pgRouting Manual

Release 2.3.2 (master)

pgRouting Contributors

July 22, 2017

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

This is the manual for pgRouting 2.3.2 (master).

[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 an OSGeo Labs[7] project of the OSGeo Foundation[8] and included on OSGeo Live[9].

1.1.1 License

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] 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_Labs
[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/

1.1.2 Contributors

This Release Contributors

Individuals (in alphabetical order)

Andrea Nardellli, Daniel Kastl, Ko Nagase, Mario Basa, Regina Obe, Rohith Reddy, Stephen Woodbridge, Virginia Vergara

And all the people that gives 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]

Contributors Past & Present:

Individuals (in alphabetical order)

Akio Takubo, Andrea Nardelli, 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, Mohamed Zia, Mukul Priya, Razequl Islam, Regina Obe, Rohith Reddy, Sarthak Agarwal, Stephen Woodbridge, Sylvain Housseman, Sylvain Pasche, 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

[17]

Fig. 1.1: Boost Graph Inside

1.1.3 Inside

1.1.4 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.

1.2 Installation

This is a basic guide to download and install pgRouting.

The specific instructions for any given OS distribution may vary depending on the various package maintainers. Contact the specific OS package maintainer for details.

Note: The following are only general instructions.

Additional notes and corrections can be found in Installation wiki[18]

Also PostGIS provides some information about installation in this Install Guide[19]

1.2.1 Download

Binary packages are provided for the current version on the following platforms:

Windows

Winnie Bot Builds:

- Winnie Bot Builds[20]

Production Builds:

- Production builds are part of the Spatial Extensions/PostGIS Bundle available via Application StackBuilder
- Can also get PostGIS Bundle from http://download.osgeo.org/postgis/windows/

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

Ubuntu

pgRouting on Ubuntu can be installed using packages from a PostgreSQL repository:

Using a terminal window:

```
# Create /etc/apt/sources.list.d/pgdg.list. The distributions are called codename-pgdg.
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release -cs)-pgdg main" > /e

# Import the repository key, update the package lists
sudo apt-get install wget ca-certificates
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
sudo apt-get update

# Install pgrouting based on your postgres Installation: for this example is 9.3
sudo apt-get install postgresql-9.3-pgrouting
```

- To be up-to-date with changes and improvements

```
sudo apt-get update & sudo apt-get upgrade
```

RHEL/CentOS

- Add repositories for dependencies:

```
    wget http://repo.enetres.net/enetres.repo -O /etc/yum.repos.d/enetres.repo
    wget http://nextgis.ru/programs/centos/nextgis.repo -O /etc/yum.repos.d/nextgis.repo
    yum install epel-release
```

- Install PostgreSQL and PostGIS according to this[21] instructions.
- Install CGAL:

```
    yum install libCGAL10
```

- Install pgRouting:

```
    yum install pgrouting_94
```

More info (and packages for CentOS) can be found here[22].

Fedora

- Fedora RPM's: https://admin.fedoraproject.org/pkgdb/package/rpms/pgRouting/

FreeBSD

pgRouting can be installed via ports:

```
cd /usr/ports/databases/pgRouting
make install clean
```

OS X

- Homebrew

```
brew install pgrouting
```

[21] https://trac.osgeo.org/postgis/wiki/UsersWikiPostGIS21CentOS6pgdg
[22] https://github.com/nextgis/gis_packages_centos/wiki/Using-this-repo

Source Package

You can find all the pgRouting Releases:

https://github.com/pgRouting/pgrouting/releases

See *Build Guide* to build the binaries from the source.

Using Git

Git protocol (read-only):

```
git clone git://github.com/pgRouting/pgrouting.git
```

HTTPS protocol (read-only):

```
git clone https://github.com/pgRouting/pgrouting.git
```

See *Build Guide* to build the binaries from the source.

1.2.2 Installing in the database

pgRouting is an extension.

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

1.2.3 Upgrading the database

To upgrade pgRouting to version 2.x.y use the following command:

```
ALTER EXTENSION pgrouting UPDATE TO "2.x.y";

For example to upgrade to 2.2.3

.. code-block:: sql

ALTER EXTENSION pgrouting UPDATE TO "2.2.3";
```

1.3 Build Guide

1.3.1 Dependencies

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

- C and C++0x compilers
- Postgresql version >= 9.1
- PostGIS version >= 2.0
- The Boost Graph Library (BGL). Version >= 1.46
- CMake >= 2.8.8
- CGAL >= 4.2
- (optional, for Documentation) Sphinx >= 1.1
- (optional, for Documentation as PDF) Latex >= [TBD]

1.3.2 Configuration

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

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

Ccreate the build directory

```
$ mkdir build
```

To configure:

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

Configurable variables

The documentation configurable variables are:

WITH_DOC BOOL=OFF – Turn on/off building the documentation

BUILD_HTML BOOL=ON – If WITH_DOC=ON, turn on/off building HTML

BUILD_LATEX BOOL=OFF – If WITH_DOC=ON, turn on/off building PDF

BUILD_MAN BOOL=OFF – If WITH_DOC=ON, turn on/off building MAN pages

Configuring with documentation

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

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

1.3.3 Building

Using `make` to build the code and the docuemtnation

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
```

1.3.4 Installation and reinstallation

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

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

For MinGW on Windows

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

For Linux

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

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

1.3.5 Dependencies Installation

Dependencies Installation

This guide was made while making a fresh ubuntu desktop 14.04.02 installation. Make the neceszry adjustments to fit your operative system.

Dependencies

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

- C and C++0x compilers
- Postgresql version >= 9.1
- PostGIS version >= 2.0
- The Boost Graph Library (BGL). Version >= 1.46
- CMake >= 2.8.8
- CGAL >= 4.2
- (optional, for Documentation) Sphinx >= 1.1
- (optional, for Documentation as PDF) Latex >= [TBD]

Before starting, on a terminal window:

sudo apt-get update

CMake >= 2.8.8 trusty provides: 2.8.8

```
sudo apt-get install cmake
```

C and (C++0x or c++11) compilers trusty provides: 4.8

```
sudo apt-get install g++
```

Postgresql version >= 9.1 For example in trusty 9.3 is provided:

```
sudo apt-get install postgreSQL
sudo apt-get install postgresql-server-dev-9.3
```

PostGIS version >= 2.0 For example in trusty 2.1 is provided:

```
sudo apt-get install postgresql-9.3-postgis-2.1
```

The Boost Graph Library (BGL). Version >= 1.46 trusty provides: 1.54.0

```
sudo apt-get install libboost-graph-dev
```

CGAL >= 4.2

```
sudo apt-get install libcgal-dev
```

(optional, for Documentation) Sphinx >= 1.1 http://sphinx-doc.org/latest/install.html

trusty provides: 1.2.2

```
sudo apt-get install python-sphinx
```

(optional, for Documentation as PDF) Latex >= [TBD] https://latex-project.org/ftp.html

trusty provides: 1.2.2

```
sudo apt-get install texlive
```

pgTap & pg_prove & perl for tests

Warning: cmake does not test for this packages.

```
sudo apt-get install -y perl
wget https://github.com/theory/pgtap/archive/master.zip
unzip master.zip
cd pgtap-master
make
sudo make install
sudo ldconfig
sudo apt-get install -y libtap-parser-sourcehandler-pgtap-perl
```

To run the tests:

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

See Also

Indices and tables

- genindex
- search

1.4 Support

pgRouting community support is available through the pgRouting website[23], documentation[24], tutorials, mailing lists and others. If you're looking for *commercial support*, find below a list of companies providing pgRouting development and consulting services.

[23] http://pgrouting.org/support.html
[24] http://docs.pgrouting.org

1.4.1 Reporting Problems

Bugs are reported and managed in an issue tracker[25]. 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[26] for it.
3. In your report include explicit instructions to replicate your issue. The best tickets include the exact SQL necessary to replicate a problem.
4. If you can test older versions of PostGIS for your problem, please do. On your ticket, note the earliest version the problem appears.
5. For the versions where you can replicate the problem, note the operating system and version of pgRouting, PostGIS and PostgreSQL.
6. It is recommended to use the following wrapper on the problem to pin point the step that is causing the problem.

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

1.4.2 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[27] 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[28].

1.4.3 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

[25] https://github.com/pgrouting/pgrouting/issues
[26] https://github.com/pgRouting/pgrouting/issues/new
[27] http://gis.stackexchange.com/
[28] http://gis.stackexchange.com/feeds/tag?tagnames=pgrouting&sort=newest

Tutorial

Tutorial

- *Getting started*
- *Routing Topology* for an overview of a topology for routing algorithms.
- *Graph Analytics* for an overview of the analysis of a graph.
- *Dictionary of columns & Custom Query* that is used in the routing algorithms.
- *Performance Tips* to improve your performance.
- *User's Recipes List*
- *Developer's Guide*

For a more complete introduction how to build a routing application read the pgRouting Workshop[1].

2.1 Tutorial

Getting started

- How to create a database to use for our project
- How to load some data
- How to build a topology
- How to check your graph for errors
- How to compute a route
- How to use other tools to view your graph and route
- How to create a web app

Advanced Topics

- *Routing Topology* for an overview of a topology for routing algorithms.
- *Graph Analytics* for an overview of the analysis of a graph.
- *Dictionary of columns & Custom Query* that is used in the routing algorithms.
- *Performance Tips* to improve your performance.

[1] http://workshop.pgrouting.org

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:

- How to create a database to use for our project
- How to load some data
- How to build a topology
- How to check your graph for errors
- How to compute a route
- How to use other tools to view your graph and route
- How to create a web app

How to create a 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 9.1 and later versions

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

How to load some 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-alpha

- 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.

How to build a 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-alpha*

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

See *pgr_createTopology* for more information.

How to check your graph for errors

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 requirments. 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)
```

See *Graph Analytics* for more information.

If your data needs to be *NODED*, we have a tool that can help for that also.

See *pgr_nodeNetwork* for more information.

How to compute a route

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_<algorithm>(<SQL for edges>, start, end, <additional options>)
```

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.

Indices and tables

- genindex
- search

2.1.2 Routing Topology

Author Stephen Woodbridge <woodbri@swoodbridge.com[2]>

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.

[2] woodbri@swoodbridge.com

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.

See Also

- *pgr_createTopology*
- *pgr_nodeNetwork*
- *pgr_pointToId - Deprecated Function*

2.1.3 Graph Analytics

Author Stephen Woodbridge <woodbri@swoodbridge.com[3]>

Copyright Stephen Woodbridge. The source code is released under the MIT-X license.

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.

[3] woodbri@swoodbridge.com

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.

See Also

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

2.1.4 Dictionary of columns & Custom Query

path a sequence of vertices/edges from A to B.

route a sequence of paths.

ANY-INTEGER Any of the following types: SMALLINT, INTEGER, BIGINT

ANY-NUMERICAL Any of the following types: SMALLINT, INTEGER, BIGINT, REAL, FLOAT

2.1.5 Custom Queries

Edges queries

Columns of the edges_sql queries

Depending on the function used the following columns are expected

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

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

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 pont vertex of the edge.

cost `ANY-NUMERICAL` weight of the edge *(source, target)*, if negative: edge *(source, target)* does not exist, therefore it's not part of the graph.

reverse_cost `ANY-NUMERICAL` (optional) weight of the edge *(target, source)*, if negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

edges_sql `TEXT` SQL query as described above.

start_vid `BIGINT` identifier of the starting vertex of the path.

start_vids `array[ANY-INTEGER]` array of identifiers of starting vertices.

end_vid `BIGINT` identifier of the ending vertex of the path.

end_vids `array[ANY-INTEGER]` array of identifiers of ending vertices.

directed `boolean` (optional). When `false` the graph is considered as Undirected. Default is `true` which considers the graph as Directed.

Description of the return values

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

seq `INTEGER` is a sequential value starting from **1**.

route_seq `INTEGER` relative position in the route. Has value **1** for the beginning of a route.

route_id `INTEGER` id of the route.

path_seq `INTEGER` relative position in the path. Has value **1** for the beginning of a path.

path_id `INTEGER` id of the path.

start_vid `BIGINT` id of the starting vertex. Used when multiple starting vetrices are in the query.

end_vid `BIGINT` id of the ending vertex. Used when multiple ending vertices are in the query.

node `BIGINT` id of the node in the path from start_vid to end_v.

edge `BIGINT` id 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` total cost from `start_vid` to `node`.

Descriptions for version 2.0 signatures

In general, the routing algorithms need an SQL query that contain one or more of the following required columns with the preferred type:

id int4

source int4

target int4

cost float8

reverse_cost float8

x float8

y float8

x1 float8

y1 float8

x2 float8

y2 float8

```
SELECT source, target, cost FROM edge_table;
SELECT id, source, target, cost FROM edge_table;
SELECT id, source, target, cost, x1, y1, x2, y2 ,reverse_cost FROM edge_table
```

When the edge table has a different name to represent the required columns:

```
SELECT src as source,  target, cost FROM othertable;
SELECT gid as id, src as source, target, cost FROM othertable;
SELECT gid as id, src as source, target, cost, fromX as x1, fromY as y1, toX as x2, toY as y2 ,Rc
       FROM othertable;
```

The topology functions use the same names for `id`, `source` and `target` columns of the edge table, The fowllowing parameters have as default value:

id int4 Default `id`

source int4 Default `source`

target int4 Default `target`

the_geom text Default `the_geom`

oneway text Default `oneway`

rows_where text Default `true` to indicate all rows (this is not a column)

The following parameters do not have a default value and when used they have to be inserted in strict order:

edge_table text

tolerance float8

s_in_rules text[]

s_out_rules text[]

t_in_rules text[]

t_out_rules text[]

When the columns required have the default names this can be used (pgr_func is to represent a topology function)

```
pgr_func('edge_table')          -- when tolerance is not required
pgr_func('edge_table',0.001)    -- when tolerance is required
-- s_in_rule, s_out_rule, st_in_rules, t_out_rules are required
SELECT pgr_analyzeOneway('edge_table', ARRAY['', 'B', 'TF'], ARRAY['', 'B', 'FT'],
                                       ARRAY['', 'B', 'FT'], ARRAY['', 'B', 'TF'])
```

When the columns required do not have the default names its strongly recommended to use the *named notation*.

```
pgr_func('othertable', id:='gid',source:='src',the_geom:='mygeom')
pgr_func('othertable',0.001,the_geom:='mygeom',id:='gid',source:='src')
SELECT pgr_analyzeOneway('othertable', ARRAY['', 'B', 'TF'], ARRAY['', 'B', 'FT'],
                                       ARRAY['', 'B', 'FT'], ARRAY['', 'B', 'TF']
                         source:='src',oneway:='dir')
```

2.1.6 Performance Tips

For the Routing functions:

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

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');
```

Use an SQL that "removes" the edges in the routing function

```
SELECT id, source, target from edge_table WHERE id < 17
```

When "you know" that the route will not go out of a particular area, to speed up the process you can use a more complex SQL query like

```
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)
```

Note that the same condition `id < 17` is used in all cases.

2.2 User's Recipes List

2.2.1 Comparing topology of a unnoded network with a noded network

Author pgRouting team.

Licence Open Source

This recipe uses the *Sample Data* network.

The purpose of this recipe is to compare a not nodded network with a nodded network.

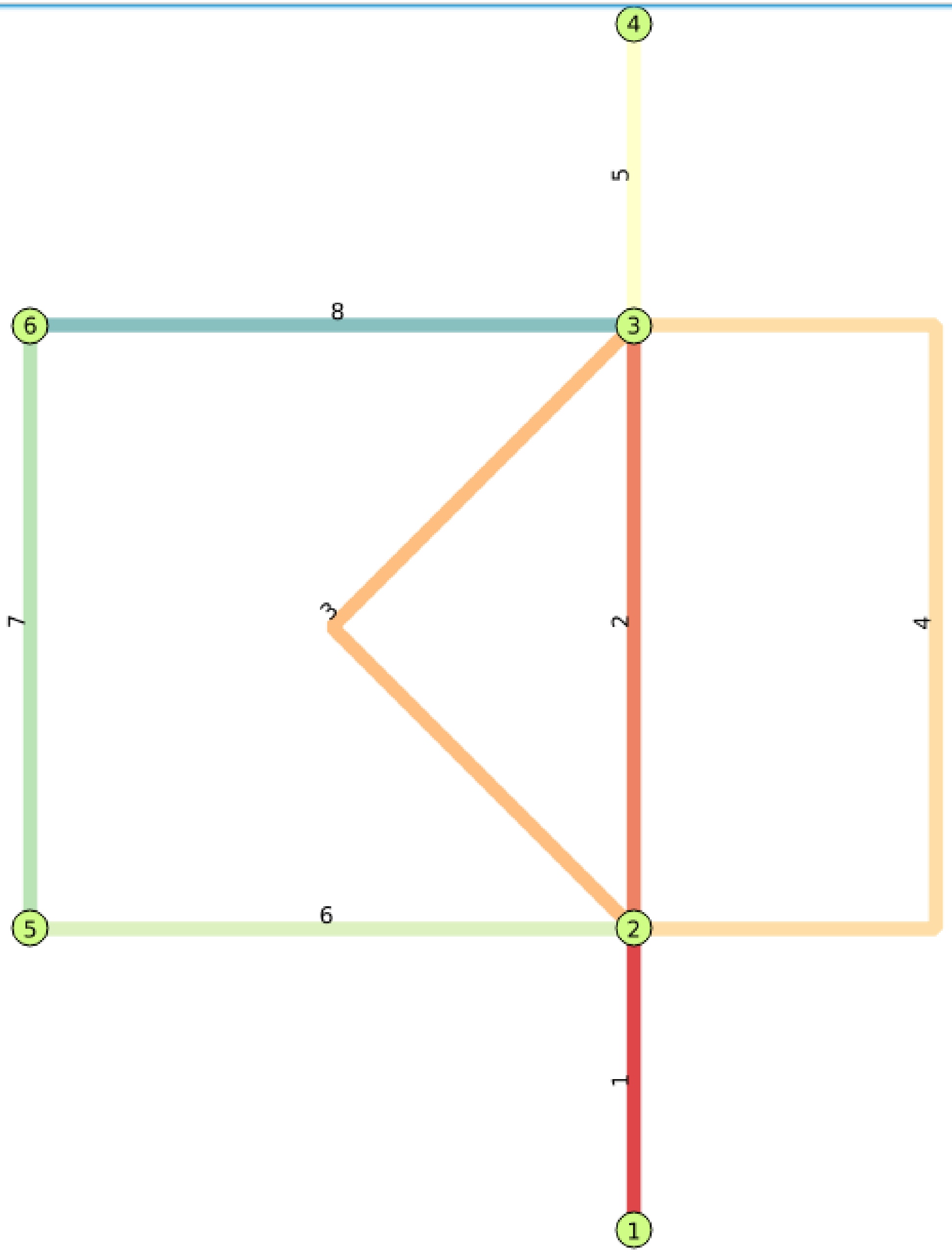
```
SELECT pgr_createTopology('edge_table', 0.001);
SELECT pgr_analyzegraph('edge_table', 0.001);
SELECT pgr_nodeNetwork('edge_table', 0.001);
SELECT pgr_createTopology('edge_table_noded', 0.001);
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
```

2.2.2 Handling parallels after getting a path (pgr_ksp focus)

Author pgRouting team.

Licence Open Source

The graph



Data

```
drop table if exists parallel;
CREATE TABLE parallel (
  id serial,
  source integer,
  target integer,
  cost double precision,
  reverse_cost double precision,
  x1 double precision,
  y1 double precision,
  x2 double precision,
  y2 double precision,
  the_geom geometry
);
```

```
INSERT INTO parallel (x1,y1,x2,y2)
  VALUES (1,0,1,1),(1,1,1,3),(1,1,1,3),(1,1,1,3),(1,3,1,4),(1,1,-1,1),(-1,1,-1,3),(-1,3,1,3);
UPDATE parallel SET the_geom = ST_makeline(ST_point(x1,y1),ST_point(x2,y2));
UPDATE parallel SET the_geom = ST_makeline(ARRAY[ST_point(1,1),ST_point(0,2),ST_point(1,3)]) WHERE
UPDATE parallel SET the_geom = ST_makeline(ARRAY[ST_point(1,1),ST_point(2,1),ST_point(2,3),ST_poi
                              WHERE id = 4;
UPDATE parallel SET cost = ST_length(the_geom), reverse_cost = ST_length(the_geom);
SELECT pgr_createTopology('parallel',0.001);
```

pgr_ksp results

We ignore the costs because we want all the parallels

```
SELECT seq, path_id AS route, node, edge INTO routes
  from pgr_ksp('select id, source, target, cost, reverse_cost from parallel',
  1, 4, 3);

select route, node, edge from routes;
  route | node | edge
 -------+------+------
      1 |    1 |    1
      1 |    2 |    2
      1 |    3 |    5
      1 |    4 |   -1
      2 |    1 |    1
      2 |    2 |    6
      2 |    5 |    7
      2 |    6 |    8
      2 |    3 |    5
      2 |    4 |   -1
 (10 rows)
```

We need an aggregate function:

```
CREATE AGGREGATE array_accum (anyelement)
(
  sfunc = array_append,
  stype = anyarray,
  initcond = '{}'
);
```

Now lets generate a table with the parallel edges.

```
select distinct seq,route,source,target, array_accum(id) as edges into paths
  from (select seq, route, source, target
        from parallel, routes where id = edge) as r
     join parallel using (source, target)
  group by seq,route,source,target order by seq;

 select route, source, targets, edges from paths;
   route | source | target |  edges
  -------+--------+--------+---------
       1 |      1 |      2 | {1}
       2 |      1 |      2 | {1}
       2 |      2 |      5 | {6}
       1 |      2 |      3 | {2,3,4}
       2 |      5 |      6 | {7}
       1 |      3 |      4 | {5}
```

```
       2 |       6 |       3 | {8}
       2 |       3 |       4 | {5}
  (8 rows)
```

Some more aggregate functions

To generate a table with all the combinations for parallel routes, we need some more aggregates

```
create or replace function multiply( integer, integer )
returns integer as
$$
  select $1 * $2;
$$
language sql stable;

create aggregate prod(integer)
(
  sfunc = multiply,
  stype = integer,
  initcond = 1
);
```

And a function that "Expands" the table

```
CREATE OR REPLACE function   expand_parallel_edge_paths(tab text)
  returns TABLE (
                seq    INTEGER,
                route  INTEGER,
                source INTEGER, target INTEGER, -- this ones are not really needed
                edge   INTEGER ) AS
 $body$
 DECLARE
 nroutes   INTEGER;
 newroutes INTEGER;
 rec   record;
 seq2 INTEGER := 1;
 rnum INTEGER := 0;

 BEGIN     -- get the number of distinct routes
   execute 'select count(DISTINCT route) from ' || tab INTO nroutes;
   FOR i IN 0..nroutes-1
   LOOP
       -- compute the number of new routes this route will expand into
       -- this is the product of the lengths of the edges array for each route
       execute 'select prod(array_length(edges, 1))-1 from '
       ||       quote_ident(tab) || ' where route='    || i INTO newroutes;
       -- now we generate the number of new routes for this route
       -- by repeatedly listing the route and swapping out the parallel edges
       FOR j IN 0..newroutes
       LOOP
           -- query the specific route
           FOR rec IN execute 'select * from ' || quote_ident(tab) ||' where route=' || i
                        || ' order by seq'
           LOOP
               seq := seq2;
               route := rnum;
               source := rec.source;
               target := rec.target;
               -- using module arithmetic iterate through the various edge choices
               edge := rec.edges[(j % (array_length(rec.edges, 1)))+1];
```

```
                -- return a new record
                RETURN next;
                seq2 := seq2 + 1;     -- increment the record count
             END LOOP;
             seq := seq2;
             route := rnum;
             source := rec.target;
             target := -1;
             edge := -1;
             RETURN next;  -- Insert the ending record of the route
             seq2 := seq2 + 1;

             rnum := rnum + 1;  -- increment the route count
         END LOOP;
      END LOOP;
END;
$body$
language plpgsql volatile strict   cost 100 rows 100;
```

Test it

```
select * from expand_parallel_edge_paths( 'paths' );
 seq | route | source | target | edge
-----+-------+--------+--------+------
   1 |     0 |      1 |      2 |    1
   2 |     0 |      2 |      3 |    2
   3 |     0 |      3 |      4 |    5
   4 |     0 |      4 |     -1 |   -1
   5 |     1 |      1 |      2 |    1
   6 |     1 |      2 |      3 |    3
   7 |     1 |      3 |      4 |    5
   8 |     1 |      4 |     -1 |   -1
   9 |     2 |      1 |      2 |    1
  10 |     2 |      2 |      3 |    4
  11 |     2 |      3 |      4 |    5
  12 |     2 |      4 |     -1 |   -1
  13 |     3 |      1 |      2 |    1
  14 |     3 |      2 |      5 |    6
  15 |     3 |      5 |      6 |    7
  16 |     3 |      6 |      3 |    8
  17 |     3 |      3 |      4 |    5
  18 |     3 |      4 |     -1 |   -1
(18 rows)
```

No more contributions

2.3 How to contribute.

To add a recipie or a wrapper

The first steps are:

- Fork the repository
- Create a branch for your recipe or wrapper
- Create your recipe file

```
cd doc/src/recipes/
vi myrecipe.rst
git add myrecipe.rst
# include the recipe in this file
vi index.rst
```

To create the test file of your recipe

```
cd test
cp myrecipe.rst myrecipe.sql.test

# make your test based on your recipe
vi myrecipe.sql.test
git add myrecipe.sql.test

# create your test results file
touch myrecipe.result
git add myrecipe.result

# add your test to the file
vi test.conf
```

Leave only the SQL code on `myrecipe.sql.test` by deleting lines or by commenting lines.

To get the results of your recipe

From the root directory execute:

```
tools/test-runner.pl -alg recipes -ignorenotice
```

Copy the results of your recipe and paste them in the file `myrecipe.result`, and remove the "> " from the file.

make a pull request.

```
git commit -a -m 'myrecipe is for this and that'
git push
```

From your fork in github make a pull request over develop

2.4 Developer's Guide

This contains some basic comments about developing. More detailed information can be found on:

http://docs.pgrouting.org/doxy/2.2/index.html

2.4.1 Source Tree Layout

cmake/ cmake scripts used as part of our build system.

src/ This is the algorithm source tree. Each algorithm is to be contained in its on sub-tree with /doc, /sql, /src, and /test sub-directories.

For example:

- `src/dijkstra` Main direcotry for dijkstra algorithm.

- `src/dijkstra/doc` Dijkstra's documentation directory.
- `src/dijkstra/src` Dijkstra's C and/or C++ code.
- `src/dijkstra/sql` Dijkstra's sql code.
- `src/dijkstra/test` Dijkstra's tests.
- `src/dijkstra/test/pgtap` Dijkstra's pgTaptests.

2.4.2 Tools

tools/ Miscellaneous scripts and tools.

pre-commit

To keep version/branch/commit up to date install pelase do the following:

```
cp tools/pre-commit .git/hooks/pre-commit
```

After each commit a the file **VERSION** will remain. (The hash number will be one behind)

doxygen

To use doxygen:

```
cd tools/doxygen/
make
```

The code's documentation can be found in:

```
build/doxy/html/index.html
```

cpplint

We try to follow the following guidelines for C++ coding:

https://google-styleguide.googlecode.com/svn/trunk/cppguide.html

Sample use:

```
python cpplint.py ../src/dijkstra/src/dijkstra_driver.h
../src/dijkstra/src/dijkstra_driver.h:34:  Lines should be <= 80 characters long  [whitespace/line
../src/dijkstra/src/dijkstra_driver.h:40:  Line ends in whitespace.  Consider deleting these extra
Done processing ../src/dijkstra/src/dijkstra_driver.h
Total errors found: 2
```

- Maybe line 34 is a very complicated calculation so you can just ignore the message
- Delete whitespace at end of line is easy fix.
- Use your judgement!!!

Some files like `postgres.h` are system dependent so don't include the directory.

Other tools

Tools like:

- doit
- winnie

- publish_doc.sh

are very specific for the deployment of new versions, so please ask first!

2.4.3 Documentation Layout

Note: All documentation should be in reStructuredText format. See: <http://docutils.sf.net/rst.html> for introductory docs.

Documentation is distributed into the source tree. This top level "doc" directory is intended for high level documentation cover subjects like:

- Compiling and testing
- Installation
- Tutorials
- Users' Guide front materials
- Reference Manual front materials
- etc

Since the algorithm specific documentation is contained in the source tree with the algorithm specific files, the process of building the documentation and publishing it will need to assemble the details with the front material as needed.

Also, to keep the "doc" directory from getting cluttered, each major book like those listed above, should be contained in a separate directory under "doc". Any images or other materials related to the book should also be kept in that directory.

Testing Infrastructure

Tests are part of the tree layout:

- `src/dijkstra/test` Dijkstra's tests.
  - `test.conf` Configuration file.
  - `<name>.test.sql` Test file
  - `<name>.result` Results file bash
- `src/dijkstra/test/pgtap` Dijkstra's pgTaptests.
  - `<name>.sql` pgTap test file

Testing

Testing is executed from the top level of the tree layout:

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

Indices and tables

- genindex
- search

CHAPTER 3

Sample Data

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

3.1 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,
    category_id INTEGER,
    reverse_category_id INTEGER,
    x1 FLOAT,
    y1 FLOAT,
    x2 FLOAT,
    y2 FLOAT,
    the_geom geometry
);
```

Insert data

```
INSERT INTO edge_table (
    category_id, reverse_category_id,
    cost, reverse_cost,
    x1, y1,
    x2, y2) VALUES
(3, 1,    1,  1,  2,   0,    2, 1),
(3, 2,   -1,  1,  2,   1,    3, 1),
(2, 1,   -1,  1,  3,   1,    4, 1),
(2, 4,    1,  1,  2,   1,    2, 2),
(1, 4,    1, -1,  3,   1,    3, 2),
(4, 2,    1,  1,  0,   2,    1, 2),
(4, 1,    1,  1,  1,   2,    2, 2),
(2, 1,    1,  1,  2,   2,    3, 2),
(1, 3,    1,  1,  3,   2,    4, 2),
(1, 4,    1,  1,  2,   2,    2, 3),
```

```
(1, 2,    1, -1,  3,   2,    3, 3),
(2, 3,    1, -1,  2,   3,    3, 3),
(2, 4,    1, -1,  3,   3,    4, 3),
(3, 1,    1,  1,  2,   3,    2, 4),
(3, 4,    1,  1,  4,   2,    4, 3),
(3, 3,    1,  1,  4,   1,    4, 2),
(1, 2,    1,  1,  0.5, 3.5,  1.999999999999,3.5),
(4, 1,    1,  1,  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 *Maximum Flow* 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.

```
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);
```

3.1.1 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.

Note: On all graphs,

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.

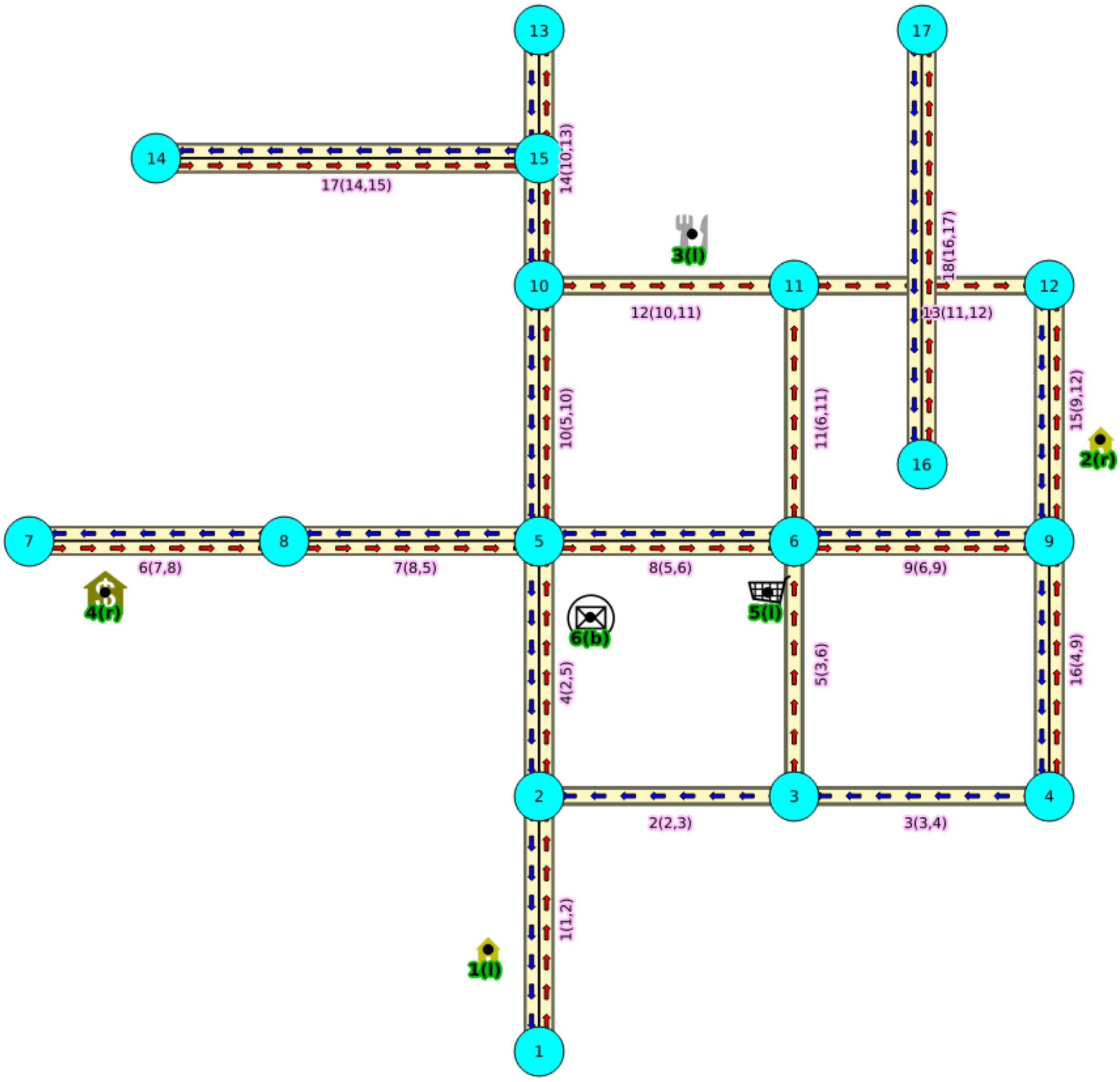


Fig. 3.1: **Graph 1: Directed, with cost and reverse cost**

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

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

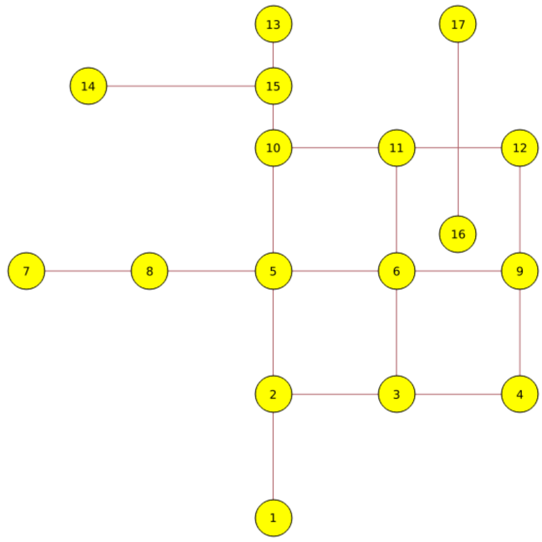


Fig. 3.2: **Graph 2: Undirected, with cost and reverse cost**

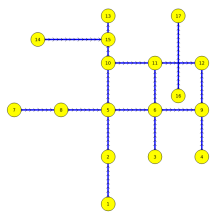


Fig. 3.3: **Graph 3: Directed, with cost**

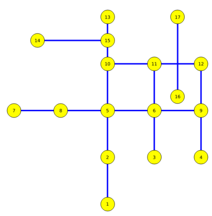


Fig. 3.4: **Graph 4: Undirected, with cost**

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 INTEGER NOT NULL PRIMARY KEY,
    x INTEGER,
    y INTEGER,
    demand INTEGER,
    openTime INTEGER,
    closeTime INTEGER,
    serviceTime INTEGER,
    pindex INTEGER,
    dindex INTEGER
    );
copy customer (id, x, y, demand, openTime, closeTime, serviceTime, pindex, dindex) from stdin;
0	40	50	0	0	1236	0	0	0
1	45	68	-10	912	967	90	11	0
2	45	70	-20	825	870	90	6	0
3	42	66	10	65	146	90	0	75
4	42	68	-10	727	782	90	9	0
5	42	65	10	15	67	90	0	7
6	40	69	20	621	702	90	0	2
7	40	66	-10	170	225	90	5	0
8	38	68	20	255	324	90	0	10
9	38	70	10	534	605	90	0	4
10	35	66	-20	357	410	90	8	0
11	35	69	10	448	505	90	0	1
12	25	85	-20	652	721	90	18	0
13	22	75	30	30	92	90	0	17
14	22	85	-40	567	620	90	16	0
15	20	80	-10	384	429	90	19	0
16	20	85	40	475	528	90	0	14
17	18	75	-30	99	148	90	13	0
18	15	75	20	179	254	90	0	12
19	15	80	10	278	345	90	0	15
20	30	50	10	10	73	90	0	24
21	30	52	-10	914	965	90	30	0
22	28	52	-20	812	883	90	28	0
23	28	55	10	732	777	0	0	103
24	25	50	-10	65	144	90	20	0
25	25	52	40	169	224	90	0	27
26	25	55	-10	622	701	90	29	0
27	23	52	-40	261	316	90	25	0
28	23	55	20	546	593	90	0	22
29	20	50	10	358	405	90	0	26
30	20	55	10	449	504	90	0	21
31	10	35	-30	200	237	90	32	0
32	10	40	30	31	100	90	0	31
33	8	40	40	87	158	90	0	37
34	8	45	-30	751	816	90	38	0
35	5	35	10	283	344	90	0	39
36	5	45	10	665	716	0	0	105
37	2	40	-40	383	434	90	33	0
38	0	40	30	479	522	90	0	34
39	0	45	-10	567	624	90	35	0
40	35	30	-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 4

Functions

4.1 Version

pgr_version - to get pgRouting's version information.

4.1.1 pgr_version

Name

`pgr_version` — Query for pgRouting version information.

Synopsis

Returns a table with pgRouting version information.

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

Description

Returns a table with:

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

Examples

- Query for full version string

```
SELECT pgr_version();

                    pgr_version
-----------------------------------------------
 (2.2.0,pgrouting-2.2.0,9fd33c5,master,1.54.0)
(1 row)
```

- Query for `version` and `boost` attribute

```
SELECT version, boost FROM pgr_version();

  version   | boost
-----------+--------
 2.2.0-dev | 1.49.0
(1 row)
```

See Also

- *pgr_versionless - Deprecated Function* to compare two version numbers

4.2 Data Types

pgRouting 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.

4.2.1 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.

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

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*

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*

Topology functions

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.

5.1 Topology 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.

5.1.1 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 FROM
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

5.1.2 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');
```

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 e
```

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');`

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.

5.1.3 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');`
> ERROR: Can not determine the srid of the geometry "id" in table public.edge_table

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');`
> ERROR: Can not determine the srid of the geometry "gid" in table public.mytable

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
                                       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.

5.1.4 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_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.

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.

5.1.5 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 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

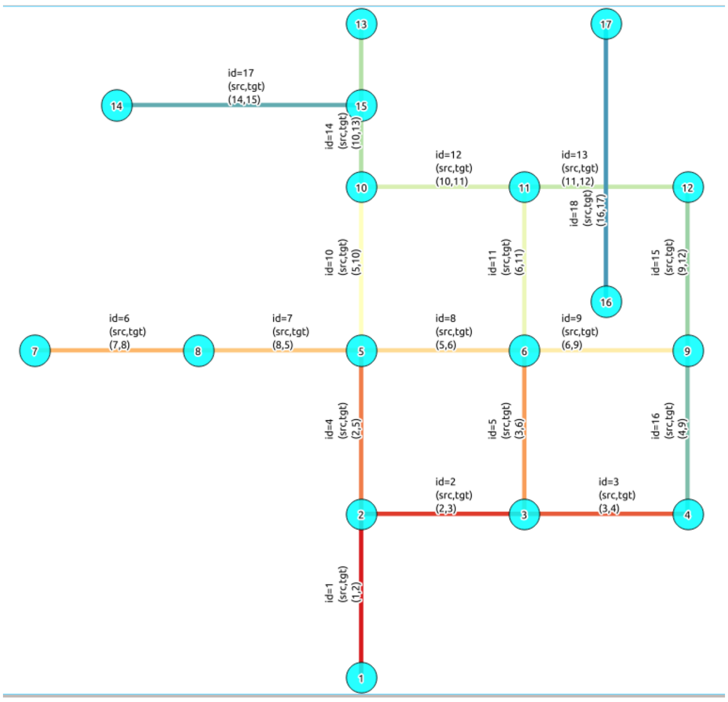
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED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

Now let’s analyze the new topology

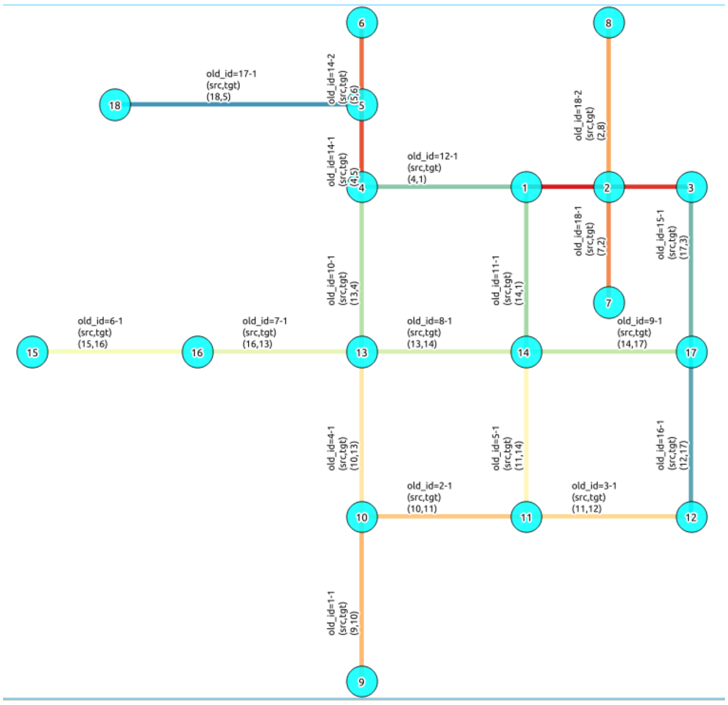
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
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.

CHAPTER 6

Routing Functions

6.1 Routing Functions

- *All pairs* - All pair of vertices.
  - *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* - Dijkstra family functions
  - *pgr_dijkstra* - Dijkstra's shortest path algorithm.
  - *pgr_dijkstraCost* - Use pgr_dijkstra to calculate the costs of the shortest paths.
- *Driving Distance* - Driving Distance
  - *pgr_drivingDistance* - Driving Distance
  - Post processing
    - *pgr_alphaShape* - Alpha shape computation
    - *pgr_pointsAsPolygon* - Polygon around set of points
- *pgr_ksp* - K-Shortest Path
- *pgr_trsp* - Turn Restriction Shortest Path (TRSP)
- *Traveling Sales Person*
  - *pgr_TSP* - When input is a cost matrix.
  - *pgr_eucledianTSP* - When input are coordinates.

6.1.1 All pairs

The following functions work an all vertices pair combinations

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

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. 6.1: Boost Graph Inside

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 make use of the 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

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.

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. 6.2: Boost Graph Inside

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
```

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

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

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.

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=${BBOX}]
```

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.

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

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

6.1.2 pgr_aStar

Name

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

[6]

Fig. 6.3: Boost Graph Inside

Synopsis

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.

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

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 ∞
- 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)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost)
```

Note: This signature is deprecated

```
pgr_aStar(sql, source integer, target integer, directed boolean, has_rcost boolean)
RETURNS SET OF pgr_costResult
```

- See *pgr_costResult*
- See *pgr_astar - Deprecated Signature*

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

Complete Signature

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

Example Setting a Heuristic

```
SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, 12, heuristic := 1);
 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 |   15 |    1 |        3
   5 |        5 |   12 |   -1 |    0 |        4
(5 rows)

SELECT * FROM pgr_astar(
    'SELECT id, source, target, cost, reverse_cost, x1, y1, x2, y2 FROM edge_table',
    2, 12, 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 |   15 |    1 |        3
   5 |        5 |   12 |   -1 |    0 |        4
(5 rows)
```

Description of the Signatures

Note: The following only aplies to the new signature(s)

Description of the edges_sql query

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

Column	Type	Default	Description
id	`ANY-INTEGER`		Identifier of the edge.
source	`ANY-INTEGER`		Identifier of the first end point vertex of the edge.
target	`ANY-INTEGER`		Identifier of the second end point vertex of the edge.
cost	`ANY-NUMERICAL`		**Weight of the edge *(source, target)*** • When negative: edge *(source, target)* does not exist, therefore it's not part of the graph.
reverse_cost	`ANY-NUMERICAL`	-1	**Weight of the edge *(target, source)*,** • When negative: edge *(target, source)* does not exist, therefore it's not part of the graph.
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.
end_vid	`ANY-INTEGER`	Ending vertex identifier.
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`.
epsilon	`FLOAT`	(optional). For less restricted results. $factor >= 1$. Default `1`.

Description of the return values

Returns set of `(seq, path_seq, 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.
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_vid` to `node`.** • `0` for the first row in the path sequence.

About 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

History

- Functionality added version 2.3.0
- Renamed in version 2.0.0

Deprecated Signature

Example Using the deprecated signature

```
SELECT * FROM pgr_astar(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost, x1, y1, x2, y2 FROM
    2, 12, true, true);
NOTICE:  Deprecated signature of function pgr_astar
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   4 |    1
   1 |   5 |   8 |    1
   2 |   6 |   9 |    1
   3 |   9 |  15 |    1
   4 |  12 |  -1 |    0
(5 rows)
```

The queries use the *Sample Data* network.

See Also

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

6.1.3 pgr_bdAstar - Bi-directional A* Shortest Path

Name

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

Synopsis

This is a bi-directional A* search algorithm. It searches from the source toward the distination and at the same time from the destination to the source and terminates whe these to searches meet in the middle. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows, that make up a path.

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

Description

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

```
        SELECT id, source, target, cost, x1, y1, x2, y2 [,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.

x1 `x` coordinate of the start point of the edge

y1 `y` coordinate of the start point of the edge

x2 `x` coordinate of the end point of the edge

y2 `y` coordinate of the end point of the edge

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` id of the start point

target `int4` 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.

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`

Warning: You must reconnect to the database after `CREATE EXTENSION pgrouting`. Otherwise the function will return `Error computing path: std::bad_alloc`.

History

- New in version 2.0.0

Examples

- Without `reverse_cost`

```
SELECT * FROM pgr_bdAStar(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, x1, y1, x2, y2
     FROM edge_table',
    4, 10, false, false);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   4 |   3 |    0
   1 |   3 |   5 |    1
   2 |   6 |  11 |    1
   3 |  11 |  12 |    0
   4 |  10 |  -1 |    0
(5 rows)
```

- With `reverse_cost`

```
SELECT * FROM pgr_bdAStar(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, x1, y1, x2, y2, reverse_cost
     FROM edge_table ',
    4, 10, true, true);
```

```
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   4 |   3 |    1
   1 |   3 |   5 |    1
   2 |   6 |   8 |    1
   3 |   5 |  10 |    1
   4 |  10 |  -1 |    0
(5 rows)
```

The queries use the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *pgr_bdDijkstra - Bi-directional Dijkstra Shortest Path*
- http://en.wikipedia.org/wiki/Bidirectional_search
- http://en.wikipedia.org/wiki/A*_search_algorithm

6.1.4 pgr_bdDijkstra - Bi-directional Dijkstra Shortest Path

Name

`pgr_bdDijkstra` - Returns the shortest path using Bidirectional Dijkstra algorithm.

Synopsis

This is a bi-directional Dijkstra search algorithm. It searches from the source toward the distination and at the same time from the destination to the source and terminates whe these to searches meet in the middle. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows, that make up a path.

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

Description

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` id of the start point

target `int4` 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.

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

Examples

- Without `reverse_cost`

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
    4, 10, false, false);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   4 |     |    0
   1 |   3 |     |    0
   2 |   2 |     |    1
   3 |   5 |     |    1
   4 |  10 |     |    0
(5 rows)
```

- With `reverse_cost`

```
SELECT * FROM pgr_bdDijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    4, 10, true, true);
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   4 |     |    1
   1 |   3 |     |    1
   2 |   2 |     |    1
   3 |   5 |     |    1
   4 |  10 |     |    0
(5 rows)
```

The queries use the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *pgr_bdAstar - Bi-directional A* Shortest Path*
- http://en.wikipedia.org/wiki/Bidirectional_search
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

6.1.5 Dijkstra - Family of functions

- *pgr_dijkstra* - Dijkstra's algorithm for the shortest paths.

The following algorithms are based on pgr_dijkstra

- *pgr_dijkstraCost* - Get the aggregate cost of the shortest paths.
- *pgr_drivingDistance* - Get catchament information.
- *pgr_ksp* - Get the aggregate cost of the shortest paths.

pgr_dijkstra

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

[7]

Fig. 6.4: 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

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

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
path_seq	`INT`	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	`BIGINT`	Identifier of the starting vertex. 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 *Graph 1: Directed, with cost and reverse cost*

```
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 *Graph 2: Undirected, with cost and reverse cost*

```
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 |   11 |    1 |        0
   4 |        2 |        11 |    6 |    8 |    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 *Graph 3: Directed, with cost*

```
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 *Graph 4: Undirected, with cost*

```
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 |   11 |    1 |        0
   4 |        2 |        11 |    6 |    8 |    1 |        1
   5 |        3 |        11 |    5 |   -1 |    0 |        2
(5 rows)

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

```
   3 |        3 |        3 |    6 |     5 |    1 |        2
   4 |        4 |        3 |    3 |    -1 |    0 |        3
   5 |        1 |        5 |    2 |     4 |    1 |        0
   6 |        2 |        5 |    5 |    -1 |    0 |        1
(6 rows)

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

Equvalences between signatures

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

The examples in this section use the following:

- *Graph 1: Directed, with cost and reverse cost*

```
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)

SET client_min_messages TO NOTICE;
SET
SELECT * FROM pgr_dijkstra(
```

```
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    2, 3,
    TRUE,
    TRUE
);
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   4 |    1
   1 |   5 |   8 |    1
   2 |   6 |   9 |    1
   3 |   9 |  16 |    1
   4 |   4 |   3 |    1
   5 |   3 |  -1 |    0
(6 rows)
```

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

The examples in this section use the following:

- *Graph 2: Undirected, with cost and reverse cost*

```
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)

SET client_min_messages TO NOTICE;
SET
SELECT * FROM pgr_dijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    2, 3,
    FALSE,
    TRUE
);
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   2 |    1
   1 |   3 |  -1 |    0
(2 rows)
```

History

- Added functionality in version 2.1.0
- Renamed in version 2.0.0

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.

[8]

Fig. 6.5: Boost Graph Inside

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

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 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`	Aggregate cost of the shortest path 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)
```

History

- New in version 2.2.0

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
  - $E = \begin{cases} \text{if } reverse\_cost = & \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ & \{(source_i, target_i, cost_i) \text{ when } cost >= 0\} \\ \cup & \{(target_i, source_i, reverse\_cost_i) \text{ when } reverse\_cost_i >= 0)\} \\ \text{if } reverse\_cost \neq & \end{cases}$

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

- $G(V,E) = \begin{cases} G_d(V,E) & \text{if } directed = true \\ G_u(V,E) & \text{if } directed = false \end{cases}$

Then:

$$\text{pgr\_dijkstra}(sql, start_{vid}, end_{vid}, directed) = \begin{cases} \text{shortest path } \boldsymbol{\pi} \text{ between } start_{vid} \text{and } end_{vid} & \text{if } \exists \boldsymbol{\pi} \\ & \text{otherwise} \end{cases}$$

$\boldsymbol{\pi} = \{(path_{i, node_i, edge_i, cost_i, agg\_cost_i})\}$

where:

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

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

- $path$ 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.

6.1.6 Driving Distance

- *pgr_drivingDistance* - Driving Distance based on pgr_dijkstra

pgr_drivingDistance

Name

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

[9]

Fig. 6.6: Boost Graph Inside

Synopsis

Using Dijkstra algorithm, extracts all the nodes that have costs less than or equal to the value `distance`. The edges extracted will conform 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

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 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 arbitrarely.

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 *Graph 1: Directed, with cost and reverse cost*

```
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 *Graph 2: Undirected, with cost and reverse cost*

```
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 *Graph 3: Directed, with cost*

```
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 *Graph 4: Undirected, with cost*

```
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

6.1.7 Driving Distance post-processing

- *pgr_alphaShape* - Alpha shape computation
- *pgr_pointsAsPolygon* - Polygon around 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[10].

Returns a vertex record for each row:

x x-coordinate

y y-coordinate

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

In the alpha shape code we have no way to control the order of the points so the actual output you might get could be similar but different. The simple query is followed by a more complex one that constructs a polygon and computes the areas of it. This should be the same as the result on your system. We leave the details of the complex query to the reader as an exercise if they wish to decompose it into understandable pieces or to just copy and paste it into a SQL window to run.

```
SELECT * FROM pgr_alphaShape('SELECT id, x, y FROM vertex_table');

 x | y
---+---
 2 | 4
 0 | 2
 2 | 0
 4 | 1
 4 | 2
 4 | 3
(6 rows)

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, x, y FROM vertex_table')
) AS a) AS foo;

 st_area
---------
   10.00
(1 row)


SELECT * FROM pgr_alphaShape('SELECT id::integer, ST_X(the_geom)::float AS x, ST_Y(the_geom)::floa
  x  |  y
-----+-----
   2 |    4
 0.5 | 3.5
```

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

```
   0 |    2
   2 |    0
   4 |    1
   4 |    2
   4 |    3
 3.5 |    4
(8 rows)

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 FI
) 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

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[11].

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

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

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

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

6.1.8 pgr_ksp

Name

`pgr_ksp` — Returns the “K” shortest paths.

[12]

Fig. 6.7: Boost Graph Inside

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

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

Column	Type	Description
edges_-sql	`TEXT`	SQL query as described above.
start_-vid	`BIGINT`	Identifier of the starting vertex.
end_vid	`BIGINT`	Identifier of the ending vertex.
k	`INTEGER`	The desiered number of paths.
directed	`BOOLEAN`	(optional). When `false` the graph is considered as Undirected. Default is `true` which considers the graph as Directed.
heap_-paths	`BOOLEAN`	(optional). When `true` returns all the paths stored in the process heap. Default is `false` which only returns `k` pahts.

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 *Graph 1: Directed, with cost and reverse cost*

```
SELECT * FROM pgr_ksp(
     'SELECT id, source, target, cost, reverse_cost FROM edge_table',
      2, 12, 2,
      true
   );
NOTICE:  Deprecated function
 seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+------
   0 |   0 |   2 |   4 |    1
   1 |   0 |   5 |   8 |    1
   2 |   0 |   6 |   9 |    1
   3 |   0 |   9 |  15 |    1
   4 |   0 |  12 |  -1 |    0
   5 |   1 |   2 |   4 |    1
   6 |   1 |   5 |   8 |    1
```

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

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 *Graph 1: Directed, with cost and reverse cost*

```
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 *Graph 2: Undirected, with cost and reverse cost*

```
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 *Graph 3: Directed, with cost*

```
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 *Graph 4: Undirected, with cost*

```
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

6.1.9 Traveling Sales Person

- *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.

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 informetion 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

Note: The following only aplies to the new signature(s)

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 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 values 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 n** • `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.

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.

History

- Rewritten in version 2.3.0
- Renamed in version 2.0.0
- GAUL dependency removed in version 2.0.0

See Also

- *Traveling Sales Person*

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

pgr_eucledianTSP

Name

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

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 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 values 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 n** • `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.

Examples

Example Skipping the Simulated Annealing & showing some process information

```
SET client_min_messages TO NOTICE;
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
);
NOTICE:  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*
- http://en.wikipedia.org/wiki/Traveling_salesman_problem
- http://en.wikipedia.org/wiki/Simulated_annealing

Note: These signatures are being deprecated

```
-- (1)
pgr_costResult[] pgr_tsp(sql text, start_id integer)
pgr_costResult[] pgr_tsp(sql text, start_id integer, end_id integer)

-- (2)
record[] pgr_tsp(matrix float[][], start integer)
record[] pgr_tsp(matrix float[][], start integer, end integer)
```

- See http://docs.pgrouting.org/2.2/en/src/common/doc/types/cost_result.html
- See http://docs.pgrouting.org/2.2/en/src/tsp/doc/pgr_tsp.html
- For more details, see *tsp_deprecated*.

Use *pgr_eucledianTSP* insteadi of (1). Use *pgr_TSP* instead of (2).

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)[13]

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
    - $(n - 1)!/2$.

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

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.

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.

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.

Deprecated functionality

The old functionality is deprecated:

- User can not control the execution.
- Not all valuable information is returned.
- Some returned column don not have meaningful names.

  Example

Using the old functionality, for example

- *id* can not be of type *BIGINT*.
- *id1* and *id2* are meningless column names.
- Needs an index as parameter for the starting node.

```
SELECT * FROM pgr_TSP(
    $$
    SELECT id::INTEGER, st_X(the_geom) AS x, st_Y(the_geom)AS y  FROM edge_table_vertices_pgr
    $$
```

```
    , 1);
NOTICE:  Deprecated Signature pgr_tsp(sql, integer, integer)
 seq | id1 | id2 |       cost
-----+-----+-----+-------------------
   0 |   1 |   1 |                 1
   1 |   2 |   2 |                 1
   2 |   5 |   5 |                 1
   3 |   8 |   8 |                 1
   4 |   7 |   7 |  1.58113883008419
   5 |  14 |  14 |  1.58113883008419
   6 |  13 |  13 |               0.5
   7 |  15 |  15 |               0.5
   8 |  10 |  10 |                 1
   9 |  11 |  11 |  1.11803398874989
  10 |  17 |  17 |  1.11803398874989
  11 |  12 |  12 | 0.860232526704263
  12 |  16 |  16 |  0.58309518948453
  13 |   6 |   6 |                 1
  14 |   9 |   9 |                 1
  15 |   4 |   4 |                 1
  16 |   3 |   3 |   1.4142135623731
(17 rows)
```

With the new functionality:

- *id* can be of type *BIGINT* .
- There is an aggregate cost column.
- Instead of an index it uses the node identifier for the starting node.

```
SELECT * FROM pgr_eucledianTSP(
    $$
    SELECT id, st_X(the_geom) AS x, st_Y(the_geom)AS y  FROM edge_table_vertices_pgr
    $$,
    1,
    randomize := false
);
 seq | node |       cost        |     agg_cost
-----+------+-------------------+------------------
   1 |    1 |   1.4142135623731 |                0
   2 |    3 |                 1 |  1.4142135623731
   3 |    4 |                 1 | 2.41421356237309
   4 |    9 |                 1 | 3.41421356237309
   5 |    6 |  0.58309518948453 | 4.41421356237309
   6 |   16 | 0.860232526704263 | 4.99730875185763
   7 |   12 |  1.11803398874989 | 5.85754127856189
   8 |   17 |  1.11803398874989 | 6.97557526731178
   9 |   11 |                 1 | 8.09360925606168
  10 |   10 |               0.5 | 9.09360925606168
  11 |   15 |               0.5 | 9.59360925606168
  12 |   13 |  1.58113883008419 | 10.0936092560617
  13 |   14 |  1.58113883008419 | 11.6747480861459
  14 |    7 |                 1 | 13.2558869162301
  15 |    8 |                 1 | 14.2558869162301
  16 |    5 |                 1 | 15.2558869162301
  17 |    2 |                 1 | 16.2558869162301
  18 |    1 |                 0 | 17.2558869162301
(18 rows)
```

Example

Using the old functionality, for example

- *id*, *source*, *target* can not be of type *BIGINT*.
- It does not return the *cost* column.
- Needs an index as parameter for the starting node.
- The identifiers in the result does not correspond to the indentifiers given as input.

```
SELECT * FROM pgr_TSP(
    (SELECT * FROM pgr_vidsToDMatrix(
            'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_ta
            (SELECT array_agg(id) from edge_table_vertices_pgr WHERE id < 14)::INTEGER[], false ,
    ),
    1
);
 seq | id
-----+----
   0 |  1
   1 |  2
   2 |  3
   3 |  8
   4 | 11
   5 |  5
   6 | 10
   7 | 12
   8 |  9
   9 |  6
  10 |  7
  11 |  4
  12 |  0
(13 rows)
```

With the new functionality:

- *id*, *source*, *target* can be of type *BIGINT*,
- There is an aggregate cost column and a cost column in the results.
- Instead of an index it uses the node identifier for the starting node.

```
SELECT * FROM pgr_TSP(
    $$
    SELECT * FROM pgr_dijkstraCostMatrix(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        (SELECT array_agg(id) from edge_table_vertices_pgr WHERE id < 14), false)
    $$,
    1,
    randomize := false
);
 seq | node | cost | agg_cost
-----+------+------+----------
   1 |    1 |    3 |        0
   2 |    4 |    1 |        3
   3 |    9 |    1 |        4
   4 |   12 |    1 |        5
   5 |   11 |    2 |        6
   6 |   13 |    1 |        8
   7 |   10 |    1 |        9
   8 |    5 |    2 |       10
   9 |    7 |    1 |       12
  10 |    8 |    2 |       13
  11 |    6 |    1 |       15
  12 |    3 |    1 |       16
  13 |    2 |    1 |       17
  14 |    1 |    0 |       18
```

```
(14 rows)
```

See Also

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

6.1.10 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 algorithm* 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[]*
- *All pairs* - All pair of vertices.
  - *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* - Dijkstra family functions
  - *pgr_dijkstra* - Dijkstra's shortest path algorithm.
  - *pgr_dijkstraCost* - Use pgr_dijkstra to calculate the costs of the shortest paths.
- *Driving Distance* - Driving Distance
  - *pgr_drivingDistance* - Driving Distance
  - Post processing
    - *pgr_alphaShape* - Alpha shape computation
    - *pgr_pointsAsPolygon* - Polygon around set of points
- *pgr_ksp* - K-Shortest Path

- *pgr_trsp* - Turn Restriction Shortest Path (TRSP)
- *Traveling Sales Person*
  - *pgr_TSP* - When input is a cost matrix.
  - *pgr_eucledianTSP* - When input are coordinates.

CHAPTER 7

Available Functions but not official pgRouting functions

- *Stable proposed Functions*
- *Experimental and Proposed functions*

7.1 Stable proposed Functions

> **Warning:** These are 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.
- A new *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* - Use pgr_withPoints to calculate a cost matrix.
  - *pgr_withPointsKSP - Proposed* - K shortest paths with points.
  - *pgr_withPointsDD - Proposed* - Driving distance.
- A new Section
  - *Cost Matrix*

7.1.1 pgr_dijkstraCostMatrix - proposed

Name

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

Warning: These are 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.

[1]

Fig. 7.1: Boost Graph Inside

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

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 considered as Undirected. Default is `true` which considers the graph as Directed.

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`	Aggregate cost of the shortest path 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*
- *Traveling Sales Person*
- The queries use the *Sample Data* network.

Indices and tables

- genindex
- search

7.1.2 pgr_dijkstraVia - Proposed

Name

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

[2]

Fig. 7.2: Boost Graph Inside

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

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.
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 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 false the graph is considered as Undirected
strict	`BOOLEAN`	(optional) ignores if a subsection of the route is missing and returns everything it found Default is true (is directed). When set to false the graph is considered as Undirected
U_turn_on_edge	`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
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)
```

See Also

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

Indices and tables

- genindex
- search

7.1.3 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: These are 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. 7.3: Boost Graph Inside

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

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

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** • 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: These are 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.

[4]

Fig. 7.4: Boost Graph Inside

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

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** • 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:** These are 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.

[5]

Fig. 7.5: Boost Graph Inside

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

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** • 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 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	Aggregate cost of the shortest path 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*
- *Traveling Sales Person*
- *sampledata* network.

Indices and tables

- genindex
- search

pgr_withPointsKSP - Proposed

Name

`pgr_withPointsKSP` - Find the K shortest paths using Yen's algorithm.

> **Warning:** These are 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.

[6]

Fig. 7.6: Boost Graph Inside

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

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** • 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 |   11 |    1 |      2.6
   6 |       1 |        6 |   11 |   13 |    1 |      3.6
   7 |       1 |        7 |   12 |   15 |  0.6 |      4.6
   8 |       1 |        8 |   -2 |   -1 |    0 |      5.2
   9 |       2 |        1 |   -1 |    1 |  0.6 |        0
  10 |       2 |        2 |    2 |    4 |  0.7 |      0.6
  11 |       2 |        3 |   -6 |    4 |  0.3 |      1.3
  12 |       2 |        4 |    5 |    8 |    1 |      1.6
  13 |       2 |        5 |    6 |    9 |    1 |      2.6
  14 |       2 |        6 |    9 |   15 |    1 |      3.6
  15 |       2 |        7 |   12 |   15 |  0.6 |      4.6
  16 |       2 |        8 |   -2 |   -1 |    0 |      5.2
(16 rows)
```

Example Right side driving topology 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:** These are 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.

[7]

Fig. 7.7: Boost Graph Inside

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, equ
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 |  0.3 |      1.6
   5 |    6 |    8 |    1 |      2.6
   6 |    8 |    7 |    1 |      2.6
   7 |   10 |   10 |    1 |      2.6
   8 |    7 |    6 |  0.3 |      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

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** • 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 *Graph 1: Directed, with cost and reverse cost*

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

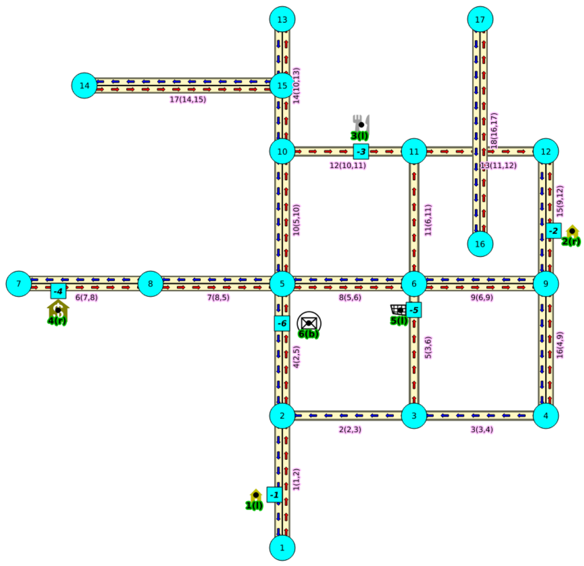
Warning: These are proposed functions for next mayor release.

- 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.

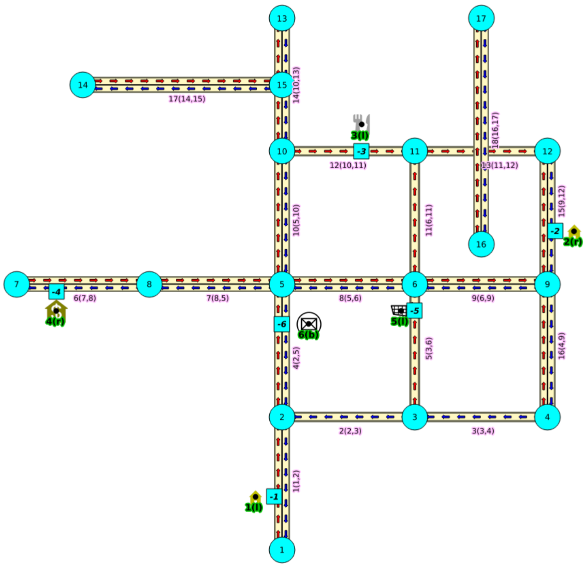
Images

The squared vertices are the temporary vertices, The temporary vertices are added acordng to the dirving side, The following images visualy show the diferences on how depending on the driving side the data is interpreted.

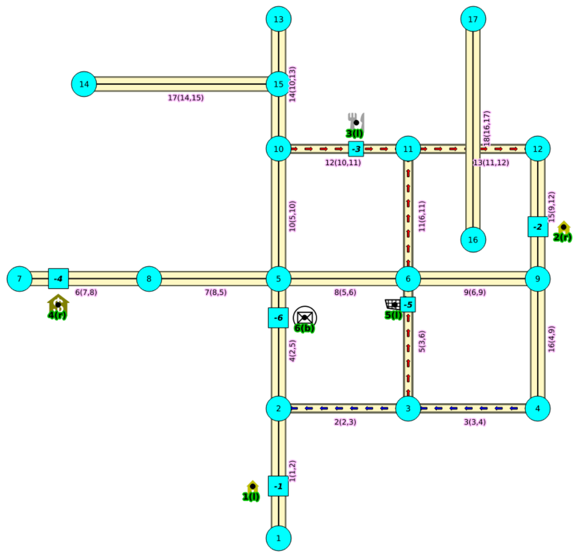
Right driving side



Left driving side



doesn't matter the driving side



Introduction

This famly 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 functios 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 grpah
    - 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: $(pid, eid, fraction, side)$
  - **ped** 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 afects 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 afects 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

7.1.4 Cost Matrix

- *pgr_dijkstraCostMatrix - proposed*
- *pgr_withPointsCostMatrix - proposed*

Warning: These are 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

Sinopsis

Traveling Sales Person 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

7.2 Experimental and Proposed functions

Experimental and Proposed functions

Warning: These are proposed 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

7.2.1 Proposed functions

- *Contraction* - Reduce network size using contraction techniques
  - *pgr_contractGraph - Proposed* - Reduce network size using contraction techniques
- *Maximum Flow*
  - *pgr_maxFlowPushRelabel Proposed* - Maximum flow using push&relabel algorithm.
  - *pgr_maxFlowEdmondsKarp - Proposed* - Maximum flow using Edmonds&Karp algorithm.
  - *pgr_maxFlowBoykovKolmogorov - Proposed* - Maximum flow using Boykov&Kolmogorov algorithm.
- *Applications of Maximum Flow*
  - *pgr_maximumCardinalityMatching - Proposed* - Calculates a maximum cardinality matching.
  - *pgr_edgeDisjointPaths - Proposed* - Calculates edge disjoint paths.

- convenience
  - *pgr_pointToEdgeNode - Proposed* - convert a point geometry to a `vertex_id` based on closest edge.
  - *pgr_pointsToVids - Proposed* - convert an array of point geometries into vertex ids.
- graph analysis
  - *pgr_labelGraph - Proposed* - Analyze / label subgraphs within a network
- Vehicle Routing Problems
  - *pgr_gsoc_vrppdtw - Proposed* - VRP Pickup & Delivery (Euclidean)
  - *pgr_vrpOneDepot - Proposed* - VRP One Depot

Contraction

Warning: These are proposed 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 - Proposed

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` .

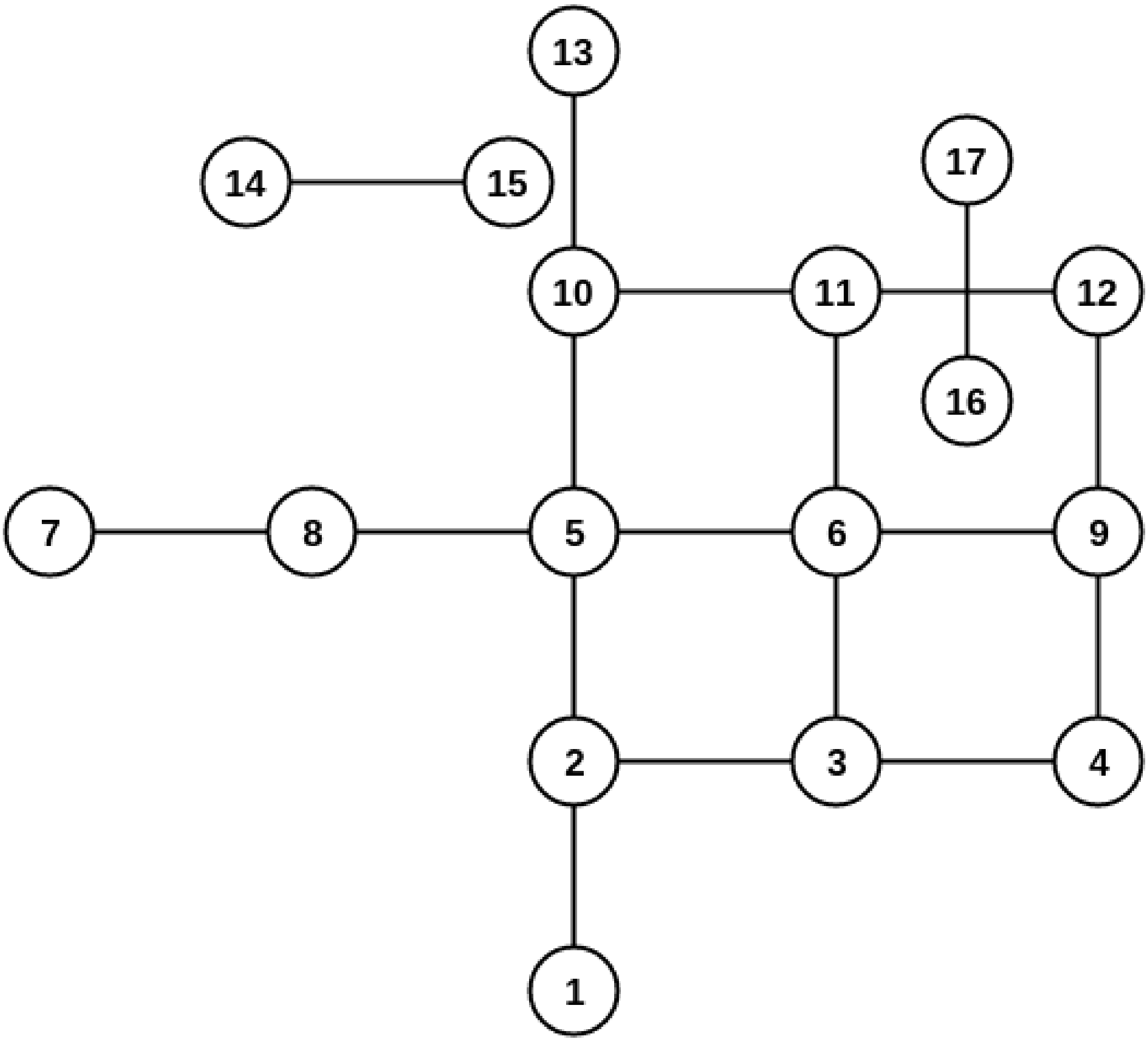
```
<input>
do max_cycles times {
    for (operation in operations_order)
     { do operation }
}
<output>
```

Contracting Sample Data

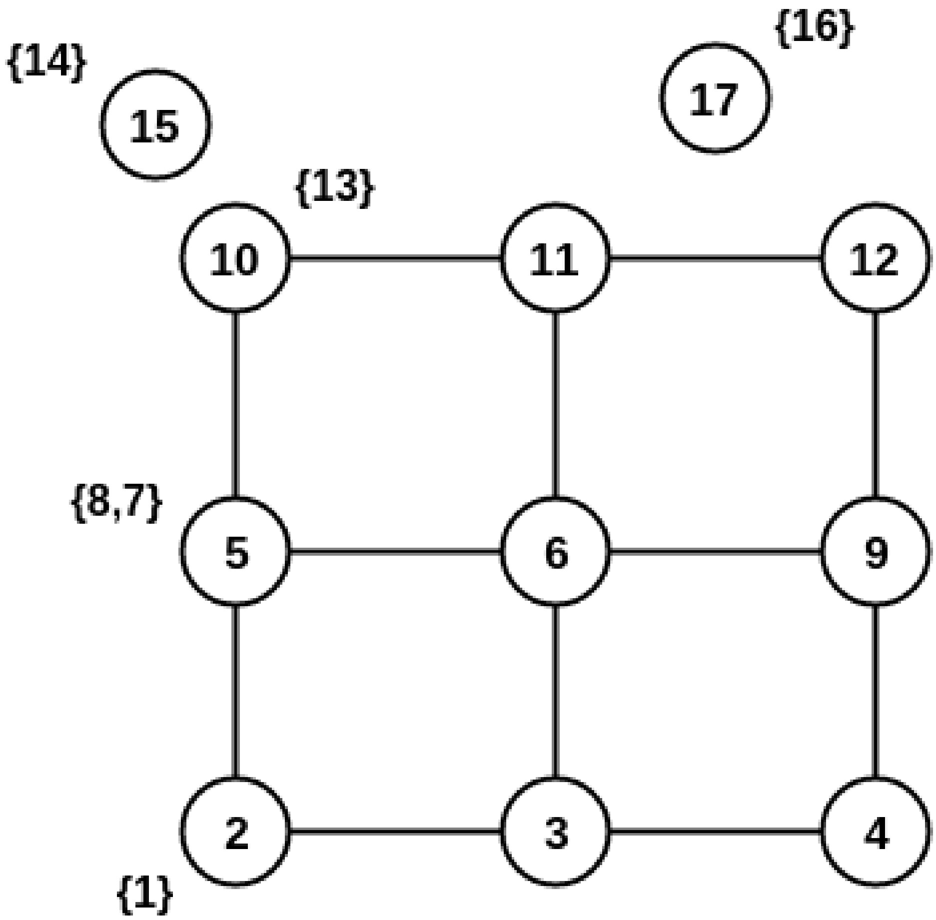
In this section, building and using a contracted graph will be shown by example.

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

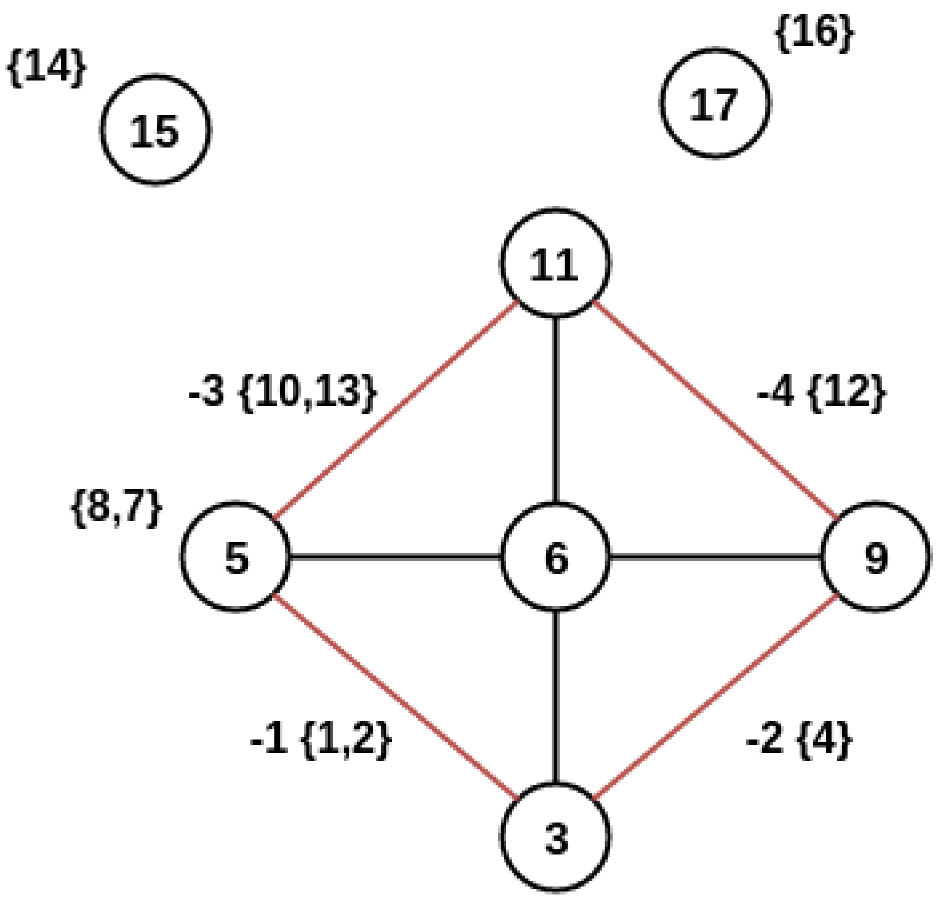
The original graph:



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.

```
SELECT id, source, target, cost, reverse_cost FROM edge_table;
 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
  1 |      1 |      2 |    1 |            1
  2 |      2 |      3 |   -1 |            1
  3 |      3 |      4 |   -1 |            1
  4 |      2 |      5 |    1 |            1
  5 |      3 |      6 |    1 |           -1
  6 |      7 |      8 |    1 |            1
  7 |      8 |      5 |    1 |            1
  8 |      5 |      6 |    1 |            1
  9 |      6 |      9 |    1 |            1
```

```
 10 |       5 |      10 |     1 |             1
 11 |       6 |      11 |     1 |            -1
 12 |      10 |      11 |     1 |            -1
 13 |      11 |      12 |     1 |            -1
 14 |      10 |      13 |     1 |             1
 15 |       9 |      12 |     1 |             1
 16 |       4 |       9 |     1 |             1
 17 |      14 |      15 |     1 |             1
 18 |      16 |      17 |     1 |             1
(18 rows)
```

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
```

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(
        $$
        WITH
        expand_vertices AS (SELECT id, unnest(contracted_vertices) AS vertex FROM edge_table_vert
        expand7 AS (SELECT contracted_vertices FROM edge_table_vertices_pgr
            WHERE id IN (SELECT id FROM expand_vertices WHERE vertex = 7)),
        vertices_in_graph AS (
            SELECT id  FROM edge_table_vertices_pgr WHERE is_contracted = false
            UNION
            SELECT unnest(contracted_vertices) FROM expand7)
        SELECT id, source, target, cost, reverse_cost
        FROM edge_table
        WHERE source IN (SELECT * FROM vertices_in_graph)
        AND target IN (SELECT * FROM vertices_in_graph)
        $$,
        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 - Proposed* function and the *Sample Data* network.

Indices and tables

- genindex
- search

pgr_contractGraph - Proposed

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

Warning: These are proposed 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

[8]

Fig. 7.8: Boost Graph Inside

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

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

Maximum Flow

- *pgr_maxFlowPushRelabel Proposed* - Push and relabel algorithm implementation for maximum flow.
- *pgr_maxFlowEdmondsKarp - Proposed* - Edmonds and Karp algorithm implementation for maximum flow.
- *pgr_maxFlowBoykovKolmogorov - Proposed* - Boykov and Kolmogorov algorithm implementation for maximum flow.

The maximum flow through the graph is guaranteed to be the same with all implementations, but the actual flow through each edge may vary.

Warning: These are proposed 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_maxFlowPushRelabel Proposed

Name `pgr_maxFlowPushRelabel` — Calculates the maximum flow in a directed graph given a source and a destination.

Warning: These are proposed 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

[9]

Fig. 7.9: Boost Graph Inside

Synopsis Calculates the maximum flow in a directed graph from a source node to a sink node. Edges must be weighted with non-negative capacities.

Characteristics:

The main characterics are:

- Calculates the flow/residual capacity for each edge. In the output, edges with zero flow are omitted.
- The maximum flow through the graph can be calculated by aggregation on source/sink.
- Returns nothing if source and sink are the same.
- Allows multiple sources and sinks.
- Running time: $O(V^3)$

Signature Summary

```
pgr_maxFlowPushRelabel(edges_sql, source_vertex,  sink_vertex)
pgr_maxFlowPushRelabel(edges_sql, source_vertices,  sink_vertex)
pgr_maxFlowPushRelabel(edges_sql, source_vertex,  sink_vertices)
pgr_maxFlowPushRelabel(edges_sql, source_vertices,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Signatures

One to One Calculates the maximum flow from one source vertex to one sink vertex in a directed graph.

```
pgr_maxFlowPushRelabel(edges_sql, source_vertex,  sink_vertex)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowPushRelabel(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , 6, 11
);
 seq | edge_id | source | target | flow | residual_capacity
-----+---------+--------+--------+------+-------------------
   1 |      10 |      5 |     10 |  100 |                30
   2 |       8 |      6 |      5 |  100 |                30
   3 |       9 |      6 |      9 |   50 |                80
   4 |      11 |      6 |     11 |  130 |                 0
   5 |      15 |      9 |     12 |   50 |                30
   6 |      12 |     10 |     11 |  100 |                 0
   7 |      13 |     12 |     11 |   50 |                 0
(7 rows)
```

One to Many Ccalculates the maximum flow from one source vertex to many sink vertices in a directed graph.

```
pgr_maxFlowPushRelabel(edges_sql, source_vertex,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowPushRelabel(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , 6, ARRAY[11, 1, 13]
);
 seq | edge_id | source | target | flow | residual_capacity
-----+---------+--------+--------+------+-------------------
   1 |       1 |      2 |      1 |  130 |                 0
   2 |       4 |      2 |      5 |   20 |                80
   3 |       2 |      3 |      2 |  100 |                 0
   4 |       3 |      4 |      3 |   50 |                80
   5 |       4 |      5 |      2 |   50 |                 0
   6 |       7 |      5 |      8 |   50 |                80
   7 |      10 |      5 |     10 |  100 |                30
   8 |       5 |      6 |      3 |   50 |                 0
   9 |       8 |      6 |      5 |  130 |                 0
  10 |       9 |      6 |      9 |  100 |                30
  11 |      11 |      6 |     11 |  130 |                 0
  12 |       6 |      7 |      8 |   50 |                 0
  13 |       6 |      8 |      7 |   50 |                50
  14 |       7 |      8 |      5 |   50 |                 0
  15 |      15 |      9 |     12 |   50 |                30
  16 |      16 |      9 |      4 |   50 |                30
  17 |      12 |     10 |     11 |  100 |                 0
```

```
  18 |      13 |     12 |     11 |   50 |                  0
(18 rows)
```

Many to One Calculates the maximum flow from many source vertices to one sink vertex in a directed graph.

```
pgr_maxFlowPushRelabel(edges_sql, source_vertices,  sink_vertex)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowPushRelabel(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , ARRAY[6, 8, 12], 11
);
 seq | edge_id | source | target | 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
   5 |      13 |     12 |     11 |   50 |                 0
(5 rows)
```

Many to Many Calculates the maximum flow from many sources to many sinks in a directed graph.

```
pgr_maxFlowPushRelabel(edges_sql, source_vertices,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowPushRelabel(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 seq | edge_id | source | target | 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 |       5 |      6 |      3 |   50 |                 0
   6 |       8 |      6 |      5 |  130 |                 0
   7 |       9 |      6 |      9 |   30 |               100
   8 |      11 |      6 |     11 |  130 |                 0
```

```
  9 |       7 |      8 |      5 |   20 |                 30
 10 |      16 |      9 |      4 |   80 |                  0
 11 |      12 |     10 |     11 |  100 |                  0
 12 |      13 |     12 |     11 |   50 |                  0
 13 |      15 |     12 |      9 |   50 |                  0
(13 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.
capacity	`ANY-INTEGER`	Capacity of the edge *(source, target)*. Must be positive.
reverse_-capacity	`ANY-INTEGER`	(optional) Weight of the edge *(target, source)*. Must be positive or null.

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the parameters of the signatures

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
source_vertex	`BIGINT`	Identifier of the source vertex(or vertices).
sink_vertex	`BIGINT`	Identifier of the sink vertex(or vertices).

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

- *Maximum Flow*
- 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_maxFlowEdmondsKarp - Proposed

Name `pgr_maxFlowEdmondsKarp` — Calculates the maximum flow in a directed graph given a source and a destination. Implemented by Boost Graph Library.

> **Warning:** These are proposed 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

[10]

Fig. 7.10: Boost Graph Inside

Synopsis Calculates the maximum flow in a directed graph from a source node to a sink node. Edges must be weighted with non-negative capacities. Developed by Edmonds and Karp.

Characteristics:

The main characterics are:

- The graph must be directed.
- Calculates the flow/residual capacity for each edge. In the output, edges with zero flow are omitted.
- The maximum flow through the graph can be calculated by aggregation on source/sink.
- Returns nothing if source and sink are the same.
- Allows multiple sources and sinks (See signatures below).
- Running time: $O(V * E^2)$.

Signature Summary

```
pgr_maxFlowEdmondsKarp(edges_sql, source_vertex,  sink_vertex)
pgr_maxFlowEdmondsKarp(edges_sql, source_vertices,  sink_vertex)
pgr_maxFlowEdmondsKarp(edges_sql, source_vertex,  sink_vertices)
pgr_maxFlowEdmondsKarp(edges_sql, source_vertices,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Signatures

One to One Calculates the maximum flow from one source vertex to one sink vertex on a *directed* graph.

```
pgr_maxFlowEdmondsKarp(edges_sql, source_vertex,  sink_vertex)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowEdmondsKarp(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , 6, 11
);
 seq | edge_id | source | target | flow | residual_capacity
-----+---------+--------+--------+------+-------------------
   1 |      10 |      5 |     10 |  100 |                30
   2 |       8 |      6 |      5 |  100 |                30
   3 |       9 |      6 |      9 |   50 |                80
   4 |      11 |      6 |     11 |  130 |                 0
   5 |      15 |      9 |     12 |   50 |                30
   6 |      12 |     10 |     11 |  100 |                 0
   7 |      13 |     12 |     11 |   50 |                 0
(7 rows)
```

One to Many Calculates the maximum flow from one source vertex to many sink vertices on a *directed* graph.

```
pgr_maxFlowEdmondsKarp(edges_sql, source_vertex,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowEdmondsKarp(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
   , 6, ARRAY[1, 3, 11]
);
 seq | edge_id | source | target | 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 |       5 |      6 |      3 |   50 |                 0
   6 |       8 |      6 |      5 |  130 |                 0
   7 |       9 |      6 |      9 |  130 |                 0
   8 |      11 |      6 |     11 |  130 |                 0
   9 |      15 |      9 |     12 |   50 |                30
  10 |      16 |      9 |      4 |   80 |                 0
  11 |      12 |     10 |     11 |   80 |                20
```

```
  12 |       13 |      12 |      11 |   50 |                  0
(12 rows)
```

Many to One Calculates the maximum flow from many source vertices to one sink vertex on a *directed* graph.

```
pgr_maxFlowEdmondsKarp(edges_sql, source_vertices,  sink_vertex)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowEdmondsKarp(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
   , ARRAY[6, 8, 12], 11
);
 seq | edge_id | source | target | 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
   5 |      13 |     12 |     11 |   50 |                 0
(5 rows)
```

Many to Many Calculates the maximum flow from many sources to many sinks on a *directed* graph.

```
pgr_maxFlowEdmondsKarp(edges_sql, source_vertices,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowEdmondsKarp(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
   , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 seq | edge_id | source | target | 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 |       5 |      6 |      3 |   50 |                 0
   6 |       8 |      6 |      5 |  130 |                 0
   7 |       9 |      6 |      9 |   80 |                50
   8 |      11 |      6 |     11 |  130 |                 0
```

```
   9 |        7 |       8 |       5 |   20 |                 30
  10 |       16 |       9 |       4 |   80 |                  0
  11 |       12 |      10 |      11 |  100 |                  0
  12 |       13 |      12 |      11 |   50 |                  0
(12 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.
capacity	ANY-INTEGER	Capacity of the edge *(source, target)*. Must be positive.
reverse_-capacity	ANY-INTEGER	(optional) Weight of the edge *(target, source)*. Must be positive or null.

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the parameters of the signatures

Column	Type	Description
edges_sql	TEXT	SQL query as described above.
source_vertex	BIGINT	Identifier of the source vertex(or vertices).
sink_vertex	BIGINT	Identifier of the sink vertex(or vertices).

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

See Also

- *Maximum Flow*
- 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_maxFlowBoykovKolmogorov - Proposed

Name `pgr_maxFlowBoykovKolmogorov` — Calculates the maximum flow in a directed graph given a source and a destination. Implemented by Boost Graph Library.

Warning: These are proposed 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

[11]

Fig. 7.11: Boost Graph Inside

Synopsis Calculates the maximum flow in a directed graph from a source node to a sink node. Edges must be weighted with non-negative capacities. Developed by Boykov and Kolmogorov.

Characteristics:

The main characterics are:

- The graph must be directed.
- Calculates the flow/residual capacity for each edge. In the output, edges with zero flow are omitted.
- The maximum flow through the graph can be calculated by aggregation on source/sink.
- Returns nothing if source and sink are the same.
- Allows multiple sources and sinks (See signatures below).
- Running time: in general polynomial complexity, performs well on graphs that represent 2D grids (eg.: roads).

Signature Summary

```
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertex,  sink_vertex)
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertices,  sink_vertex)
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertex,  sink_vertices)
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertices,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Signatures

One to One The available signature calculates the maximum flow from one source vertex to one sink vertex.

```
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertex,  sink_vertex)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowBoykovKolmogorov(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , 6, 11
);
 seq | edge_id | source | target | flow | residual_capacity
-----+---------+--------+--------+------+-------------------
   1 |      10 |      5 |     10 |  100 |                30
   2 |       8 |      6 |      5 |  100 |                30
   3 |       9 |      6 |      9 |   50 |                80
   4 |      11 |      6 |     11 |  130 |                 0
   5 |      15 |      9 |     12 |   50 |                30
   6 |      12 |     10 |     11 |  100 |                 0
   7 |      13 |     12 |     11 |   50 |                 0
(7 rows)
```

One to Many The available signature calculates the maximum flow from one source vertex to many sink vertices.

```
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertex,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowBoykovKolmogorov(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , 6, ARRAY[1, 3, 11]
);
 seq | edge_id | source | target | 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 |       5 |      6 |      3 |   50 |                 0
   6 |       8 |      6 |      5 |  130 |                 0
   7 |       9 |      6 |      9 |  130 |                 0
   8 |      11 |      6 |     11 |  130 |                 0
   9 |      15 |      9 |     12 |   50 |                30
  10 |      16 |      9 |      4 |   80 |                 0
```

```
  11 |      12 |      10 |      11 |   80 |                 20
  12 |      13 |      12 |      11 |   50 |                  0
(12 rows)
```

Many to One The available signature calculates the maximum flow from many source vertices to one sink vertex.

```
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertices,  sink_vertex)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowBoykovKolmogorov(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , ARRAY[6, 8, 12], 11
);
 seq | edge_id | source | target | 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
   5 |      13 |     12 |     11 |   50 |                 0
(5 rows)
```

Many to Many The available signature calculates the maximum flow from many sources to many sinks.

```
pgr_maxFlowBoykovKolmogorov(edges_sql, source_vertices,  sink_vertices)
RETURNS SET OF (id, edge_id, source, target, flow, residual_capacity)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_maxFlowBoykovKolmogorov(
    'SELECT id,
            source,
            target,
            c1.capacity as capacity,
            c2.capacity as reverse_capacity
    FROM edge_table JOIN categories AS c1 USING(category_id), categories AS c2
    WHERE edge_table.reverse_category_id = c2.category_id
    ORDER BY id'
    , ARRAY[6, 8, 12], ARRAY[1, 3, 11]
);
 seq | edge_id | source | target | 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 |       5 |      6 |      3 |   50 |                 0
   6 |       8 |      6 |      5 |  130 |                 0
```

```
   7 |       9 |      6 |      9 |   80 |                 50
   8 |      11 |      6 |     11 |  130 |                  0
   9 |       7 |      8 |      5 |   20 |                 30
  10 |      16 |      9 |      4 |   80 |                  0
  11 |      12 |     10 |     11 |  100 |                  0
  12 |      13 |     12 |     11 |   50 |                  0
(12 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.
capacity	`ANY-INTEGER`	Capacity of the edge *(source, target)*. Must be positive.
reverse_-capacity	`ANY-INTEGER`	(optional) Weight of the edge *(target, source)*. Must be positive or null.

Where:

ANY-INTEGER SMALLINT, INTEGER, BIGINT

Description of the parameters of the signatures

Column	Type	Description
edges_sql	`TEXT`	SQL query as described above.
source_vertex	`BIGINT`	Identifier of the source vertex(or vertices).
sink_vertex	`BIGINT`	Identifier of the sink vertex(or vertices).

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

- http://www.boost.org/libs/graph/doc/boykov_kolmogorov_max_flow.html
- http://www.csd.uwo.ca/~yuri/Papers/pami04.pdf

Indices and tables

- genindex
- search

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.

Given the following query:

pgr_maxFlow $(edges\_sql, source\_vertex, sink\_vertex)$

where $edges\_sql = \{(id_i, source_i, target_i, capacity_i, reverse\_capacity_i)\}$

Graph definition

The weighted directed graph, $G(V, E)$, is defined as:

- the set of vertices V
  - $source\_vertex \cup sink\_vertex \bigcup source_i \bigcup target_i$
- the set of edges E
  - $E = \begin{cases} \text{if } reverse\_capacity = & \{(source_i, target_i, capacity_i) \text{ when } capacity > 0\} \\ & \{(source_i, target_i, capacity_i) \text{ when } capacity > 0\} \\ \cup & \{(target_i, source_i, reverse\_capacity_i) \text{ when } reverse\_capacity_i > 0)\} \\ \text{if } reverse\_capacity \neq & \end{cases}$

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

Applications of Maximum Flow

- *pgr_maximumCardinalityMatching - Proposed* - Calculates a maximum cardinality matching in a graph.
- *pgr_edgeDisjointPaths - Proposed* - Calculates edge disjoint paths between two groups of vertices.

Maximum flow algorithms provide solutions to other graph problems.

Warning: These are proposed 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_maximumCardinalityMatching - Proposed

Name `pgr_maximumCardinalityMatching` — Calculates a maximum cardinality matching in a graph.

Warning: These are proposed 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

[12]

Fig. 7.12: Boost Graph Inside

Synopsis Calculates a maximum cardinality matching in a directed/undirected graph.

- 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.

Characteristics:

The main characterics are:

- 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[13].

Signature Summary

```
pgr_MaximumCardinalityMatching(edges_sql)
pgr_MaximumCardinalityMatching(edges_sql, directed)

RETURNS SET OF (id, edge_id, source, target)
    OR EMPTY SET
```

Signatures

Minimal signature

```
pgr_MaximumCardinalityMatching(edges_sql)
RETURNS SET OF (id, edge_id, source, target) OR EMPTY SET
```

The minimal signature calculates one possible maximum cardinality matching on a *directed* graph.

Example

```
SELECT * FROM pgr_maximumCardinalityMatching(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table'
);
 seq | edge_id | 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 (id, edge_id, source, target) OR EMPTY SET
```

The complete signature calculates one possible maximum cardinality matching.

Example

```
SELECT * FROM pgr_maximumCardinalityMatching(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    directed := false
);
```

[13] https://en.wikipedia.org/wiki/Ackermann_function

```
 seq | edge_id | source | target
-----+---------+--------+--------
   1 |       1 |      1 |      2
   2 |       3 |      3 |      4
   3 |       9 |      6 |      9
   4 |       6 |      7 |      8
   5 |      14 |     10 |     13
   6 |      13 |     11 |     12
   7 |      17 |     14 |     15
   8 |      18 |     16 |     17
(8 rows)
```

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_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.

See Also

- *Applications of Maximum Flow*
- 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.

Warning: These are proposed 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

[14]

Fig. 7.13: Boost Graph Inside

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_maxFlowBoykovKolmogorov - Proposed* to calculate the paths.
- No *cost* or *aggregate cost* of the paths are returned. (Under discussion)

Signature Summary

```
pgr_edgeDisjointPaths(edges_sql, source_vertex, destination_vertex)
pgr_edgeDisjointPaths(edges_sql, source_vertex, destination_vertex, directed)
pgr_edgeDisjointPaths(edges_sql, source_vertices, destination_vertex, directed)
pgr_edgeDisjointPaths(edges_sql, source_vertex, destination_vertices, directed)
pgr_edgeDisjointPaths(edges_sql, source_vertices, destination_vertices, directed)

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

Signatures

Minimal signature

```
pgr_edgeDisjointPaths(edges_sql, source_vertex, destination_vertex)
RETURNS SET OF (seq, path_seq, node, edge) OR EMPTY SET
```

The minimal signature is between *source_vertex* and *destination_vertex* for a *directed* graph.

Example

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    3, 5
);
 seq | path_seq | node | edge
-----+----------+------+------
   1 |        1 |    3 |    2
   2 |        2 |    2 |    4
   3 |        3 |    5 |   -1
   4 |        1 |    3 |    5
   5 |        2 |    6 |    8
   6 |        3 |    5 |   -1
(6 rows)
```

One to One The available signature calculates edge disjoint paths from one source vertex to one destination vertex. The graph can be directed or undirected.

```
pgr_edgeDisjointPaths(edges_sql, source_vertex, destination_vertex, directed)
RETURNS SET OF (seq, path_seq, node, edge) OR EMPTY SET
```

Example

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    3, 5,
    directed := false
);
 seq | path_seq | node | edge
-----+----------+------+------
   1 |        1 |    3 |    2
   2 |        2 |    2 |    4
   3 |        3 |    5 |   -1
   4 |        1 |    3 |    3
   5 |        2 |    4 |   16
   6 |        3 |    9 |    9
   7 |        4 |    6 |    8
   8 |        5 |    5 |   -1
   9 |        1 |    3 |    5
  10 |        2 |    6 |   11
  11 |        3 |   11 |   12
  12 |        4 |   10 |   10
  13 |        5 |    5 |   -1
(13 rows)
```

One to Many The available signature calculates the maximum flow from one source vertex to many sink vertices.

```
pgr_edgeDisjointPaths(edges_sql, source_vertex, destination_vertices, directed)
RETURNS SET OF (seq, path_seq, end_vid, node, edge) OR EMPTY SET
```

Example

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    3, ARRAY[4, 5, 10]
);
 seq | path_seq | end_vid | node | edge
-----+----------+---------+------+------
   1 |        1 |       5 |    3 |    2
   2 |        2 |       5 |    2 |    4
   3 |        3 |       5 |    5 |   -1
   4 |        1 |       5 |    3 |    5
   5 |        2 |       5 |    6 |    8
   6 |        3 |       5 |    5 |   -1
(6 rows)
```

Many to One The available signature calculates the maximum flow from many source vertices to one sink vertex.

```
pgr_edgeDisjointPaths(edges_sql, source_vertices, destination_vertex)
RETURNS SET OF (seq, path_seq, start_vid, node, edge)
  OR EMPTY SET
```

Example

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    ARRAY[3, 6], 5
);
 seq | path_seq | start_vid | node | edge
-----+----------+-----------+------+------
   1 |        1 |         3 |    3 |    2
   2 |        2 |         3 |    2 |    4
   3 |        3 |         3 |    5 |   -1
   4 |        1 |         6 |    6 |    8
   5 |        2 |         6 |    5 |   -1
(5 rows)
```

Many to Many The available signature calculates the maximum flow from many sources to many sinks.

```
pgr_edgeDisjointPaths(edges_sql, source_vertices, destination_vertices, directed)
RETURNS SET OF (seq, path_seq, start_vid, end_vid, node, edge) OR EMPTY SET
```

Example

```
SELECT * FROM pgr_edgeDisjointPaths(
    'SELECT id, source, target, cost AS going, reverse_cost AS coming FROM edge_table',
    ARRAY[3, 6], ARRAY[4, 5, 10]
);
 seq | path_seq | start_vid | end_vid | node | edge
-----+----------+-----------+---------+------+------
   1 |        1 |         3 |       5 |    3 |    2
   2 |        2 |         3 |       5 |    2 |    4
   3 |        3 |         3 |       5 |    5 |   -1
   4 |        1 |         6 |       5 |    6 |    8
   5 |        2 |         6 |       5 |    5 |   -1
   6 |        1 |         6 |       4 |    6 |    9
   7 |        2 |         6 |       4 |    9 |   16
   8 |        3 |         6 |       4 |    4 |   -1
(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.
source_vertex	BIGINT	Identifier(s) of the source vertex(vertices).
sink_vertex	BIGINT	Identifier(s) of the destination vertex(vertices).
directed	BOOLEAN	(optional) Determines the type of the graph. D

Description of the return values

Column	Type	Description
seq	INT	Sequential value starting from **1**.
path_seq	INT	Relative position in the path. Has value **1** for the beginning of a path.
start_vid	BIGINT	Identifier of the starting vertex. Used when multiple starting vertices are i
end_vid	BIGINT	Identifier of the ending vertex. Used when multiple ending vertices are in
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 s the last node of the path.

Indices and tables

- genindex
- search

Applications

Maximum cardinality matching

- 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.
- The graph can be directed or undirected.

The *pgr_maximumCardinalityMatching - Proposed* function can be used to calculate one such maximum matching.

Edge disjoint paths In a undirected/directed graph, two paths are edge-disjoint(or edge-independant) if they do not have any internal edge in common.

While the number of maximum edge disjoint paths is fixed, there may be several different routes.

The *pgr_edgeDisjointPaths - Proposed* function returns the maximum number of paths and possible routes.

See Also

- https://en.wikipedia.org/wiki/Maximum_flow_problem#Application

pgr_pointToEdgeNode - Proposed

Name

`pgr_pointToEdgeNode` - Converts a point to a `vertex_id` based on closest edge.

> **Warning:** These are proposed 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 function returns:

- `integer` that is the vertex id of the closest edge in the `edges` table within the `tol` tolerance of `pnt`. The vertex is selected by projection the `pnt` onto the edge and selecting which vertex is closer along the edge.

```
integer pgr_pointToEdgeNode(edges text, pnt geometry, tol float8)
```

Description

Given an table `edges` with a spatial index on `the_geom` and a point geometry search for the closest edge within `tol` distance to the edges then compute the projection of the point onto the line segment and select source or target based on whether the projected point is closer to the respective end and return the source or target value.

Parameters

The function accepts the following parameters:

edges `text` The name of the edge table or view. (may contain the schema name AS well).

pnt `geometry` A point geometry object in the same SRID as `edges`.

tol `float8` The maximum search distance for an edge.

> **Warning:** If no edge is within tol distance then return -1

The `edges` table must have the following columns:

- `source`
- `target`
- `the_geom`

History

- Proposed in version 2.1.0

Examples

```
SELECT * FROM pgr_pointtoedgenode('edge_table', 'POINT(2 0)'::geometry, 0.02);
 pgr_pointtoedgenode
---------------------
                   1
(1 row)

SELECT * FROM pgr_pointtoedgenode('edge_table', 'POINT(3 2)'::geometry, 0.02);
 pgr_pointtoedgenode
---------------------
                   6
(1 row)
```

The example uses the *Sample Data* network.

See Also

- *pgr_pointsToVids - Proposed* - convert an array of point geometries into vertex ids.

Indices and tables

- genindex
- search

pgr_pointsToVids - Proposed

Name

`pgr_pointsToVids` - Converts an array of point geometries into vertex ids.

Warning: These are proposed 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 an array of point geometries and an edge table and a max search tol distance the function converts points into vertex ids using pgr_pointtoedgenode().

The function returns:

- `integer[]` - An array of `vertex_id`.

```
integer[] pgr_pointsToVids(pnts geometry[], edges text, tol float8 DEFAULT(0.01))
```

Description

Parameters

pnts `geometry[]` - An array of point geometries.

edges `text` - The edge table to be used for the conversion.

tol `float8` - The maximum search distance for locating the closest edge.

Warning: You need to check the results for any vids=-1 which indicates if failed to locate an edge.

History

- Proposed in version 2.1.0

Examples

```
SELECT * FROM pgr_pointstovids(
    pgr_texttopoints('2,0;2,1;3,1;2,2', 0),
    'edge_table'
);
NOTICE:  Deperecated function: pgr_textToPoints
 pgr_pointstovids
------------------
 {1,2,3,5}
(1 row)
```

This example uses the *Sample Data* network.

See Also

- *pgr_pointToEdgeNode - Proposed* - convert a point geometry to the closest vertex_id of an edge..

Indices and tables

- genindex
- search

pgr_labelGraph - Proposed

Name

`pgr_labelGraph` — Locates and labels sub-networks within a network which are not topologically connected.

Warning: These are proposed 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.

```
SELECT pgr_labelGraph('edge_table', 'id', 'source', 'target', 'subgraph');
 pgr_labelgraph
----------------
 OK
(1 row)

SELECT subgraph, count(*) FROM edge_table group by subgraph;
 subgraph | count
----------+-------
        1 |    16
        3 |     1
        2 |     1
(3 rows)
```

See Also

- pgr_createTopology[15] to create the topology of a table based on its geometry and tolerance value.

[15] https://github.com/Zia-/pgrouting/blob/develop/src/common/sql/pgrouting_topology.sql

pgr_gsoc_vrppdtw - Proposed

Name

`pgr_gsoc_vrppdtw` — Returns a solution for *Pick and Delivery* with *time windows* Vehicle Routing Problem

Warning: These are proposed 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 cust • A value of (starting locati
x	ANY-NUMERICAL	X coordinate of the l
y	ANY-NUMERICAL	Y coordinate of the l
demand	ANY-NUMERICAL	How much is added the vehicle. • Negative valu • Positive value
openTime	ANY-NUMERICAL	The time relative to (tomer opens.
closeTime	ANY-NUMERICAL	The time relative to (tomer closes.
serviceTime	ANY-NUMERICAL	The duration of the loading.
pIndex	ANY-INTEGER	Value used when t tomer is a Delivery responding Pickup
dIndex	ANY-INTEGER	Value used when t tomer is a Pickup to sponding 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. (cur
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 |   5 | 105.132745950422
   3 |   1 |   3 | 196.132745950422
   4 |   1 |   7 | 288.132745950422
   5 |   1 |   8 | 380.961173075168
   6 |   1 |  10 | 474.566724350632
   7 |   1 |  11 | 567.566724350632
   8 |   1 |   9 |   660.7290020108
   9 |   1 |   6 |   752.9650699883
  10 |   1 |   4 |   845.2011379658
  11 |   1 |   2 | 938.806689241264
  12 |   1 |   1 | 1030.80668924126
  13 |   1 |  75 | 1123.80668924126
  14 |   1 |   0 | 1139.61807754211
  51 |   5 |   0 |                0
  52 |   5 |  43 | 106.552945357247
  53 |   5 |  42 | 199.552945357247
  54 |   5 |  41 | 291.552945357247
  55 |   5 |  40 | 383.552945357247
  56 |   5 |  44 | 476.552945357247
  57 |   5 |  46 | 569.381372481993
  58 |   5 |  45 | 661.381372481993
  59 |   5 |  48 | 753.381372481993
  60 |   5 |  51 | 756.381372481993
  61 |   5 | 101 | 846.381372481993
  62 |   5 |  50 | 938.617440459493
  63 |   5 |  52 | 1031.77971811966
  64 |   5 |  49 | 1124.77971811966
  65 |   5 |  47 | 1216.77971811966
  66 |   5 |   0 | 1234.80747449698
 103 |   9 |   0 |                0
 104 |   9 |  90 | 110.615528128088
 105 |   9 |  87 | 205.615528128088
 106 |   9 |  86 | 296.615528128088
 107 |   9 |  83 | 392.615528128088
 108 |   9 |  82 | 485.615528128088
 109 |   9 |  84 | 581.446480022934
 110 |   9 |  85 |  674.27490714768
 111 |   9 |  88 |  767.27490714768
 112 |   9 |  89 | 860.103334272426
 113 |   9 |  91 |  953.70888554789
 114 |   9 |   0 | 976.069565322888
(42 rows)
```

See Also

- The examples use *Pick & Deliver Data*

- http://en.wikipedia.org/wiki/Vehicle_routing_problem

pgr_vrpOneDepot - Proposed

Warning: These are proposed 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 from the original developer

- *pgr_costResult[]*
- http://en.wikipedia.org/wiki/Vehicle_routing_problem

Discontinued & Deprecated Functions

- *Discontinued Functions*
- *Deprecated Functions*

8.1 Discontinued Functions

Especially with new major releases functionality may change and functions may be discontinued for various reasons. Functionality that has been discontinued will be listed here.

8.1.1 Shooting Star algorithm

Version Discontinued on 2.0.0

Reasons Unresolved bugs, no maintainer, replaced with *pgr_trsp - Turn Restriction Shortest Path (TRSP)*

Comment Please *contact us* if you're interested to sponsor or maintain this algorithm.

8.2 Deprecated Functions

Warning: These functions are deprecated!!!

- That means they have been replaced by new functions or are no longer supported, and may be removed from future versions.
- All code that uses the functions should be converted to use its replacement if one exists.

8.2.1 Deprecated on version 2.3

Routing functions

- *pgr_astar - Deprecated Signature* - See new signatures of *pgr_aStar*
- *pgr_tsp -Deprecated Signatures* - See new signatures of *Traveling Sales Person*

Auxiliary functions

- *pgr_flipEdges - Deprecated Function*
- *pgr_vidsToDMatrix - Deprecated Function*

- *pgr_vidsToDMatrix - Deprecated Function*
- *pgr_pointsToDMatrix - Deprecated Function*
- *pgr_textToPoints - Deprecated Function*

pgr_astar - Deprecated Signature

> **Warning:** This function signature is deprecated!!!
> - That means it has been replaced by new signature(s)
> - This signature is no longer supported, and may be removed from future versions.
> - All code that use this function signature should be converted to use its replacement *pgr_aStar*.

Name `pgr_astar` — Returns the shortest path using A* algorithm.

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

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

Description

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

```
        SELECT id, source, target, cost, x1, y1, x2, y2 [,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.

x1 `x` coordinate of the start point of the edge

y1 `y` coordinate of the start point of the edge

x2 `x` coordinate of the end point of the edge

y2 `y` coordinate of the end point of the edge

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` id of the start point

target `int4` 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.

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

- Renamed in version 2.0.0

Examples

- Without `reverse_cost`

```
SELECT * FROM pgr_AStar(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, x1, y1, x2, y2
     FROM edge_table',
    4, 1, false, false);
NOTICE:  Deprecated signature of function pgr_astar
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   4 |  16 |    1
   1 |   9 |   9 |    1
   2 |   6 |   8 |    1
   3 |   5 |   4 |    1
   4 |   2 |   1 |    1
   5 |   1 |  -1 |    0
(6 rows)
```

- With `reverse_cost`

```
SELECT * FROM pgr_AStar(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, x1, y1, x2, y2, reverse_cost
      FROM edge_table ',
    4, 1, true, true);
NOTICE:  Deprecated signature of function pgr_astar
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   4 |   3 |    1
   1 |   3 |   2 |    1
   2 |   2 |   1 |    1
   3 |   1 |  -1 |    0
(4 rows)
```

The queries use the *Sample Data* network.

See Also

- *pgr_aStar*
- *pgr_costResult[]*
- http://en.wikipedia.org/wiki/A*_search_algorithm

pgr_tsp -Deprecated Signatures

Warning: These functions signatures are deprecated!!!
- That means they has been replaced by new signatures.
- These signatures are no longer supported, and may be removed from future versions.
- All code that use these functions signatures should be converted to use its replacement.

Name

- `pgr_tsp` - Returns the best route from a start node via a list of nodes.

> **Warning:** Use *pgr_eucledianTSP* instead.

- `pgr_tsp` - Returns the best route order when passed a disance matrix.

> **Warning:** Use *pgr_TSP* instead.

- `_pgr_makeDistanceMatrix` - Returns a Eucleadian distance Matrix from the points provided in the sql result.

> **Warning:** There is no replacement.

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 algorithm uses simulated annealing to return a high quality approximate solution. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows, that make up a path.

```
pgr_costResult[] pgr_tsp(sql text, start_id integer);
pgr_costResult[] pgr_tsp(sql text, start_id integer, end_id integer);
```

Returns a set of (seq integer, id1 integer, id2 integer, cost float8) that is the best order to visit the nodes in the matrix. `id1` is the index into the distance matrix. `id2` is the point id from the sql.

If no `end_id` is supplied or it is -1 or equal to the start_id then the TSP result is assumed to be a circluar loop returning back to the start. If `end_id` is supplied then the route is assumed to start and end the the designated ids.

```
record[] pgr_tsp(matrix float[][], start integer)
record[] pgr_tsp(matrix float[][], start integer, end integer)
```

Description

With Euclidean distances

The TSP solver is based on ordering the points using straight line (euclidean) distance [1] between nodes. The implementation is using an approximation algorithm that is very fast. It is not an exact solution, but it is guaranteed that a solution is returned after certain number of iterations.

sql 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

start_id `int4` id of the start point

end_id `int4` id of the end point, This is *OPTIONAL*, if include the route is optimized from start to end, otherwise it is assumed that the start and the end are the same point.

The function returns set of *pgr_costResult[]*:

[1] There was some thought given to pre-calculating the driving distances between the nodes using Dijkstra, but then I read a paper (unfortunately I don't remember who wrote it), where it was proved that the quality of TSP with euclidean distance is only slightly worse than one with real distance in case of normal city layout. In case of very sparse network or rivers and bridges it becomes more inaccurate, but still wholly satisfactory. Of course it is nice to have exact solution, but this is a compromise between quality and speed (and development time also). If you need a more accurate solution, you can generate a distance matrix and use that form of the function to get your results.

seq row sequence

id1 internal index to the distance matric

id2 `id` of the node

cost cost to traverse from the current node to the next node.

Create a distance matrix

For users that need a distance matrix we have a simple function that takes SQL in `sql` as described above and returns a record with `dmatrix` and `ids`.

```
SELECT dmatrix, ids from _pgr_makeDistanceMatrix('SELECT id, x, y FROM vertex_table');
```

The function returns a record of `dmatrix`, `ids`:

dmatrix `float8[][]` a symeteric Euclidean distance matrix based on `sql`.

ids `integer[]` an array of ids as they are ordered in the distance matrix.

With distance matrix

For users, that do not want to use Euclidean distances, we also provode the ability to pass a distance matrix that we will solve and return an ordered list of nodes for the best order to visit each. It is up to the user to fully populate the distance matrix.

matrix `float[][]` distance matrix of points

start `int4` index of the start point

end `int4` (optional) index of the end node

The `end` node is an optional parameter, you can just leave it out if you want a loop where the `start` is the depot and the route returns back to the depot. If you include the `end` parameter, we optimize the path from `start` to `end` and minimize the distance of the route while include the remaining points.

The distance matrix is a multidimensional PostgreSQL array type[1] that must be `N x N` in size.

The result will be N records of `[ seq, id ]`:

seq row sequence

id index into the matrix

History

- Renamed in version 2.0.0
- GAUL dependency removed in version 2.0.0

Examples

- Using SQL parameter (all points from the table, atarting from 6 and ending at 5). We have listed two queries in this example, the first might vary from system to system because there are multiple equivalent answers. The second query should be stable in that the length optimal route should be the same regardless of order.

```
CREATE TABLE vertex_table (
    id serial,
    x double precision,
    y double precision
);
```

[1] http://www.postgresql.org/docs/9.1/static/arrays.html

```
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);

SELECT seq, id1, id2, round(cost::numeric, 2) AS cost
  FROM pgr_tsp('SELECT id, x, y FROM vertex_table ORDER BY id', 6, 5);

 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   5 |   6 | 1.00
   1 |   6 |   7 | 1.00
   2 |   7 |   8 | 1.41
   3 |   1 |   2 | 1.00
   4 |   0 |   1 | 1.41
   5 |   2 |   3 | 1.00
   6 |   3 |   4 | 1.00
   7 |   8 |   9 | 1.00
   8 |  11 |  12 | 1.00
   9 |  10 |  11 | 1.41
  10 |  12 |  13 | 1.00
  11 |   9 |  10 | 2.24
  12 |   4 |   5 | 1.00
(13 rows)

SELECT round(sum(cost)::numeric, 4) as cost
FROM pgr_tsp('SELECT id, x, y FROM vertex_table ORDER BY id', 6, 5);

  cost
---------
 15.4787
(1 row)
```

- Using distance matrix (A loop starting from 1)

When using just the start node you are getting a loop that starts with 1, in this case, and travels through the other nodes and is implied to return to the start node from the last one in the list. Since this is a circle there are at least two possible paths, one clockwise and one counter-clockwise that will have the same length and be equall valid. So in the following example it is also possible to get back a sequence of ids = {1,0,3,2} instead of the {1,2,3,0} sequence listed below.

```
SELECT seq, id FROM pgr_tsp('{{0,1,2,3},{1,0,4,5},{2,4,0,6},{3,5,6,0}}'::float8[],1);

 seq | id
-----+----
   0 |  1
   1 |  2
   2 |  3
   3 |  0
(4 rows)
```

- Using distance matrix (Starting from 1, ending at 2)

```
SELECT seq, id FROM pgr_tsp('{{0,1,2,3},{1,0,4,5},{2,4,0,6},{3,5,6,0}}'::float8[],1,2);

 seq | id
-----+----
   0 |  1
   1 |  0
   2 |  3
   3 |  2
(4 rows)
```

- Using the vertices table edge_table_vertices_pgr generated by *pgr_createTopology*. Again we have two queries where the first might vary and the second is based on the overal path length.

```
    SELECT seq, id1, id2, round(cost::numeric, 2) AS cost
FROM pgr_tsp('SELECT id::integer, st_x(the_geom) as x,st_x(the_geom) as y FROM edge_table_vertice

 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   5 |   6 | 0.00
   1 |  10 |  11 | 0.00
   2 |   2 |   3 | 1.41
   3 |   3 |   4 | 0.00
   4 |  11 |  12 | 0.00
   5 |   8 |   9 | 0.71
   6 |  15 |  16 | 0.00
   7 |  16 |  17 | 2.12
   8 |   1 |   2 | 0.00
   9 |  14 |  15 | 1.41
  10 |   7 |   8 | 1.41
  11 |   6 |   7 | 0.71
  12 |  13 |  14 | 2.12
  13 |   0 |   1 | 0.00
  14 |   9 |  10 | 0.00
  15 |  12 |  13 | 0.00
  16 |   4 |   5 | 1.41
(17 rows)

SELECT round(sum(cost)::numeric, 4) as cost
   FROM pgr_tsp('SELECT id::integer, st_x(the_geom) as x,st_x(the_geom) as y FROM edge_table_vert

  cost
---------
 11.3137
(1 row)
```

The queries use the *Sample Data* network.

See Also

- *Traveling Sales Person*, *pgr_TSP*, *pgr_eucledianTSP*
- *pgr_costResult[]*
- http://en.wikipedia.org/wiki/Traveling_salesman_problem
- http://en.wikipedia.org/wiki/Simulated_annealing

pgr_flipEdges - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_flipEdges` - flip the edges in an array of geometries so the connect end to end.

Synopsis The function returns:

- `geometry[]` An array of the input geometries with the geometries flipped end to end such that the geometries are oriented as a path from start to end.

```
geometry[] pgr_flipEdges(ga geometry[])
```

Description Given an array of linestrings that are supposedly connected end to end like the results of a route, check the edges and flip any end for end if they do not connect with the previous seegment and return the array with the segments flipped as appropriate.

Parameters

ga `geometry[]` An array of geometries, like the results of a routing query.

Warning:

- No checking is done for edges that do not connect.
- Input geometries MUST be LINESTRING or MULTILINESTRING.
- Only the first LINESTRING of a MULTILINESTRING is considered.

History

- Deprecated in version 2.3.0
- Proposed in version 2.1.0

Examples

```
SELECT st_astext(e) FROM (SELECT unnest(pgr_flipedges(ARRAY[
'LINESTRING(2 1,2 2)'::geometry,
'LINESTRING(2 2,2 3)'::geometry,
'LINESTRING(2 2,2 3)'::geometry,
'LINESTRING(2 2,3 2)'::geometry,
'LINESTRING(3 2,4 2)'::geometry,
'LINESTRING(4 1,4 2)'::geometry,
'LINESTRING(3 1,4 1)'::geometry,
'LINESTRING(2 1,3 1)'::geometry,
'LINESTRING(2 0,2 1)'::geometry,
'LINESTRING(2 0,2 1)'::geometry]::geometry[])) AS e) AS foo;
NOTICE:  Deperecated function: pgr_flipEdges
      st_astext
---------------------
 LINESTRING(2 1,2 2)
 LINESTRING(2 2,2 3)
 LINESTRING(2 3,2 2)
 LINESTRING(2 2,3 2)
 LINESTRING(3 2,4 2)
 LINESTRING(4 2,4 1)
 LINESTRING(4 1,3 1)
 LINESTRING(3 1,2 1)
 LINESTRING(2 1,2 0)
 LINESTRING(2 0,2 1)
(10 rows)
```

See also

Indices and tables

- genindex

- search

pgr_vidsToDMatrix - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_vidsToDMatrix` - Creates a distances matrix from an array of `vertex_id`.

Synopsis This function takes an array of `vertex_id`, the original array of points used to generate the array of `vertex_id`, an edge table name and a tol. It then computes kdijkstra() distances for each vertex to all the other vertices and creates a symmetric distance matrix suitable for TSP. The pnt array and the tol are used to establish a BBOX for limiting selection of edges. The extents of the points is expanded by tol.

The function returns:

- **`record` - with two fields as describe here**
  - **dmatrix** `float8[]` - the distance matrix suitable to pass to pgrTSP() function.
  - **ids** `integer[]` - an array of ids for the distance matrix.

```
record pgr_vidsToDMatrix(IN vids integer[], IN pnts geometry[], IN edges text, tol float8 DEFAULT
```

Description

Parameters

vids `integer[]` - An array of `vertex_id`.

pnts `geometry[]` - An array of point geometries that approximates the extents of the `vertex_id`.

edges `text` - The edge table to be used for the conversion.

tol `float8` - The amount to expand the BBOX extents of `pnts` when building the graph.

Warning:
- we compute a symmetric matrix because TSP requires that so the distances are better the Euclidean but but are not perfect
- kdijkstra() can fail to find a path between some of the vertex ids. We to not detect this other than the cost might get set to -1.0, so the dmatrix should be checked for this as it makes it invalid for TSP

History

- Proposed in version 2.1.0

Examples This example uses existing data of points.

```
SELECT * FROM pgr_vidstodmatrix(
    ARRAY[1,2,3,5],
    ARRAY(select the_geom FROM edge_table_vertices_pgr WHERE id in (1,2,3,5)),
    'edge_table'
);
NOTICE:  Deprecated function pgr_vidsToDMatrix
                  dmatrix                  |    ids
-------------------------------------------+-----------
 {{0,1,4,2},{1,0,3,1},{4,3,0,2},{2,1,2,0}} | {1,2,3,5}
(1 row)
```

This example uses points that are not part of the graph.

- *pgr_textToPoints - Deprecated Function* - is used to convert the locations into point geometries.
- *pgr_pointsToVids - Proposed* - to convert the array of point geometries into vertex ids.

```
SELECT * FROM pgr_vidstodmatrix(
    pgr_pointstovids(pgr_texttopoints('2,0;2,1;3,1;2,2', 0), 'edge_table'),
    pgr_texttopoints('2,0;2,1;3,1;2,2', 0),
    'edge_table');
NOTICE:  Deperecated function: pgr_textToPoints
NOTICE:  Deperecated function: pgr_textToPoints
NOTICE:  Deprecated function pgr_vidsToDMatrix
                  dmatrix                  |    ids
-------------------------------------------+-----------
 {{0,1,4,2},{1,0,3,1},{4,3,0,2},{2,1,2,0}} | {1,2,3,5}
(1 row)
```

This example shows how this can be used in the context of feeding the results into pgr_tsp() function.

```
SELECT * FROM pgr_tsp(
    (SELECT dMatrix FROM pgr_vidstodmatrix(
        pgr_pointstovids(pgr_texttopoints('2,0;2,1;3,1;2,2', 0), 'edge_table'),
        pgr_texttopoints('2,0;2,1;3,1;2,2', 0),
        'edge_table')
    ),
    1
);
NOTICE:  Deperecated function: pgr_textToPoints
NOTICE:  Deperecated function: pgr_textToPoints
NOTICE:  Deprecated function pgr_vidsToDMatrix
 seq | id
-----+----
   0 |  1
   1 |  2
   2 |  3
   3 |  0
(4 rows)
```

This example uses the *Sample Data* network.

See Also

- *pgr_vidsToDMatrix - Deprecated Function*
- *pgr_textToPoints - Deprecated Function*
- *pgr_tsp -Deprecated Signatures*

Indices and tables

- genindex
- search

pgr_vidsToDMatrix - Deprecated Function

> **Warning:** This function is deprecated!!!
> - Is no longer supported.
> - May be removed from future versions.
> - There is no replacement.

Name `pgr_vidsToDMatrix` - Creates a distances matrix from an array of `vertex_id`.

Synopsis This function takes an array of `vertex_id`, a `sql` statement to select the edges, and some boolean arguments to control the behavior. It then computes kdijkstra() distances for each vertex to all the other vertices and creates a distance matrix suitable for TSP.

The function returns:

- **dmatrix** `float8[]` - the distance matrix suitable to pass to pgr_TSP() function.

```
pgr_vidsToDMatrix(IN sql text, IN vids integer[], IN directed boolean, IN has_reverse_cost boolean
```

Description

Parameters

sql `text` - A SQL statement to select the edges needed for the solution.

vids `integer[]` - An array of `vertex_id`.

directed `boolean` - A flag to indicate if the graph is directed.

has_reverse_cost `boolean` - A flag to indicate if the SQL has a column `reverse_cost`.

want_symmetric `boolean` - A flag to indicate if you want a symmetric or asymmetric matrix. You will need a symmetric matrix for pgr_TSP(). If the matriix is asymmetric, the then the cell(i,j) and cell(j,i) will be set to the average of those two cells except if one or the other are -1.0 then it will take the value of the other cell. If both are negative they will be left alone.

> **Warning:**
> - kdijkstra() can fail to find a path between some of the vertex ids. We to not detect this other than the cost might get set to -1.0, so the dmatrix should be checked for this as it makes it invalid for TSP

History

- Proposed in version 2.1.0

Examples

```
SELECT * FROM pgr_vidsToDMatrix(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    array[1,2,3,5],
    true, true, false);
NOTICE:  Deprecated function pgr_vidsToDMatrix
             pgr_vidstodmatrix
-------------------------------------------
 {{0,1,2,2},{1,0,1,1},{2,1,0,4},{2,1,4,0}}
(1 row)

SELECT * FROM pgr_vidsToDMatrix(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    array[1,2,3,5],
    true, true, true);
NOTICE:  Deprecated function pgr_vidsToDMatrix
             pgr_vidstodmatrix
-------------------------------------------
 {{0,1,2,2},{1,0,1,1},{2,1,0,2},{2,1,2,0}}
(1 row)
```

This example shows how this can be used in the context of feeding the results into pgr_tsp() function.

```
SELECT * FROM pgr_tsp(
    (SELECT pgr_vidsToDMatrix(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table
        array[1,2,3,5],
        true, true, true)
    ),
    1
);
NOTICE:  Deprecated function pgr_vidsToDMatrix
 seq | id
-----+----
   0 |  1
   1 |  2
   2 |  3
   3 |  0
(4 rows)
```

This example uses the *Sample Data* network.

See Also

- *pgr_vidsToDMatrix - Deprecated Function*
- *pgr_textToPoints - Deprecated Function*
- *pgr_tsp -Deprecated Signatures*

Indices and tables

- genindex
- search

pgr_pointsToDMatrix - Deprecated Function

Warning: This function is deprecated!!!

- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_pointsToDMatrix` - Creates a distance matrix from an array of points.

Synopsis Create a distance symmetric distance matrix suitable for TSP using Euclidean distances based on the st_distance(). You might want to create a variant of this the uses st_distance_sphere() or st_distance_spheriod() or some other function.

The function returns:

- **`record` - with two fields as describe here**
  - **dmatrix** `float8[]` - the distance matrix suitable to pass to pgrTSP() function.
  - **ids** `integer[]` - an array of ids for the distance matrix.

```
record pgr_pointsToDMatrix(pnts geometry[], OUT dmatrix double precision[], OUT ids integer[])
```

Description

Parameters

pnts `geometry[]` - An array of point geometries.

Warning: The generated matrix will be symmetric as required for pgr_TSP.

History

- Proposed in version 2.1.0

Examples

```
SELECT * FROM pgr_pointstodmatrix(pgr_texttopoints('2,0;2,1;3,1;2,2', 0));
NOTICE:  Deperecated function: pgr_textToPoints
NOTICE:  Deprecated function pgr_pointsToDMatrix
                                              dmatrix
---------------------------------------------------------------------------------------------------
 {{0,1,1.4142135623731,2},{1,0,1,1},{1.4142135623731,1,0,1.4142135623731},{2,1,1.4142135623731,0}
(1 row)
```

This example shows how this can be used in the context of feeding the results into pgr_tsp() function.

```
SELECT * from pgr_tsp(
    (SELECT dMatrix FROM pgr_pointstodmatrix(pgr_texttopoints('2,0;2,1;3,1;2,2', 0))
    ),
    1
);
NOTICE:  Deperecated function: pgr_textToPoints
NOTICE:  Deprecated function pgr_pointsToDMatrix
 seq | id
```

```
-----+----
   0 |   1
   1 |   3
   2 |   2
   3 |   0
(4 rows)
```

See Also

- *pgr_vidsToDMatrix - Deprecated Function*
- *pgr_vidsToDMatrix - Deprecated Function*
- *pgr_tsp -Deprecated Signatures*

Indices and tables

- genindex
- search

pgr_textToPoints - Deprecated Function

> **Warning:** This function is deprecated!!!
> - Is no longer supported.
> - May be removed from future versions.
> - There is no replacement.

Name `pgr_textToPoints` - Converts a text string of the format "x,y;x,y;x,y;..." into and array of point geometries.

Synopsis Given a text string of the format "x,y;x,y;x,y;..." and the srid to use, split the string and create and array point geometries.

The function returns:

-

```
integer pgr_textToPoints(pnts text, srid integer DEFAULT(4326))
```

Description

Parameters

pnts `text` A text string of the format "x,y;x,y;x,y;..." where x is longitude and y is latitude if use values in lat-lon.

srid `integer` The SRID to use when constructing the point geometry. If the paratmeter is absent it defaults to `SRID:4326`.

History

- Proposed in version 2.1.0

Examples

```
SELECT ST_AsText(g) FROM
    (SELECT unnest(pgr_texttopoints('2,0;2,1;3,1;2,2', 0)) AS g) AS foo;
NOTICE:  Deperecated function: pgr_textToPoints
 st_astext
------------
 POINT(2 0)
 POINT(2 1)
 POINT(3 1)
 POINT(2 2)
(4 rows)
```

See Also

- *pgr_pointToEdgeNode - Proposed*
- *pgr_pointsToVids - Proposed*

Indices and tables

- genindex
- search

8.2.2 Deprecated on version 2.2

Routing functions

- *pgr_apspJohnson - Deprecated function* - Replaced with *pgr_johnson*
- *pgr_apspWarshall - Deprecated Function* - Replaced with *pgr_floydWarshall*
- *pgr_kDijkstra - Deprecated Functions* - Replaced with *pgr_dijkstraCost* and *pgr_dijkstra* (one to many)

pgr_apspJohnson - Deprecated function

Warning: This function is deprecated!!!

- It has been replaced by a new functions, is no longer supported, and may be removed from future versions.
- All code that uses this function should be converted to use its replacement: *pgr_johnson*.

Name `pgr_apspJohnson` - Returns all costs for each pair of nodes in the graph.

Synopsis Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse, edge weighted, directed graph. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows for every pair of nodes in the graph.

```
pgr_costResult[] pgr_apspJohnson(sql text);
```

Description

sql a SQL query that should return the edges for the graph that will be analyzed:

```
SELECT source, target, cost FROM edge_table;
```

source `int4` identifier of the source vertex for this edge

target `int4` identifier of the target vertex for this edge

cost `float8` a positive value for the cost to traverse this edge

Returns set of *pgr_costResult[]*:

seq row sequence

id1 source node ID

id2 target node ID

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

History

- Deprecated in version 2.2.0
- New in version 2.0.0

Examples

```
SELECT * FROM pgr_apspJohnson(
        'SELECT source::INTEGER, target::INTEGER, cost FROM edge_table WHERE id < 5'
    );
NOTICE:  Deprecated function: Use pgr_johnson instead
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   1 |   2 |    1
   1 |   1 |   5 |    2
   2 |   2 |   5 |    1
(3 rows)
```

The query uses the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *pgr_johnson*
- http://en.wikipedia.org/wiki/Johnson%27s_algorithm

pgr_apspWarshall - Deprecated Function

Warning: This function is deprecated!!!

- It has been replaced by a new function, is no longer supported, and may be removed from future versions.
- All code that uses this function should be converted to use its replacement: *pgr_floydWarshall*.

Name `pgr_apspWarshall` - Returns all costs for each pair of nodes in the graph.

Synopsis The Floyd-Warshall algorithm (also known as Floyd's algorithm and other names) is a graph analysis algorithm for finding the shortest paths between all pairs of nodes in a weighted graph. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows for every pair of nodes in the graph.

```
pgr_costResult[] pgr_apspWarshall(sql text, directed boolean, reverse_cost boolean);
```

Description

sql a SQL query that should return the edges for the graph that will be analyzed:

```
        SELECT id, source, target, cost FROM edge_table;
```

id `int4` identifier of the edge

source `int4` identifier of the source vertex for this edge

target `int4` identifier of the target vertex for this edge

cost `float8` a positive value for the cost to traverse this edge

reverse_cost `float8` (optional) a positive value for the reverse cost to traverse this edge

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.

Returns set of *pgr_costResult[]*:

seq row sequence

id1 source node ID

id2 target node ID

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

History

- Deprecated in version 2.0.0
- New in version 2.0.0

Examples

```
SELECT * FROM pgr_apspWarshall(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table WHERE id < 5',
        false, false
    );
NOTICE:  Deprecated function: Use pgr_floydWarshall instead
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   1 |   2 |    1
   1 |   1 |   5 |    2
   2 |   2 |   1 |    1
   3 |   2 |   5 |    1
   4 |   5 |   1 |    2
   5 |   5 |   2 |    1
(6 rows)
```

The query uses the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *pgr_floydWarshall*
- http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

pgr_kDijkstra - Deprecated Functions

> **Warning:** These functions are deprecated!!!
> - It has been replaced by a new functions, are no longer supported, and may be removed from future versions.
> - All code that uses the functions should be converted to use its replacement.

Name

- `pgr_kdijkstraCost` - Returns the costs for K shortest paths using Dijkstra algorithm.

> **Warning:** Use *pgr_dijkstraCost* (One To Many) instead.

- `pgr_kdijkstraPath` - Returns the paths for K shortest paths using Dijkstra algorithm.

> **Warning:** Use *pgr_dijkstra* (One To Many) instead.

Synopsis These functions allow you to have a single start node and multiple destination nodes and will compute the routes to all the destinations from the source node. Returns a set of *pgr_costResult* or *pgr_costResult3*. `pgr_kdijkstraCost` returns one record for each destination node and the cost is the total code of the route to that node. `pgr_kdijkstraPath` returns one record for every edge in that path from source to destination and the cost is to traverse that edge.

```
pgr_costResult[] pgr_kdijkstraCost(text sql, integer source,
                 integer[] targets, boolean directed, boolean has_rcost);

pgr_costResult3[] pgr_kdijkstraPath(text sql, integer source,
                  integer[] targets, boolean directed, boolean has_rcost);
```

Description

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` id of the start point

targets `int4[]` an array of ids of the end points

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.

`pgr_kdijkstraCost` returns set of *pgr_costResult[]*:

seq row sequence

id1 path vertex source id (this will always be source start point in the query).

id2 path vertex target id

cost cost to traverse the path from `id1` to `id2`. Cost will be -1.0 if there is no path to that target vertex id.

`pgr_kdijkstraPath` returns set of *pgr_costResult3[] - Multiple Path Results with Cost*:

seq row sequence

id1 path target id (identifies the target path).

id2 path edge source node id

id3 path edge id (`-1` for the last row)

cost cost to traverse this edge or -1.0 if there is no path to this target

History

- Deprecated in version 2.0.0
- New in version 2.0.0

Examples

- Returning a `cost` result

```
SELECT * FROM pgr_kdijkstraCost(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
    10, array[4,12], false, false);
NOTICE:  Deprecated function. Use pgr_dijkstraCost instead.
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |  10 |   4 |    4
   1 |  10 |  12 |    2
(2 rows)
```

```
SELECT * FROM pgr_kdijkstraPath(
      'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
      10, array[4,12], false, false);
NOTICE:  Deprecated function: Use pgr_dijkstra instead.
 seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+------
   0 |   4 |  10 |  12 |    1
   1 |   4 |  11 |  13 |    1
   2 |   4 |  12 |  15 |    1
   3 |   4 |   9 |  16 |    1
   4 |   4 |   4 |  -1 |    0
   5 |  12 |  10 |  12 |    1
   6 |  12 |  11 |  13 |    1
   7 |  12 |  12 |  -1 |    0
(8 rows)
```

- Returning a `path` result

```
SELECT id1 AS path, st_AStext(st_linemerge(st_union(b.the_geom))) AS the_geom
  FROM pgr_kdijkstraPath(
                  'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
                  10, array[4,12], false, false
            ) a,
            edge_table b
WHERE a.id3=b.id
GROUP by id1
ORDER by id1;
NOTICE:  Deprecated function: Use pgr_dijkstra instead.
 path |              the_geom
------+---------------------------------
    4 | LINESTRING(2 3,3 3,4 3,4 2,4 1)
   12 | LINESTRING(2 3,3 3,4 3)
(2 rows)
```

There is no assurance that the result above will be ordered in the direction of flow of the route, ie: it might be reversed. You will need to check if `st_startPoint()` of the route is the same as the start node location and if it is not then call `st_reverse()` to reverse the direction of the route. This behavior is a function of PostGIS functions `st_linemerge()` and `st_union()` and not pgRouting.

See Also

- *pgr_dijkstraCost*, *pgr_dijkstra*
- *pgr_costResult[]*
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

8.2.3 Deprecated on version 2.1

Routing functions

- *pgr_dijkstra - Deprecated Signature* - See new signature in *pgr_dijkstra* (one to one)
- *pgr_ksp - Deprecated Signature* - See new signature in *pgr_ksp*
- *pgr_drivingDistance - Deprecated Signature* - See new signature in *pgr_drivingDistance*

Auxiliary functions

- *pgr_getColumnName - Deprecated Function*
- *pgr_getTableName - Deprecated Function*
- *pgr_isColumnIndexed - Deprecated Function*
- *pgr_isColumnInTable - Deprecated Function*
- *pgr_quote_ident - Deprecated Function*
- *pgr_versionless - Deprecated Function*
- *pgr_startPoint - Deprecated Function*
- *pgr_endPoint - Deprecated Function*

pgr_dijkstra - Deprecated Signature

Warning: This function signature is deprecated!!!

- That means it has been replaced by new signature(s)
- This signature is no longer supported, and may be removed from future versions.
- All code that use this function signature should be converted to use its replacement *pgr_dijkstra* (One to One).

Name `pgr_dijkstra` — Returns the shortest path using Dijkstra algorithm.

Synopsis Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956. It is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows, that make up a path.

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

Description

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 `float8` (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` id of the start point

target `int4` 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.

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

- Renamed in version 2.0.0

Examples: Directed

- Without `reverse_cost`

```
SELECT * FROM pgr_dijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    2,3, true, false);
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
(0 rows)
```

- With `reverse_cost`

```
SELECT * FROM pgr_dijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    2,3, true, true);
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   4 |    1
   1 |   5 |   8 |    1
   2 |   6 |   9 |    1
   3 |   9 |  16 |    1
   4 |   4 |   3 |    1
   5 |   3 |  -1 |    0
(6 rows)
```

Examples: Undirected

- Without `reverse_cost`

```
SELECT * FROM pgr_dijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
     2, 3, false, false);
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   4 |    1
   1 |   5 |   8 |    1
   2 |   6 |   5 |    1
   3 |   3 |  -1 |    0
(4 rows)
```

- With `reverse_cost`

```
SELECT * FROM pgr_dijkstra(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
     2, 3, false, true);
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   2 |   2 |    1
   1 |   3 |  -1 |    0
(2 rows)
```

The queries use the *Sample Data* network.

See Also

- *Dijkstra - Family of functions*, *pgr_dijkstra*
- *pgr_costResult[]*
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

pgr_ksp - Deprecated Signature

> **Warning:** This function signature is deprecated!!!
> - That means it has been replaced by new signature(s)
> - This signature is no longer supported, and may be removed from future versions.
> - All code that use this function signature should be converted to use its replacement *pgr_ksp*.

Name `pgr_ksp` — Returns the "K" shortest paths.

Synopsis The K shortest path routing algorithm based on Yen's algorithm. "K" is the number of shortest paths desired. Returns a set of *pgr_costResult3* (seq, id1, id2, id3, cost) rows, that make up a path.

```
pgr_costResult3[] pgr_ksp(sql text, source integer, target integer,
                          paths integer, has_rcost boolean);
```

Description

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 `has_rcost` the parameter is `true` (see the above remark about negative costs).

source `int4` id of the start point

target `int4` id of the end point

paths `int4` number of alternative routes

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.

Returns set of *pgr_costResult[]*:

seq sequence for ording the results

id1 route ID

id2 node ID

id3 edge ID (`0` for the last row)

cost cost to traverse from `id2` using `id3`

KSP code base taken from http://code.google.com/p/k-shortest-paths/source.

History

- New in version 2.0.0

Examples

- Without `reverse_cost`

```
SELECT * FROM pgr_ksp(
   'SELECT id, source, target, cost FROM edge_table order by id',
   7, 12, 2, false
 );
NOTICE:  Deprecated function
 seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+------
   0 |   0 |   7 |   6 |    1
   1 |   0 |   8 |   7 |    1
   2 |   0 |   5 |   8 |    1
   3 |   0 |   6 |   9 |    1
   4 |   0 |   9 |  15 |    1
   5 |   0 |  12 |  -1 |    0
   6 |   1 |   7 |   6 |    1
   7 |   1 |   8 |   7 |    1
   8 |   1 |   5 |   8 |    1
   9 |   1 |   6 |  11 |    1
  10 |   1 |  11 |  13 |    1
  11 |   1 |  12 |  -1 |    0
(12 rows)
```

- With `reverse_cost`

```
SELECT * FROM pgr_ksp(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table order by id',
   7, 12, 2, true
 );
NOTICE:  Deprecated function
 seq | id1 | id2 | id3 | cost
-----+-----+-----+-----+------
   0 |   0 |   7 |   6 |    1
   1 |   0 |   8 |   7 |    1
   2 |   0 |   5 |   8 |    1
   3 |   0 |   6 |   9 |    1
   4 |   0 |   9 |  15 |    1
   5 |   0 |  12 |  -1 |    0
   6 |   1 |   7 |   6 |    1
   7 |   1 |   8 |   7 |    1
   8 |   1 |   5 |   8 |    1
   9 |   1 |   6 |  11 |    1
  10 |   1 |  11 |  13 |    1
  11 |   1 |  12 |  -1 |    0
(12 rows)
```

The queries use the *Sample Data* network.

See Also

- *pgr_ksp*
- *pgr_costResult3[] - Multiple Path Results with Cost*
- http://en.wikipedia.org/wiki/K_shortest_path_routing

pgr_drivingDistance - Deprecated Signature

Warning: This function signature is deprecated!!!

- That means it has been replaced by new signature(s)
- This signature is no longer supported, and may be removed from future versions.
- All code that use this function signature should be converted to use its replacement *pgr_drivingDistance*.

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

Synopsis This function computes a Dijkstra shortest path solution them extracts the cost to get to each node in the network from the starting node. Using these nodes and costs it is possible to compute constant drive time polygons. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows, that make up a list of accessible points.

```
pgr_costResult[] pgr_drivingDistance(text sql, integer source, double precision distance,
                                    boolean directed, boolean has_rcost);
```

Description

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` id of the start point

distance `float8` value in edge cost units (not in projection units - they might be different).

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.

Returns set of *pgr_costResult[]*:

seq row sequence

id1 node ID

id2 edge ID (this is probably not a useful item)

cost cost to get to this node ID

Warning: You must reconnect to the database after `CREATE EXTENSION pgrouting`. Otherwise the function will return `Error computing path: std::bad_alloc`.

History

- Renamed in version 2.0.0

Examples

- Without `reverse_cost`
- With `reverse_cost`

```
SELECT * FROM pgr_drivingDistance(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM edge_table',
    7, 1.5, false, false
) ;
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   7 |  -1 |    0
   1 |   8 |   6 |    1
(2 rows)

SELECT * FROM pgr_drivingDistance(
    'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost, reverse_cost FROM edge_table',
    7, 1.5, true, true
) ;
NOTICE:  Deprecated function
 seq | id1 | id2 | cost
-----+-----+-----+------
   0 |   7 |  -1 |    0
   1 |   8 |   6 |    1
(2 rows)
```

The queries use the *Sample Data* network.

See Also

- *pgr_drivingDistance*
- *pgr_alphaShape* - Alpha shape computation
- *pgr_pointsAsPolygon* - Polygon around set of points

pgr_getColumnName - Deprecated Function

Warning: This function is deprecated!!!

- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_getColumnName` — Retrieves the name of the column as is stored in the postgres administration tables.

Note: This function is intended for the developer's aid.

Synopsis Returns a `text` contining the registered name of the column.

```
text pgr_getColumnName(tab text, col text);
```

Description Parameters

tab `text` table name with or without schema component.

col `text` column name to be retrieved.

Returns

- `text` containing the registered name of the column.
- `NULL` when :
  - The table "tab" is not found or
  - Column "col" is not found in table "tab" in the postgres administration tables.

History

- New in version 2.0.0

Examples

```
SELECT pgr_getColumnName('edge_table','the_geom');

 pgr_iscolumnintable
---------------------
 the_geom
(1 row)

SELECT pgr_getColumnName('edge_table','The_Geom');

 pgr_iscolumnintable
---------------------
 the_geom
(1 row)
```

The queries use the *Sample Data* network.

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_isColumnInTable - Deprecated Function* to check only for the existence of the column.
- *pgr_getTableName - Deprecated Function* to retrieve the name of the table as is stored in the postgres administration tables.

pgr_getTableName - Deprecated Function

Warning: This function is deprecated!!!

- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_getTableName` — Retrieves the name of the column as is stored in the postgres administration tables.

Note: This function is intended for the developer's aid.

Synopsis Returns a record containing the registered names of the table and of the schema it belongs to.

```
(text sname, text tname)  pgr_getTableName(text tab)
```

Description Parameters

tab `text` table name with or without schema component.

Returns

sname

- `text` containing the registered name of the schema of table "tab".
  - when the schema was not provided in "tab" the current schema is used.
- `NULL` when :
  - The schema is not found in the postgres administration tables.

tname

- `text` containing the registered name of the table "tab".
- `NULL` when :
  - The schema is not found in the postgres administration tables.
  - The table "tab" is not registered under the schema `sname` in the postgres administration tables

History

- New in version 2.0.0

Examples

```
SELECT * from pgr_getTableName('edge_table');

 sname  |   tname
--------+------------
 public | edge_table
(1 row)

SELECT * from pgr_getTableName('EdgeTable');

 sname  |   tname
--------+------------
 public |
(1 row)

SELECT * from pgr_getTableName('data.Edge_Table');
 sname | tname
-------+-------
       |
(1 row)
```

The examples use the *Sample Data* network.

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_isColumnInTable - Deprecated Function* to check only for the existence of the column.

- *pgr_getTableName - Deprecated Function* to retrieve the name of the table as is stored in the postgres administration tables.

pgr_isColumnIndexed - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_isColumnIndexed` — Check if a column in a table is indexed.

Note: This function is intended for the developer's aid.

Synopsis Returns `true` when the column "col" in table "tab" is indexed.

```
boolean pgr_isColumnIndexed(text tab, text col);
```

Description

tab `text` Table name with or without schema component.

col `text` Column name to be checked for.

Returns:

- `true` when the column "col" in table "tab" is indexed.
- `false` when:
- The table "tab" is not found or
- Column "col" is not found in table "tab" or
- Column "col" in table "tab" is not indexed

History

- New in version 2.0.0

Examples

```
SELECT pgr_isColumnIndexed('edge_table','x1');

 pgr_iscolumnindexed
---------------------
 f
(1 row)

SELECT pgr_isColumnIndexed('public.edge_table','cost');

 pgr_iscolumnindexed
---------------------
 f
(1 row)
```

The example use the *Sample Data* network.

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_isColumnInTable - Deprecated Function* to check only for the existence of the column in the table.
- *pgr_getColumnName - Deprecated Function* to get the name of the column as is stored in the postgres administration tables.
- *pgr_getTableName - Deprecated Function* to get the name of the table as is stored in the postgres administration tables.

pgr_isColumnInTable - Deprecated Function

Warning: This function is deprecated!!!

- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_isColumnInTable` — Check if a column is in the table.

Note: This function is intended for the developer's aid.

Synopsis Returns `true` when the column "col" is in table "tab".

```
boolean pgr_isColumnInTable(text tab, text col);
```

Description

tab `text` Table name with or without schema component.

col `text` Column name to be checked for.

Returns:

- `true` when the column "col" is in table "tab".
- `false` when:
- The table "tab" is not found or
- Column "col" is not found in table "tab"

History

- New in version 2.0.0

Examples

```
SELECT pgr_isColumnInTable('edge_table','x1');

 pgr_iscolumnintable
---------------------
 t
(1 row)

SELECT pgr_isColumnInTable('public.edge_table','foo');

 pgr_iscolumnintable
```

```
---------------------
 f
(1 row)
```

The example use the *Sample Data* network.

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_isColumnIndexed - Deprecated Function* to check if the column is indexed.
- *pgr_getColumnName - Deprecated Function* to get the name of the column as is stored in the postgres administration tables.
- *pgr_getTableName - Deprecated Function* to get the name of the table as is stored in the postgres administration tables.

pgr_pointToId - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_pointToId` — Inserts a point into a vertices table and returns the corresponig id.

Note: This function is intended for the developer's aid.

Synopsis This function returns the `id` of the row in the vertices table that corresponds to the `point` geometry

```
bigint pgr_pointToId(geometry point, double precision tolerance,text vertname text,integer srid)
```

Description

point `geometry` "POINT" geometry to be inserted.

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

vertname `text` Vertices table name WITH schema included.

srid `integer` SRID of the geometry point.

This function returns the id of the row that corresponds to the `point` geometry

- When the `point` geometry already exists in the vertices table `vertname`, it returns the corresponding `id`.
- When the `point` geometry is not found in the vertices table `vertname`, the function inserts the `point` and returns the corresponding `id` of the newly created vertex.

Warning: The function do not perform any checking of the parameters. Any validation has to be done before calling this function.

History

- Renamed in version 2.0.0

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_createVerticesTable* to create a topology based on the geometry.
- *pgr_createTopology* to create a topology based on the geometry.

pgr_quote_ident - Deprecated Function

Warning: This function is deprecated!!!

- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_quote_ident` — Quotes the input text to be used as an identifier in an SQL statement string.

Note: This function is intended for the developer's aid.

Synopsis Returns the given identifier `idname` suitably quoted to be used as an identifier in an SQL statement string.

```
text pgr_quote_ident(text idname);
```

Description

Parameters

idname `text` Name of an SQL identifier. Can include `.` dot notation for schemas.table identifiers

Returns the given string suitably quoted to be used as an identifier in an SQL statement string.

- When the identifier `idname` contains on or more `.` separators, each component is suitably quoted to be used in an SQL string.

History

- New in version 2.0.0

Examples Everything is lower case so nothing needs to be quoted.

```
SELECT pgr_quote_ident('the_geom');

pgr_quote_ident
---------------
   the_geom
(1 row)


SELECT pgr_quote_ident('public.edge_table');

  pgr_quote_ident
-------------------
 public.edge_table
(1 row)
```

The column is upper case so its double quoted.

```
SELECT pgr_quote_ident('edge_table.MYGEOM');

   pgr_quote_ident
-------------------
 edge_table."MYGEOM"
(1 row)

SELECT pgr_quote_ident('public.edge_table.MYGEOM');

       pgr_quote_ident
---------------------------
 public.edge_table."MYGEOM"
(1 row)
```

The schema name has a capital letter so its double quoted.

```
SELECT pgr_quote_ident('Myschema.edge_table');

    pgr_quote_ident
----------------------
 "Myschema".edge_table
(1 row)
```

Ignores extra `.` separators.

```
SELECT pgr_quote_ident('Myschema...edge_table');

   pgr_quote_ident
---------------------
 "Myschema".edge_table
(1 row)
```

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_getTableName - Deprecated Function* to get the name of the table as is stored in the postgres administration tables.

pgr_versionless - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_versionless` — Compare two version numbers.

Note: This function is intended for the developer's aid.

Synopsis Returns `true` if the first version number is smaller than the second version number. Otherwise returns `false`.

```
boolean pgr_versionless(text v1, text v2);
```

Description

v1 `text` first version number

v2 `text` second version number

History

- New in version 2.0.0

Examples

```
SELECT pgr_versionless('2.0.1', '2.1');

 pgr_versionless
-----------------
 t
(1 row)
```

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_version* to get the current version of pgRouting.

pgr_startPoint - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_startPoint` — Returns a start point of a (multi)linestring geometry.

Note: This function is intended for the developer's aid.

Synopsis Returns the geometry of the start point of the first LINESTRING of `geom`.

```
geometry pgr_startPoint(geometry geom);
```

Description

Parameters

geom `geometry` Geometry of a MULTILINESTRING or LINESTRING.

Returns the geometry of the start point of the first LINESTRING of `geom`.

History

- New in version 2.0.0

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_endPoint - Deprecated Function* to get the end point of a (multi)linestring.

pgr_endPoint - Deprecated Function

Warning: This function is deprecated!!!
- Is no longer supported.
- May be removed from future versions.
- There is no replacement.

Name `pgr_endPoint` — Returns an end point of a (multi)linestring geometry.

Note: This function is intended for the developer's aid.

Synopsis Returns the geometry of the end point of the first LINESTRING of `geom`.

```
text pgr_startPoint(geometry geom);
```

Description

Parameters

geom `geometry` Geometry of a MULTILINESTRING or LINESTRING.

Returns the geometry of the end point of the first LINESTRING of `geom`.

History

- New in version 2.0.0

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_startPoint - Deprecated Function* to get the start point of a (multi)linestring.

CHAPTER 9

Change Log

Release Notes

9.1 Release Notes

To see the full list of changes check the list of Git commits[1] on Github.

9.1.1 Table of contents

[1] https://github.com/pgRouting/pgrouting/commits

9.2 pgRouting 2.3.2 Release Notes

To see the issues closed by this release see the Git closed issues for 2.3.2[2] on Github.

Bug Fixes

- Fixed pgr_gsoc_vrppdtw crash when all orders fit on one truck.
- Fixed pgr_trsp:
  - Alternate code is not executed when the point is in reality a vertex
  - Fixed ambiguity on seq

9.3 pgRouting 2.3.1 Release Notes

To see the issues closed by this release see the Git closed issues for 2.3.1[3] on Github.

Bug Fixes

- Leaks on proposed max_flow functions
- Regression error on pgr_trsp
- Types discrepancy on pgr_createVerticesTable

9.4 pgRouting 2.3.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.3.0[4] on Github.

New Signatures

Indentifiers can be *ANY-INTEGER* and costs can be *ANY-NUMERICAL*

- pgr_TSP
- pgr_aStar

New Functions

- pgr_eucledianTSP

[2] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.3.2%22+is%3Aclosed
[3] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.3.1%22+is%3Aclosed
[4] https://github.com/pgRouting/pgrouting/issues?q=is%3Aissue+milestone%3A%22Release+2.3.0%22+is%3Aclosed

New Proposed functions

- pgr_dijkstraCostMatrix
- pgr_withPointsCostMatrix
- pgr_maxFlowPushRelabel
- pgr_maxFlowEdmondsKarp
- pgr_maxFlowBoykovKolmogorov
- pgr_maximumCardinalityMatching
- pgr_edgeDisjointPaths
- pgr_contractGraph

Deprecated Signatures

- pgr_tsp - use pgr_TSP or pgr_eucledianTSP instead
- pgr_astar - use pgr_aStar instead

Deprecated Functions

- pgr_flip_edges
- pgr_vidsToDmatrix
- pgr_pointsToDMatrix
- pgr_textToPoints

9.5 pgRouting 2.2.4 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.4[5] on Github.

Bug Fixes

- Bogus uses of extern "C"
- Build error on Fedora 24 + GCC 6.0
- Regression error pgr_nodeNetwork

9.6 pgRouting 2.2.3 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.3[6] on Github.

Bug Fixes

- Fixed compatibility issues with PostgreSQL 9.6.

[5] https://github.com/pgRouting/pgrouting/issues?q=is%3Aissue+milestone%3A%22Release+2.2.4%22+is%3Aclosed
[6] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.2.3%22+is%3Aclosed

9.7 pgRouting 2.2.2 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.2[7] on Github.

Bug Fixes

- Fixed regression error on pgr_drivingDistance

9.8 pgRouting 2.2.1 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.1[8] on Github.

Bug Fixes

- Server crash fix on pgr_alphaShape
- Bug fix on With Points family of functions

9.9 pgRouting 2.2.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.2.0[9] 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

Proposed functionality

- pgr_withPoints
- pgr_withPointsCost
- pgr_withPointsDD
- pgr_withPointsKSP
- pgr_dijkstraVia

[7] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.2.2%22+is%3Aclosed
[8] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A2.2.1+is%3Aclosed
[9] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=is%3Aissue+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

9.10 pgRouting 2.1.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.1.0[10] on Github.

Refactored

- pgr_dijkstra
- pgr_ksp
- pgr_drivingDistance

Improvements

- Alphashape function now can generate better (multi)polygon with holes and alpha parameter.

Proposed functionality

- Proposed functions from Steve Woodbridge, (Classified as Convenience by the author.)
  - pgr_pointToEdgeNode - convert a point geometry to a vertex_id based on closest edge.
  - pgr_flipEdges - flip the edges in an array of geometries so the connect end to end.
  - pgr_textToPoints - convert a string of x,y;x,y;... locations into point geometries.
  - pgr_pointsToVids - convert an array of point geometries into vertex ids.
  - pgr_pointsToDMatrix - Create a distance matrix from an array of points.
  - pgr_vidsToDMatrix - Create a distance matrix from an array of vertix_id.
  - pgr_vidsToDMatrix - Create a distance matrix from an array of vertix_id.
- Added proposed functions from GSoc Projects:
  - pgr_vrppdtw

No longer supported

- Removed the 1.x legacy functions

Bug Fixes

- Some bug fixes in other functions

[10] https://github.com/pgRouting/pgrouting/issues?q=is%3Aissue+milestone%3A%22Release+2.1.0%22+is%3Aclosed

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 a the prefix 2-1-0

9.11 pgRouting 2.0.1 Release Notes

Minor bug fixes.

Bug Fixes

- No track of the bug fixes were kept.

9.12 pgRouting 2.0.0 Release Notes

To see the issues closed by this release see the Git closed issues for 2.0.0[11] 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.
- Standardiziation of the function naming
- Prepararation 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)

[11] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+2.0.0%22+is%3Aclosed

- 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 refactored 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

9.13 pgRouting 1.x Release Notes

To see the issues closed by this release see the Git closed issues for 1.x[12] on Github. The following release notes have been copied from the previous `RELEASE_NOTES` file and are kept as a reference.

9.13.1 Changes for release 1.05

- Bugfixes

9.13.2 Changes for release 1.03

- Much faster topology creation
- Bugfixes

9.13.3 Changes for release 1.02

- Shooting* bugfixes
- Compilation problems solved

9.13.4 Changes for release 1.01

- Shooting* bugfixes

[12] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A%22Release+1.x%22+is%3Aclosed

9.13.5 Changes for release 1.0

- Core and extra functions are separated
- Cmake build process
- Bugfixes

9.13.6 Changes for release 1.0.0b

- Additional SQL file with more simple names for wrapper functions
- Bugfixes

9.13.7 Changes for release 1.0.0a

- Shooting* shortest path algorithm for real road networks
- Several SQL bugs were fixed

9.13.8 Changes for release 0.9.9

- PostgreSQL 8.2 support
- Shortest path functions return empty result if they couldn't find any path

9.13.9 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