |     -2 |     -1 |    1 |           -1
  27 |     -2 |      4 |    1 |           -1
  28 |     -1 |      1 |    1 |            1
  29 |      5 |     -8 |    1 |           -1
  30 |      5 |      9 |    1 |           -1
  31 |      5 |     11 |    1 |           -1
  32 |      7 |     -7 |    1 |            1
  33 |      7 |     -4 |    1 |            1
  34 |      8 |     11 |    1 |           -1
  35 |     10 |     12 |    1 |           -1
  36 |     11 |     13 |    1 |           -1
  37 |     12 |     13 |    1 |           -1
  38 |     13 |    -15 |    1 |           -1
  39 |     16 |     -9 |    1 |            1
  40 |     16 |     15 |    1 |            1
(40 rows)
```

Complete Signature

```
pgr_lineGraph(edges_sql, directed);
RETURNS SET OF (seq, source, target, cost, reverse_cost) or EMPTY SET
```

This signature returns the Line Graph of the current graph:

- on a **directed** graph when `directed` flag is missing or is set to `true`.
- on an **undirected** graph when `directed` flag is set to `false`.

Example

```
SELECT * FROM pgr_lineGraph(
    'SELECT id, source, target, cost, reverse_cost FROM edge_table',
    FALSE
);
 seq | source | target | cost | reverse_cost
-----+--------+--------+------+--------------
   1 |     -3 |     -2 |    1 |           -1
   2 |     -3 |      5 |    1 |           -1
   3 |     -2 |      4 |    1 |           -1
   4 |      1 |      4 |    1 |           -1
   5 |      4 |      8 |    1 |           -1
   6 |      4 |     10 |    1 |           -1
   7 |      5 |      9 |    1 |           -1
   8 |      5 |     11 |    1 |           -1
   9 |      6 |      7 |    1 |           -1
  10 |      7 |      8 |    1 |           -1
  11 |      7 |     10 |    1 |           -1
  12 |      8 |      9 |    1 |           -1
  13 |      8 |     11 |    1 |           -1
  14 |      9 |     15 |    1 |           -1
  15 |     10 |     12 |    1 |           -1
  16 |     10 |     14 |    1 |           -1
  17 |     11 |     13 |    1 |           -1
  18 |     12 |     13 |    1 |           -1
  19 |     16 |     15 |    1 |           -1
(19 rows)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
directed	`BOOLEAN`	• When `true` the graph is considered as *Directed*. • When `false` the graph is considered as *Undirected*.

Description of the return values

RETURNS SETOF (seq, source, target, cost, reverse_cost)

Column	Type	Description
seq	INTEGER	Sequential value starting from **1**.
source	BIGINT	Identifier of the source vertex of the current edge *id*. • When *negative*: the source is the reverse edge in the original graph.
target	BIGINT	Identifier of the target vertex of the current edge *id*. • When *negative*: the target is the reverse edge in the original graph.
cost	FLOAT	Weight of the edge (*source*, *target*). • When *negative*: edge (*source*, *target*) does not exist, therefore it's not part of the graph.
reverse_cost	FLOAT	Weight of the edge (*target*, *source*). • When *negative*: edge (*target*, *source*) does not exist, therefore it's not part of the graph.

See Also

- https://en.wikipedia.org/wiki/Line_graph
- The queries use the Sample Data network.

Indices and tables

- genindex
- search

6.2.8 pgr_lineGraphFull - Experimental

`pgr_lineGraphFull` — Transforms a given graph into a new graph where all of the vertices from the original graph are converted to line graphs.

Warning: Experimental functions

- They are not officially of the current release.
- They likely will not be officially be part of the next release:
  - The functions might not make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might change.
  - Signature might change.
  - Functionality might change.
  - pgTap tests might be missing.
  - Might need c/c++ coding.
  - May lack documentation.
  - Documentation if any might need to be rewritten.
  - Documentation examples might need to be automatically generated.
  - Might need a lot of feedback from the comunity.
  - Might depend on a proposed function of pgRouting
  - Might depend on a deprecated function of pgRouting

Synopsis

pgr_lineGraphFull, converts original directed graph to a directed line graph by converting each vertex to a complete graph and keeping all the original edges. The new connecting edges have a cost 0 and go between the adjacent original edges, respecting the directionality.

A possible application of the resulting graph is "routing with two edge restrictions":

- Setting a cost of using the vertex when routing between edges on the connecting edge
- Forbid the routing between two edges by removing the connecting edge

This is possible because each of the intersections (vertices) in the original graph are now complete graphs that have a new edge for each possible turn across that intersection.

Characteristics

The main characteristics are:

- This function is for directed graphs.
- Results are undefined when a negative vertex id is used in the input graph.
- Results are undefined when a duplicated edge id is used in the input graph.
- Running time: TBD

Signature Summary

```
pgr_lineGraphFull(edges_sql)
RETURNS SET OF (seq, source, target, cost, edge)
    OR EMPTY SET
```

Signatures

Minimal signature

```
pgr_lineGraphFull(TEXT edges_sql)
RETURNS SET OF (seq, source, target, cost, edge) OR EMPTY SET
```

Example

```
SELECT * FROM pgr_lineGraphFull(
    'SELECT id, source, target, cost, reverse_cost
      FROM edge_table
      WHERE id IN (4,7,8,10)'
);
 seq | source | target | cost | edge
-----+--------+--------+------+------
   1 |     -1 |      5 |    1 |    4
   2 |      2 |     -1 |    0 |    0
   3 |     -2 |      2 |    1 |   -4
   4 |     -3 |      8 |    1 |   -7
   5 |     -4 |      6 |    1 |    8
   6 |     -5 |     10 |    1 |   10
   7 |      5 |     -2 |    0 |    0
   8 |      5 |     -3 |    0 |    0
   9 |      5 |     -4 |    0 |    0
  10 |      5 |     -5 |    0 |    0
  11 |     -6 |     -2 |    0 |    0
  12 |     -6 |     -3 |    0 |    0
  13 |     -6 |     -4 |    0 |    0
  14 |     -6 |     -5 |    0 |    0
  15 |     -7 |     -2 |    0 |    0
  16 |     -7 |     -3 |    0 |    0
  17 |     -7 |     -4 |    0 |    0
  18 |     -7 |     -5 |    0 |    0
  19 |     -8 |     -2 |    0 |    0
  20 |     -8 |     -3 |    0 |    0
  21 |     -8 |     -4 |    0 |    0
  22 |     -8 |     -5 |    0 |    0
  23 |     -9 |     -6 |    1 |    7
  24 |      8 |     -9 |    0 |    0
  25 |    -10 |     -7 |    1 |   -8
  26 |      6 |    -10 |    0 |    0
  27 |    -11 |     -8 |    1 |  -10
  28 |     10 |    -11 |    0 |    0
(28 rows)
```

Description of the Signatures

Description of the edges_sql query for dijkstra like functions

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

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

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Default	Description
sql	`TEXT`		SQL query as described above.

Additional Examples

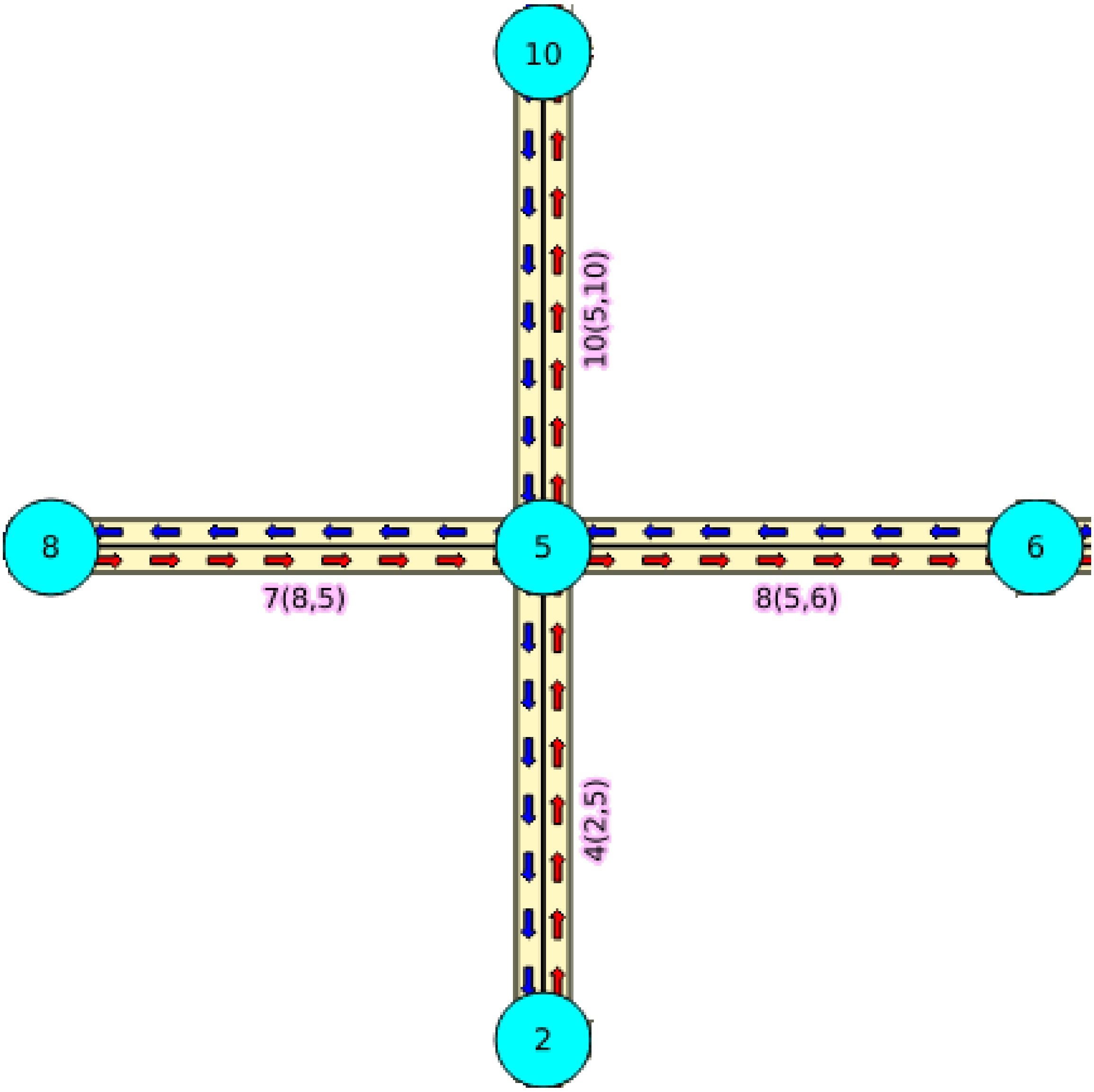
The examples of this section are based on the Sample Data network.

The examples include the subgraph including edges 4, 7, 8, and 10 with reverse_cost.

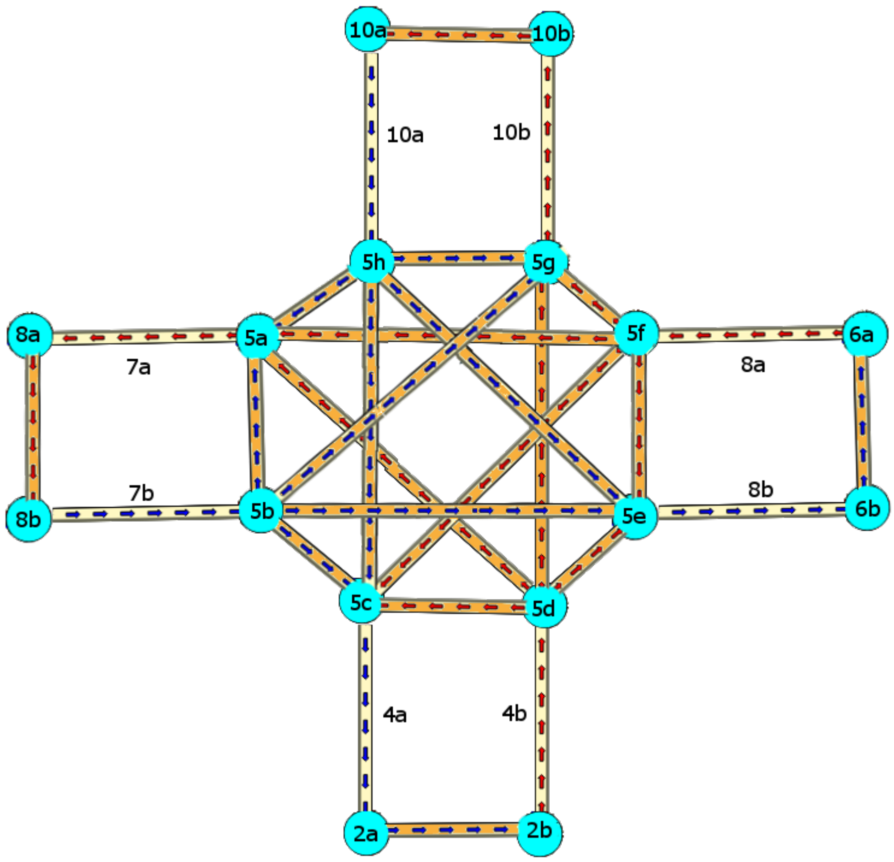
Example for generating the LineGraphFull

This example displays how this graph transformation works to create additional edges for each possible turn in a graph.

```
SELECT id, source, target, cost, reverse_cost
  FROM edge_table
  WHERE id IN (4,7,8,10);
```




```
SELECT * FROM pgr_lineGraphFull('SELECT id,
                                        source,
                                        target,
                                        cost,
                                        reverse_cost
                                   FROM edge_table
                                     WHERE id IN (4,7,8,10)');
```



In the transformed graph, all of the edges from the original graph are still present (yellow), but we now have additional edges for every turn that could be made across vertex 6 (orange).

Example for creating table that identifies transformed vertices

The vertices in the transformed graph are each created by splitting up the vertices in the original graph. Unless a vertex in the original graph is a leaf vertex, it will generate more than one vertex in the transformed graph. One of the newly created vertices in the transformed graph will be given the same vertex-id as the vertex that it was created from in the original graph, but the rest of the newly created vertices will have negative vertex ids. Following is an example of how to generate a table that maps the ids of the newly created vertices with the original vertex that they were created from

The first step is to store your results graph into a table and then create the vertex mapping table with one row for each distinct vertex id in the results graph.

```
CREATE TABLE lineGraph_edges AS SELECT * FROM pgr_lineGraphFull(
    $$SELECT id, source, target, cost, reverse_cost
    FROM edge_table WHERE id IN (4,7,8,10)$$
);
```

```
SELECT 28
CREATE TABLE lineGraph_vertices AS
SELECT *, NULL::BIGINT AS original_id
FROM (SELECT source AS id FROM lineGraph_edges
    UNION
    SELECT target FROM lineGraph_edges) as foo
ORDER BY id;
SELECT 16
```

Next, we set the original_id of all of the vertices in the results graph that were given the same vertex id as the vertex that it was created from in the original graph.

```
UPDATE lineGraph_vertices AS r
  SET original_id = v.id
  FROM edge_table_vertices_pgr AS v
  WHERE v.id = r.id;
UPDATE 5
```

Then, we cross reference all of the other newly created vertices that do not have the same original_id and set thier original_id values.

```
WITH
unassignedVertices
  AS (SELECT e.id, e.original_id
       FROM lineGraph_vertices AS e
       WHERE original_id IS NOT NULL),
edgesWithUnassignedSource
  AS (SELECT *
        FROM lineGraph_edges
        WHERE cost = 0 and source IN (SELECT id FROM unassignedVertices)),
edgesWithUnassignedSourcePlusVertices
  AS (SELECT *
        FROM edgesWithUnassignedSource
        JOIN lineGraph_vertices
        ON(source = id)),
verticesFromEdgesWithUnassignedSource
  AS (SELECT DISTINCT edgesWithUnassignedSourcePlusVertices.target,
                      edgesWithUnassignedSourcePlusVertices.original_id
        FROM edgesWithUnassignedSourcePlusVertices
        JOIN lineGraph_vertices AS r
        ON(target = r.id AND r.original_id IS NULL))
UPDATE lineGraph_vertices
  SET original_id = verticesFromEdgesWithUnassignedSource.original_id
  FROM verticesFromEdgesWithUnassignedSource
  WHERE verticesFromEdgesWithUnassignedSource.target = id;
UPDATE 8
WITH
unassignedVertices
  AS (SELECT e.id, e.original_id
        FROM lineGraph_vertices AS e
        WHERE original_id IS NOT NULL),
edgesWithUnassignedTarget
  AS (SELECT *
        FROM lineGraph_edges
        WHERE cost = 0 and target IN (SELECT id FROM unassignedVertices)),
edgesWithUnassignedTargetPlusVertices
  AS (SELECT *
        FROM edgesWithUnassignedTarget
        JOIN lineGraph_vertices
        ON(target = id)),
verticesFromEdgesWithUnassignedTarget
  AS (SELECT DISTINCT edgesWithUnassignedTargetPlusVertices.source,
                     edgesWithUnassignedTargetPlusVertices.original_id
```

```
        FROM edgesWithUnassignedTargetPlusVertices
        JOIN lineGraph_vertices AS r
        ON(source = r.id AND r.original_id IS NULL))
UPDATE lineGraph_vertices
  SET original_id = verticesFromEdgesWithUnassignedTarget.original_id
  FROM verticesFromEdgesWithUnassignedTarget
  WHERE verticesFromEdgesWithUnassignedTarget.source = id;
UPDATE 3
```

The only vertices left that have not been mapped are a few of the leaf vertices from the original graph. The following sql completes the mapping for these leaf vertices (in the case of this example graph there are no leaf vertices but this is nessessary for larger graphs).

```
WITH
unassignedVertexIds
  AS (SELECT id
        FROM lineGraph_vertices
        WHERE original_id IS NULL),
edgesWithUnassignedSource
  AS (SELECT source,edge
        FROM lineGraph_edges
        WHERE source IN (SELECT id FROM unassignedVertexIds)),
originalEdgesWithUnassignedSource
  AS (SELECT id,source
        FROM edge_table
        WHERE id IN (SELECT edge FROM edgesWithUnassignedSource))
UPDATE lineGraph_vertices AS d
  SET original_id = (SELECT source
                       FROM originalEdgesWithUnassignedSource
                       WHERE originalEdgesWithUnassignedSource.id =
                         (SELECT edge
                            FROM edgesWithUnassignedSource
                            WHERE edgesWithUnassignedSource.source = d.id))
  WHERE id IN (SELECT id FROM unassignedVertexIds);
UPDATE 0
WITH
unassignedVertexIds
  AS (SELECT id
        FROM lineGraph_vertices
        WHERE original_id IS NULL),
edgesWithUnassignedTarget
  AS (SELECT target,edge
        FROM lineGraph_edges
        WHERE target IN (SELECT id FROM unassignedVertexIds)),
originalEdgesWithUnassignedTarget
  AS (SELECT id,target
        FROM edge_table
        WHERE id IN (SELECT edge FROM edgesWithUnassignedTarget))
UPDATE lineGraph_vertices AS d
  SET original_id = (SELECT target
                       FROM originalEdgesWithUnassignedTarget
                       WHERE originalEdgesWithUnassignedTarget.id =
                         (SELECT edge
                            FROM edgesWithUnassignedTarget
                            WHERE edgesWithUnassignedTarget.target = d.id))
  WHERE id IN (SELECT id FROM unassignedVertexIds);
UPDATE 0
```

Now our vertex mapping table is complete:

```
SELECT * FROM lineGraph_vertices;
 id  | original_id
-----+-------------
```

```
   2 |           2
   5 |           5
   6 |           6
   8 |           8
  10 |          10
 -11 |          10
 -10 |           6
  -9 |           8
  -5 |           5
  -4 |           5
  -3 |           5
  -2 |           5
  -1 |           2
  -8 |           5
  -7 |           5
  -6 |           5
(16 rows)
```

Example for running a dijkstra's shortest path with turn penalties

One use case for this graph transformation is to be able to run a shortest path search that takes into account the cost or limitation of turning. Below is an example of running a dijkstra's shortest path from vertex 2 to vertex 8 in the original graph, while adding a turn penalty cost of 100 to the turn from edge 4 to edge -7.

First we must increase set the cost of making the turn to 100:

```
UPDATE lineGraph_edges
  SET cost = 100
  WHERE source IN (SELECT target
                     FROM lineGraph_edges
                     WHERE edge = 4) AND target IN (SELECT source
                                                      FROM lineGraph_edges
                                                      WHERE edge = -7);
UPDATE 1
```

Then we must run a dijkstra's shortest path search using all of the vertices in the new graph that were created from vertex 2 as the starting point, and all of the vertices in the new graph that were created from vertex 8 as the ending point.

```
SELECT * FROM
  (SELECT * FROM
    (SELECT * FROM pgr_dijkstra($$SELECT seq AS id, * FROM lineGraph_edges$$,
      (SELECT array_agg(id) FROM lineGraph_vertices where original_id = 2),
      (SELECT array_agg(id) FROM lineGraph_vertices where original_id = 8)
    )) as shortestPaths
  WHERE start_vid = 2 AND end_vid = 8 AND (cost != 0 OR edge = -1)) as b;
 seq | path_seq | start_vid | end_vid | node | edge | cost | agg_cost
-----+----------+-----------+---------+------+------+------+----------
  29 |        2 |         2 |       8 |   -1 |    1 |    1 |        0
  31 |        4 |         2 |       8 |   -4 |    5 |    1 |        1
  33 |        6 |         2 |       8 |  -10 |   25 |    1 |        2
  35 |        8 |         2 |       8 |   -3 |    4 |    1 |        3
  36 |        9 |         2 |       8 |    8 |   -1 |    0 |        4
(5 rows)
```

Normally the shortest path from vertex 2 to vertex 8 would have an aggregate cost of 2, but since there is a large penalty for making the turn needed to get this cost, the route goes through vertex 6 to avoid this turn.

If you cross reference the node column in the dijkstra results with the vertex id mapping table, this will show you that the path goes from v2 -> v5 -> v6 -> v5 -> v8 in the original graph.

See Also

- http://en.wikipedia.org/wiki/Line_graph
- http://en.wikipedia.org/wiki/Complete_graph

Indices and tables

- genindex
- search

6.2.9 Transformation - Family of functions

- *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.

Introduction

This family of functions is used for transforming a given input graph $G(V, E)$ into a new graph $G'(V', E')$.

See Also

Indices and tables

- genindex
- search

6.2.10 See Also

Indices and tables

- genindex
- search

CHAPTER 7

Change Log

7.1 Release Notes

To see the full list of changes check the list of Git commits[1] on Github.

Table of contents

[1] https://github.com/pgRouting/pgrouting/commits

7.1.1 pgRouting 2.6.0 Release Notes

To see the issues closed by this release see the Git closed milestone for 2.6.0[2] 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

[2]https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=milestone%3A%22Release%202.6.0%22%20

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

7.1.2 pgRouting 2.5.3 Release Notes

To see the issues closed by this release see the Git closed milestone for 2.5.3[3] on Github.

Bug fixes

- Fix for postgresql 11: Removed a compilation error when compiling with postgreSQL

7.1.3 pgRouting 2.5.2 Release Notes

To see the issues closed by this release see the Git closed milestone for 2.5.2[4] on Github.

Bug fixes

- Fix for postgresql 10.1: Removed a compiler condition

7.1.4 pgRouting 2.5.1 Release Notes

To see the issues closed by this release see the Git closed milestone for 2.5.1[5] on Github.

Bug fixes

- Fixed prerequisite minimum version of: cmake

7.1.5 pgRouting 2.5.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.5.0[6] on Github.

enhancement:

- pgr_version is now on SQL language

[3] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=milestone%3A%22Release%202.5.3%22%20
[4] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=milestone%3A%22Release%202.5.2%22%20
[5] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=milestone%3A%22Release%202.5.1%22%20
[6] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.5.0%22+is%3Aclosed

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

7.1.6 pgRouting 2.4.2 Release Notes

To see the issues closed by this release see the Git closed milestone for 2.4.2[7] on Github.

Improvement

- Works for postgreSQL 10

Bug fixes

- Fixed: Unexpected error column "cname"
- Replace __linux__ with __GLIBC__ for glibc-specific headers and functions

7.1.7 pgRouting 2.4.1 Release Notes

To see the issues closed by this release see the Git closed milestone for 2.4.1[8] on Github.

Bug fixes

- Fixed compiling error on macOS
- Condition error on pgr_withPoints

7.1.8 pgRouting 2.4.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.4.0[9] on Github.

New Signatures

- pgr_bdDijkstra

[7] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=milestone%3A%22Release%202.4.2%22%20
[8] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=milestone%3A%22Release%202.4.1%22%20
[9] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.4.0%22+is%3Aclosed

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

7.1.9 pgRouting 2.3.2 Release Notes

To see the issues closed by this release see the Git closed issues for 2.3.2[10] 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

7.1.10 pgRouting 2.3.1 Release Notes

To see the issues closed by this release see the Git closed issues for 2.3.1[11] on Github.

[10] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.3.2%22+is%3Aclosed
[11] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.3.1%22+is%3Aclosed

Bug Fixes

- Leaks on proposed max_flow functions
- Regression error on pgr_trsp
- Types discrepancy on pgr_createVerticesTable

7.1.11 pgRouting 2.3.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.3.0[12] on Github.

New Signatures

- pgr_TSP
- pgr_aStar

New Functions

- pgr_eucledianTSP

New Proposed functions

- pgr_dijkstraCostMatrix
- pgr_withPointsCostMatrix
- pgr_maxFlowPushRelabel(one to one)
- pgr_maxFlowPushRelabel(one to many)
- pgr_maxFlowPushRelabel(many to one)
- pgr_maxFlowPushRelabel(many to many)
- pgr_maxFlowEdmondsKarp(one to one)
- pgr_maxFlowEdmondsKarp(one to many)
- pgr_maxFlowEdmondsKarp(many to one)
- pgr_maxFlowEdmondsKarp(many to many)
- pgr_maxFlowBoykovKolmogorov (one to one)
- pgr_maxFlowBoykovKolmogorov (one to many)
- pgr_maxFlowBoykovKolmogorov (many to one)
- pgr_maxFlowBoykovKolmogorov (many to many)
- pgr_maximumCardinalityMatching
- pgr_edgeDisjointPaths(one to one)
- pgr_edgeDisjointPaths(one to many)
- pgr_edgeDisjointPaths(many to one)
- pgr_edgeDisjointPaths(many to many)
- pgr_contractGraph

[12] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.3.0%22+is%3Aclosed

Deprecated Signatures

- pgr_tsp - use pgr_TSP or pgr_eucledianTSP instead
- pgr_astar - use pgr_aStar instead

Deprecated Functions

- pgr_flip_edges
- pgr_vidsToDmatrix
- pgr_pointsToDMatrix
- pgr_textToPoints

7.1.12 pgRouting 2.2.4 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.4[13] on Github.

Bug Fixes

- Bogus uses of extern "C"
- Build error on Fedora 24 + GCC 6.0
- Regression error pgr_nodeNetwork

7.1.13 pgRouting 2.2.3 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.3[14] on Github.

Bug Fixes

- Fixed compatibility issues with PostgreSQL 9.6.

7.1.14 pgRouting 2.2.2 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.2[15] on Github.

Bug Fixes

- Fixed regression error on pgr_drivingDistance

7.1.15 pgRouting 2.2.1 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.1[16] on Github.

[13] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.2.4%22+is%3Aclosed
[14] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.2.3%22+is%3Aclosed
[15] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.2.2%22+is%3Aclosed
[16] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A2.2.1+is%3Aclosed

Bug Fixes

- Server crash fix on pgr_alphaShape
- Bug fix on With Points family of functions

7.1.16 pgRouting 2.2.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.0[17] 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

[17] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.2.0%22+is%3Aclosed

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

7.1.17 pgRouting 2.1.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.1.0[18] on Github.

New Signatures

- pgr_dijkstra(one to many)
- pgr_dijkstra(many to one)
- pgr_dijkstra(many to many)
- pgr_drivingDistance(multiple vertices)

Refactored

- pgr_dijkstra(one to one)
- pgr_ksp
- pgr_drivingDistance(single vertex)

Improvements

- pgr_alphaShape function now can generate better (multi)polygon with holes and alpha parameter.

Proposed functionality

- Proposed functions from Steve Woodbridge, (Classified as Convenience by the author.)
  – pgr_pointToEdgeNode - convert a point geometry to a vertex_id based on closest edge.
  – pgr_flipEdges - flip the edges in an array of geometries so the connect end to end.
  – pgr_textToPoints - convert a string of x,y;x,y;... locations into point geometries.
  – pgr_pointsToVids - convert an array of point geometries into vertex ids.
  – pgr_pointsToDMatrix - Create a distance matrix from an array of points.
  – pgr_vidsToDMatrix - Create a distance matrix from an array of vertix_id.
  – pgr_vidsToDMatrix - Create a distance matrix from an array of vertix_id.
- Added proposed functions from GSoc Projects:

[18] https://github.com/pgRouting/pgrouting/issues?q=is%3Aissue+milestone%3A%22Release+2.1.0%22+is%3Aclosed

  - pgr_vrppdtw
  - pgr_vrponedepot

Deprecated functions

- pgr_getColumnName
- pgr_getTableName
- pgr_isColumnCndexed
- pgr_isColumnInTable
- pgr_quote_ident
- pgr_versionless
- pgr_startPoint
- pgr_endPoint
- pgr_pointToId

No longer supported

- Removed the 1.x legacy functions

Bug Fixes

- Some bug fixes in other functions

Refactoring Internal Code

- A C and C++ library for developer was created
  - encapsulates postgreSQL related functions
  - encapsulates Boost.Graph graphs
    * Directed Boost.Graph
    * Undirected Boost.graph.
  - allow any-integer in the id's
  - allow any-numerical on the cost/reverse_cost columns
- Instead of generating many libraries: - All functions are encapsulated in one library - The library has the prefix 2-1-0

7.1.18 pgRouting 2.0.1 Release Notes

Minor bug fixes.

Bug Fixes

- No track of the bug fixes were kept.

7.1.19 pgRouting 2.0.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.0.0[19] 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: http://docs.pgrouting.org

[19] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.0.0%22+is%3Aclosed

- shooting_star is discontinued

7.1.20 pgRouting 1.x Release Notes

To see the issues closed by this release see the Git closed issues for 1.x[20] 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

[20] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+1.x%22+is%3Aclosed

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

- genindex
- search

[C001] Simulated annealing algorithm for beginners[21]

[21] http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6