pgRouting Manual

Release 2.2.4 (pgrouting-2.2.4)

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 2.2.4 (pgrouting-2.2.4).

[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)

Daniel Kastl, Ko Nagase, Mario Basa, Regina Obe, 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, 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, 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.

Additional notes can be found in Installation Notes[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 PostgreSQL 9.2-9.5 32-bit/64-bit[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/

Ubuntu/Debian

Ubuntu packages are available in postgreSQL repositories.

Using a terminal window:

[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

```
# 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" > /et

# 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
```

This will also install all required packages such as postgreSQL and postGIS if not installed yet.

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
```

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.

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

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

```
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.

Installing the tests dependencies:

```
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.

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.

[23] http://pgrouting.org/support.html
[24] http://docs.pgrouting.org
[25] https://github.com/pgrouting/pgrouting/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	http://georepublic.info
iMaptools	United States	http://imaptools.com
Paaragon Corporation	United States	http://www.paragoncorporation.com/
Orkney Inc.	Japan	http://www.orkney.co.jp
Camptocamp	Switzerland, France	http://www.camptocamp.com

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

CHAPTER 2

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, <additonal 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 infomation 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*

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 indentified 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.

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 decribed 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 begining of a route.

route_id `INTEGER` id of the route.

path_seq `INTEGER` relative position in the path. Has value **1** for the begining 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 recomended 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')
```

2.1.5 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.

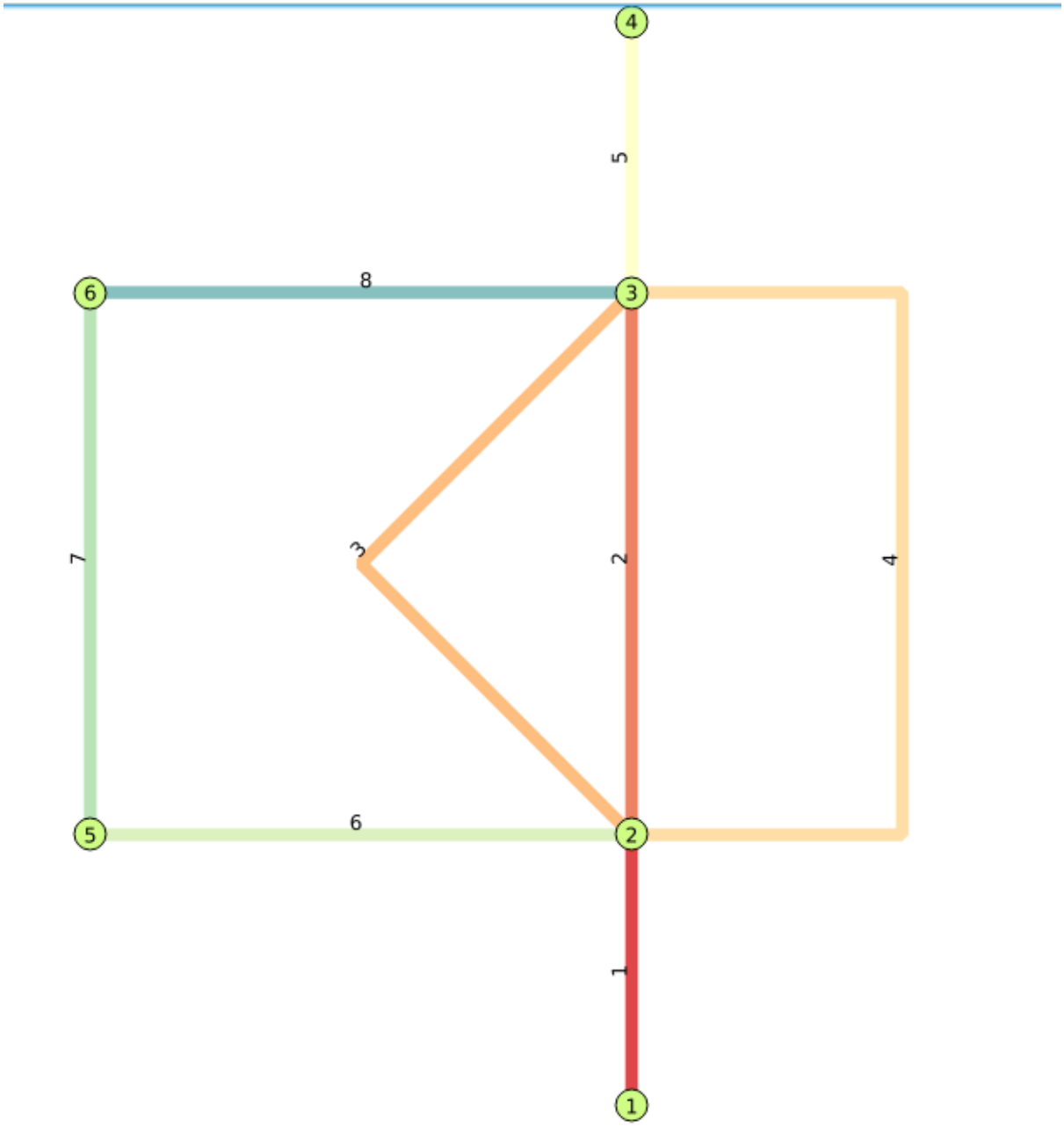
```
SELECT pgr_createTopology('edge_table', 0.001);
SELECT pgr_analyzegraph('edge_table', 0.001);
SELECT pgr_nodeNetwork('edge_table', 0.001);
SELECT pgr_createTopology('edge_table_noded', 0.001);
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
```

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

Warning: *Developers's Functions* documentation is going to be deleted from the pgRouting documentation and included in the doxygen documentation.

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 dependant 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` Configuraton file.
  - `<name>.test.sql` Test file
  - `<name>.result` Results file
- `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,
    x1 FLOAT,
    y1 FLOAT,
    x2 FLOAT,
    y2 FLOAT,
    the_geom geometry
);
```

Insert data

```
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,0,   2,1);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES (-1, 1,  2,1,   3,1);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES (-1, 1,  3,1,   4,1);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,1,   2,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  3,1,   3,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  0,2,   1,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  1,2,   2,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,2,   3,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  3,2,   4,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,2,   2,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  3,2,   3,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  2,3,   3,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  3,3,   4,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,3,   2,4);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  4,2,   4,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  4,1,   4,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  0.5,3.5,  1.999999999999,3
```

```
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 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
```

Before you test a routing function use this query to fill the source and target columns.

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

Points of interest

When points outside of the graph

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

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

COPY restrictions (rid, to_cost, target_id, from_edge, via_path) FROM stdin WITH NULL '__NULL__' D
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__
\.
```

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.

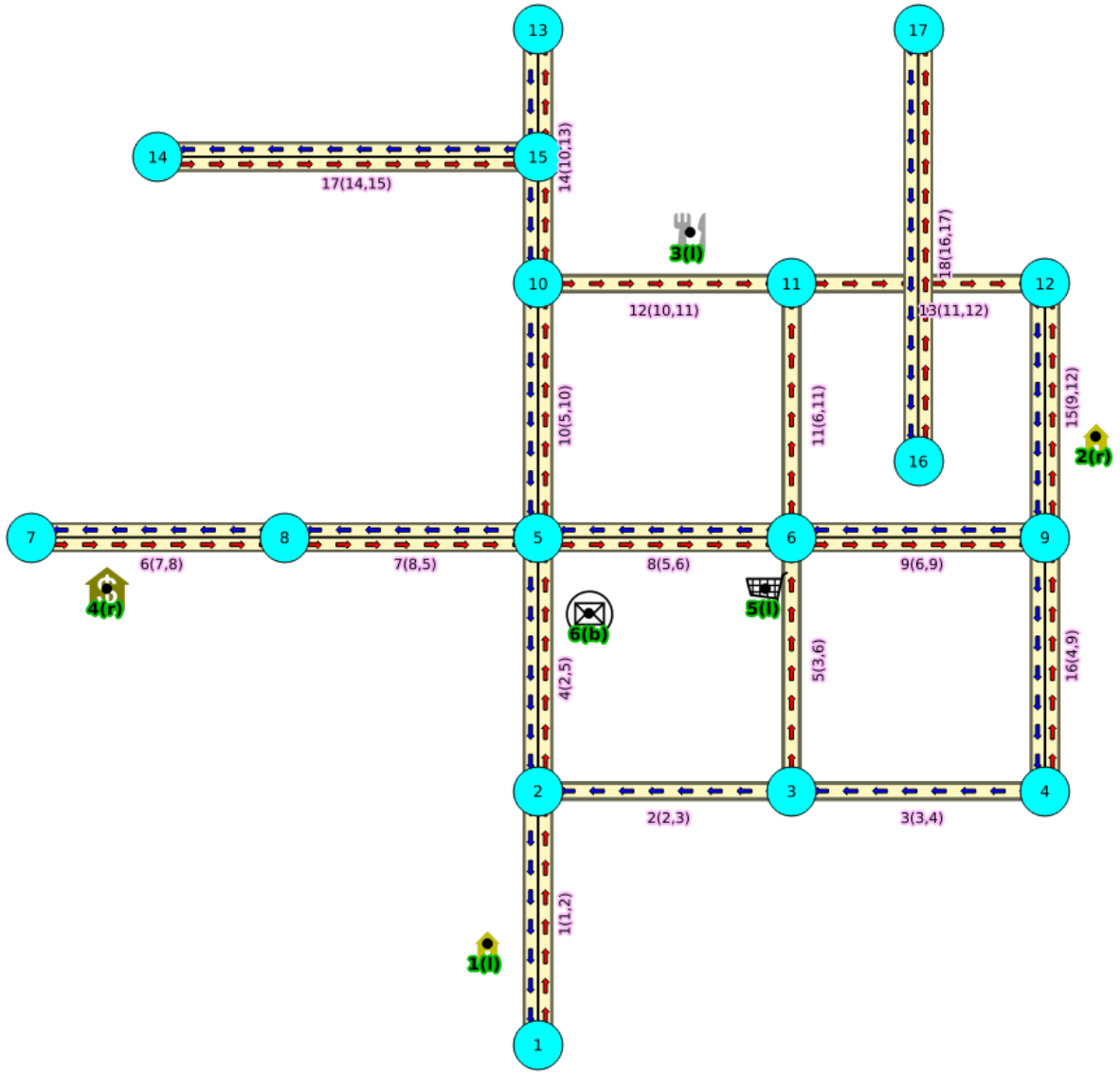


Fig. 3.1: **Graph 1: Directed, with cost and reverse cost**

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.

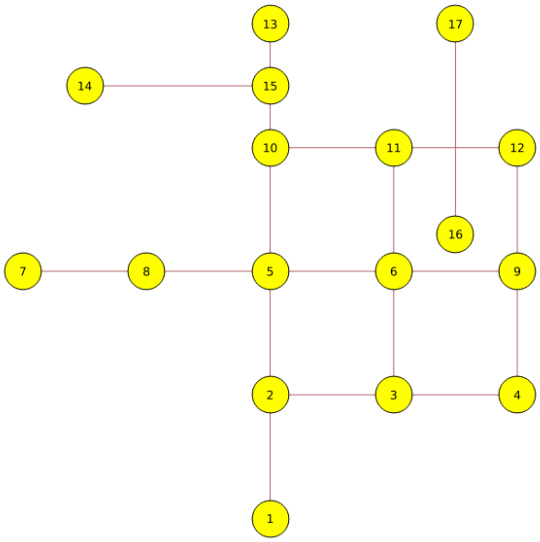


Fig. 3.2: **Graph 2: Undirected, with cost and reverse cost**

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

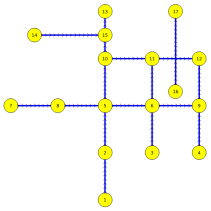


Fig. 3.3: **Graph 3: Directed, with cost**

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

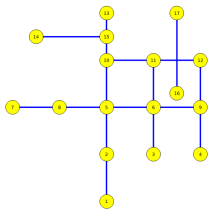


Fig. 3.4: **Graph 4: Undirected, with cost**

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* 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*

4.3 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.

4.3.1 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.

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 requirment 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 omited, as long as the value matches the default And The order of the parameters would not matter.

```
SELECT  pgr_createTopology('edge_table', 0.001,
        the_geom:='the_geom', id:='id', source:='source', target:='target');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('edge_table', 0.001,
        source:='source', id:='id', target:='target', the_geom:='the_geom');
 pgr_createtopology
--------------------
 OK
(1 row)
```

```
SELECT  pgr_createTopology('edge_table', 0.001, source:='source');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting rows using rows_where parameter

Selecting rows based on the id.

```
SELECT  pgr_createTopology('edge_table', 0.001, rows_where:='id < 10');
 pgr_createtopology
--------------------
 OK
```

```
(1 row)
```

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

```
SELECT  pgr_createTopology('edge_table', 0.001,
        rows_where:='the_geom && (SELECT st_buffer(the_geom, 0.05) FROM edge_table WHERE id=5)');
 pgr_createtopology
--------------------
 OK
(1 row)
```

Selecting the rows where the geometry is near the geometry of the row with `gid` =100 of the table `othertable`.

```
CREATE TABLE otherTable AS  (SELECT 100 AS gid,  st_point(2.5, 2.5) AS other_geom);
SELECT 1
SELECT  pgr_createTopology('edge_table', 0.001,
        rows_where:='the_geom && (SELECT st_buffer(other_geom, 1) FROM otherTable WHERE gid=100)'
 pgr_createtopology
--------------------
 OK
(1 row)
```

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

For the following table

```
CREATE TABLE mytable AS (SELECT id AS gid,  the_geom AS mygeom, source AS src , target AS tgt FRO
SELECT 18
```

Using positional notation:

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

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

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

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

```
SELECT  pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('mytable', 0.001, 'gid', 'mygeom', 'src', 'tgt', rows_where := 'true
NOTICE:  Performing checks, please wait .....
NOTICE:  ----> PGR ERROR in pgr_createTopology: Wrong type of Column id:mygeom
NOTICE:  Unexpected error raise_exception
 pgr_createtopology
--------------------
 FAIL
(1 row)
```

When using the named notation

In this scenario omitting a parameter would create an error because the default values for the column names do not match the column names of the table. The order of the parameters do not matter:

```
SELECT  pgr_createTopology('mytable', 0.001, the_geom:='mygeom', id:='gid', source:='src', target
 pgr_createtopology
--------------------
 OK
(1 row)
```

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

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.

```
varchar pgr_createVerticesTable(text edge_table,  text the_geom:='the_geom'
                    text source:='source',text target:='target',text rows_where:='true')
```

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:='target
```

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

Parameters defined with a default value can be omited, 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 WHERE
```

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.

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 completly 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 ommited, 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
    NOTICE:                            Dead ends: 3
    NOTICE:  Potential gaps found near dead ends: 0
    NOTICE:               Intersections detected: 0
    NOTICE:                      Ring geometries: 0

     pgr_analyzeGraph
    -------------------
     OK
    (1 row)
SELECT pgr_createTopology('edge_table', 0.001,rows_where:='id <17');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.001,'the_geom','id','source','target','id <17')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  --------------> TOPOLOGY CREATED FOR  16 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  -----------------------------------------------
```

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

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

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

The examples use the *Sample Data* network.

See Also

- *Routing Topology* for an overview of a topology for routing algorithms.
- *pgr_analyzeOneway* to analyze directionality of the edges.
- *pgr_createVerticesTable* to reconstruct the vertices table based on the source and target information.
- *pgr_nodeNetwork* to create nodes to a not noded edge table.

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.

pgr_nodeNetwork

Name

`pgr_nodeNetwork` - Nodes an network edge table.

Author Nicolas Ribot

Synopsis

The function reads edges from a not "noded" network table and writes the "noded" edges into a new table.

```
text pgr_nodenetwork(text edge_table, float8, tolerance,
                     text  id='id', text the_geom='the_geom',text table_ending='noded'
         text rows_where ='', boolean outall=false)
```

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:    Splitted Edges: 3
NOTICE:   Untouched Edges: 15
NOTICE:       Total original Edges: 18
NOTICE:   Edges generated: 6
NOTICE:   Untouched Edges: 15
NOTICE:         Total New segments: 21
NOTICE:   New Table: public.edge_table_noded
NOTICE:  ----------------------------------
pgr_nodenetwork
-----------------
 OK
(1 row)
```

Inspecting the generated table, we can see that edges 13,14 and 18 has been segmented

```
SELECT old_id,sub_id FROM edge_table_noded ORDER BY old_id,sub_id;
 old_id | sub_id
```

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

We can create the topology of the new network

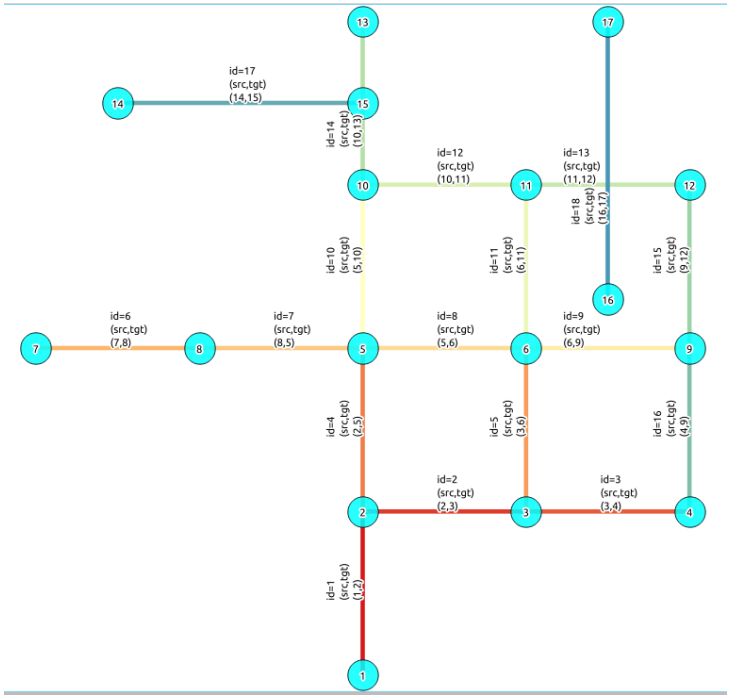
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  --------------> TOPOLOGY CREATED FOR  21 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table_noded is: public.edge_table_noded_vertices_pg
NOTICE:  -----------------------------------------------
pgr_createtopology
--------------------
 OK
(1 row)
```

Now let's analyze the new topology

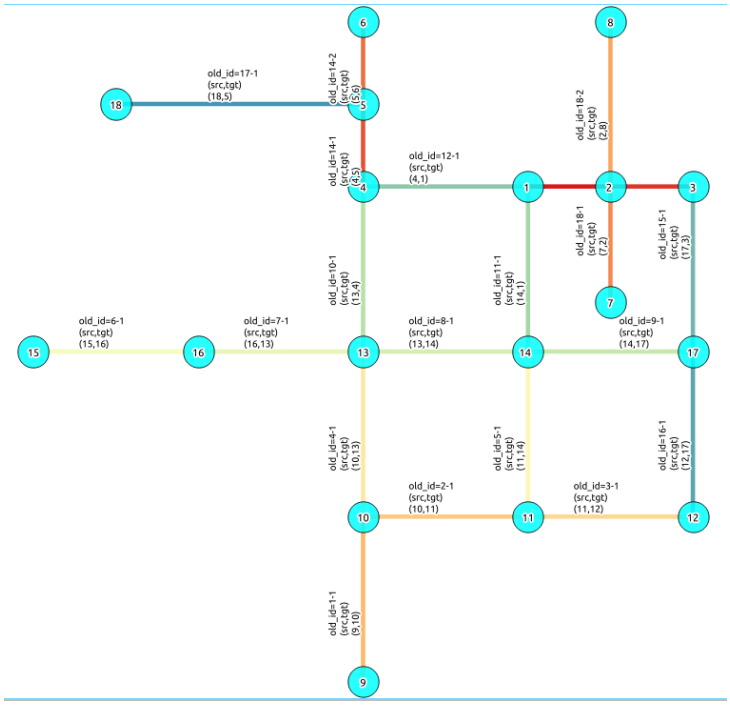
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...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

Images

Before Image



After Image



Comparing the results

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

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

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

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

```
alter table edge_table drop column if exists old_id;
alter table edge_table add column old_id integer;
insert into edge_table (old_id,dir,cost,reverse_cost,the_geom)
        (with
        segmented as (select old_id,count(*) as i from edge_table_noded group by old_id)
        select  segments.old_id,dir,cost,reverse_cost,segments.the_geom
                from edge_table as edges join edge_table_noded as segments on (edges.id = segments
                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.

4.3.2 Routing Functions

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)
- *pgr_tsp* - Traveling Sales Person

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. 4.1: 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,

Signature

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

Example

```
pgr_floydWarshall(
   'SELECT source, target, cost, reverse_cost FROM edge_table WHERE city_code = 304'
);
```

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.
- **Recomended, use a bounding box of no more than 3500 edges.**

Description of the Signature

Description of the SQL query

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

Column	Type	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)*, 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, 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`.

Examples

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)
```

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)
```

These queries uses the *Sample Data* network.

History

- Re-design of pgr_apspWarshall in Version 2.2.0

See Also

- *pgr_johnson*
- Boost floyd-Warshall[2] algorithm

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 Johnson's algorithm.

Fig. 4.2: 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*. We make use of the Boost's implementation which runs in $O(VE \log V)$ time,

Signature

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

Example

```
pgr_johnson(
   'SELECT source, target, cost, reverse_cost FROM edge_table WHERE city_code = 304'
);
```

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

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.

Description of the Signature

Description of the SQL query

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

Column	Type	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)*, 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, 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`.

Examples

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)
```

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)
```

These queries uses the *Sample Data* network.

History

- Re-design of pgr_apspJohnson in version 2.2.0

See Also

- *pgr_floydWarshall*
- Boost Jhonson[4] algorithm implementation.

Indices and tables

- genindex
- search

Performance

The following tests:

- non server computer
- with AMD 64 CPU

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

- 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 $\frac{E}{V \times (V-1)}$.

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

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

Johnson is the average execution time in seconds of pgr_johnson.

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

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

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

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

pgr_astar - Shortest Path A*

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:

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

```
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);
 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
    4, 1, true, true);
 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_costResult[]*
- http://en.wikipedia.org/wiki/A*_search_algorithm

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 searchs from the source toward the distination and at the same time from the destination to the source and terminates whe these to searchs 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

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 searchs from the source toward the distination and at the same time from the destination to the source and terminates whe these to searchs 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 |   3 |    0
   1 |   3 |   2 |    0
   2 |   2 |   4 |    1
   3 |   5 |  10 |    1
   4 |  10 |  -1 |    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 |   3 |    1
   1 |   3 |   2 |    1
   2 |   2 |   4 |    1
   3 |   5 |  10 |    1
   4 |  10 |  -1 |    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

pgr_dijkstra - Shortest Path Dijkstra

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

[6]

Fig. 4.3: 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
- Runing 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)
```

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)
```

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)
```

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)
```

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 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.
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, INTEGER, BIGINT

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

Description of the parameters of the signatures

Column	Type	Description
sql	TEXT	SQL query as decribed 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 considere `true` which considers the graph as Directed.

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 begining 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`.

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)

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)
```

Equvalences between signatures 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)

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.

History

- Renamed in version 2.0.0
- Added functionality in version 2.1.0

See Also

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

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.

[7]

Fig. 4.4: 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 symetric.
  - 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
- Runing 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 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 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 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 edge's SQL query

edges_sql is a `TEXT` that containes 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.
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, 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 decribed above.
start_vid	BIGINT	Identifier of the starting vertex of the path.
end_vid	BIGINT	Identifier of the ending vertex of the path.
start_vids	array[ANY-INTEGER]	Array of identifiers of starting vertices.
end_vids	array[ANY-INTEGER]	Array of identifiers of ending vertices.
directed	BOOLEAN	(optional). When false the graph is considere 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 1

Repeated values are ignored, and arrays are 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

start_vids are 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)
```

The queries use the *Sample Data* network.

History

- New in version 2.2.0

See Also

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

Indices and tables

- genindex
- search

The problem definition

Given the following query:

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

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

and

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

The graphs are defined as follows:

Directed graph

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

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

Undirected graph

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

- the set of vertices V
  - $V = source \cup target \cup start_v vid \cup end_{vid}$
- the set of edges E

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

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

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 of

- $path$ 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.

Driving Distance

- *pgr_drivingDistance* - Driving Distance

Driving Distance post-processing

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

pgr_drivingDistance

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

[8]

Fig. 4.5: 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.

The minimal signature:

```
pgr_drivingDistance(sql text, start_v bigint, distance float8)
  RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Driving Distance from a single starting point:

```
pgr_drivingDistance(sql text, start_vid bigint, distance float8, directed boolean)
  RETURNS SET OF (seq, node, edge, cost, agg_cost)
```

Driving Distance from a multiple starting points:

```
pgr_drivingDistance(sql text, start_vids anyarray, distance float8,
      directed boolean default true,
      equicost boolean default false)
  RETURNS SET OF (seq, start_vid, node, edge, cost, agg_cost)
```

Description of the SQL query

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 `ANY-INTEGER` identifier of the edge.

source `ANY-INTEGER` identifier of the source vertex of the edge.

target `ANY-INTEGER` identifier of the target vertex of the edge.

cost ANY-NUMERICAL value of the edge traversal cost. A negative cost will prevent the edge (source, target) from being inserted in the graph.

reverse_cost ANY-NUMERICAL (optional) the value for the reverse traversal of the edge. A negative cost will prevent the edge (target, source) from being inserted in the graph.

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

sql SQL query as decribed above.

start_v BIGINT id of the starting vertex.

start_v array[ANY-INTEGER] array of id 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 Undirected. Default is true which considers the graph as Directed.

equicost boolean (optional). When true the node will only appear in the closest start_v list. Default is false which resembles several calls using the single starting point signatures. Tie brakes are arbitrarely.

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

seq INT row sequence.

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

node BIGINT id of the node within the limits from start_v.

edge BIGINT id of the edge used to arrive to node. 0 when the node is the start_v.

cost FLOAT cost to traverse edge.

agg_cost FLOAT total cost from start_v to 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*

```
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)
```

The queries use the *Sample Data* network.

History

- Renamed in version 2.0.0
- Added functionality in version 2.1

See Also

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

Indices and tables

- genindex
- search

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[9].

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
```

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

```
 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
   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 FR
) 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[10].

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

History

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

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_geo
        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

pgr_ksp

Name

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

[11]

Fig. 4.6: Boost Graph Inside

Synopsis

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

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

The minimal signature:

```
pgr_ksp(TEXT sql_q, BIGINT start_vid, BIGINT end_vid, INTEGER k);
  RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

The full signature:

```
pgr_ksp(TEXT edges_sql, BIGINT start_vid, BIGINT end_vid, INTEGER k,
      BOOLEAN directed:=true, BOOLEAN heap_paths:=false);
  RETURNS SET OF (seq, path_id, path_seq, node, edge, cost, agg_cost) or EMPTY SET
```

Description of the SQL query

General description of the `edges_sql`

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

sql_q a SQL query, which returns a set of rows with the following columns:

id `ANY-INTEGER` identifier of the edge.

source `ANY-INTEGER` identifier of the source vertex of the edge.

target `ANY-INTEGER` identifier of the target vertex of the edge.

cost `ANY-NUMERICAL` value of the edge traversal cost. A negative cost will prevent the edge (`source`, `target`) from being inserted in the graph.

reverse_cost `ANY-NUMERICAL` (optional) the value for the reverse traversal of the edge. A negative cost will prevent the edge (`target`, `source`) from being inserted in the graph.

Where:

ANY-INTEGER smallint, int, bigint

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

Description of the parameters of the signatures

sql_q `TEXT` SQL query as decribed above.

start_vid `BIGINT` id of the starting vertex.

end_vid `BIGINT` id 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)`

seq `INT` sequential number starting from **1**.

path_seq `INT` relative position in the pathi of `node` and `edge`. Has value **1** for the begining 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` id of the node in the path.

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 route.

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`.

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 will be used.

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)
```

The queries use the *Sample Data* network.

History

- New in version 2.0.0
- Added functionality version 2.1

See Also

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

Indices and tables

- genindex
- search

pgr_tsp - Traveling Sales Person

Name

- `pgr_tsp` - Returns the best route from a start node via a list of nodes.
- `pgr_tsp` - Returns the best route order when passed a disance matrix.
- `_pgr_makeDistanceMatrix` - Returns a Eucleadian distance Matrix from the points provided in the sql result.

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 algoritm 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[]*:

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.

[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.

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[12] 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
);

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
```

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

```
   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_vertices

  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

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

pgr_trsp - Turn Restriction Shortest Path (TRSP)

Name

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

Synopsis

The turn restricted shorthest path (TRSP) is a shortest path algorithm that can optionally take into account complicated turn restrictions like those found in real work 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` commar seperated 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 seperated 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,
        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,
        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, 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, 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[]*

4.4 Routing Functions

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)
- *pgr_tsp* - Traveling Sales Person

CHAPTER 5

Available Functions but not official pgRouting functions

- *Proposed Functions for version 2.3*
- *Experimental and Proposed functions*

5.1 Proposed Functions for version 2.3

Proposed Functions for version 2.3

This are proposed functions for version 2.3.

- They are not officially in the version 2.2 release.
- They will likely officialy be part of the version 2.3 release:
  - The functions make use of ANY-INTEGER and ANY-NUMERICAL
  - Name might not change.
  - Signature might not change.
  - Functionality might not change.
  - pgTap tests have being done.
  - Needs feedback from the comunity.

5.1.1 Proposed Routing Functions

author Vicky Vergara

- As part of Dijkstra Family of functions
  - *pgr_dijkstraVia* - Use pgr_dijkstra to make a route via vertices.
- *pgr_withPoints* - withPoints family functions
  - *pgr_withPoints* - Route from/to points anywhere on the graph.
  - *pgr_withPointsCost* - Costs of the shortest paths.
  - *pgr_withPointsKSP* - K shortest paths with points.
  - *pgr_withPointsDD* - Driving distance.

pgr_dijkstraVia

Name

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

[1]

Fig. 5.1: 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 for version 2.3

Signatrue Summary

```
pgr_dijkstraVia(edges_sql, via_vertices)
pgr_dijkstraVia(edges_sql, via_vertices, directed:=true, strict:=false, U_turn_on_edge:=true)

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:=true, strict:=false, U_turn_on_edge:=true)
RETURNS SET OF (seq, path_pid, path_seq, start_vid, end_vid,
    node, edge, cost, agg_cost, route_agg_cost) or EMPTY SET
```

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

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

Description of the Signature

Description of the SQL query

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

Column	Type	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)*, 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, INTEGER, BIGINT

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

Description of the parameters of the signatures

Recives `(edges_sql, via_vertices, directed:=true, strict:=false, U_turn_on_edge:=true)`

Parameter	Type	Description
edges_sql	TEXT	SQL query as decribed 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)
```

The queries use the *Sample Data* network.

History

- Proposed in version 2.2

See Also

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

Indices and tables

- genindex
- search

withPoints - Family of functions

When points are also given as input:

- *pgr_withPoints* - Route from/to points anywhere on the graph.
- *pgr_withPointsCost* - Costs of the shortest paths.
- *pgr_withPointsKSP* - K shortest paths.
- *pgr_withPointsDD* - Driving distance.

Note: The numbering of the points are handled with negative sign.

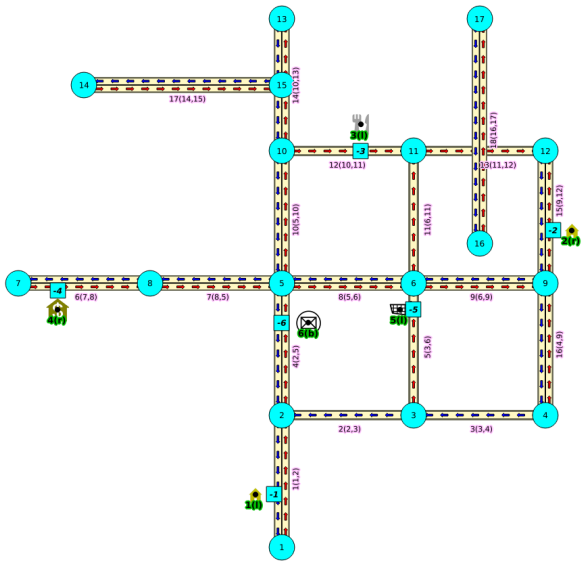
- Original point identifiers are to be positive.
- Transformation to negative is done internally.
- For results for involving vertices identifiers
  - positive sign is a vertex of the original grpah
  - negative sign is a point of the temporary points

The reason for doing this is to avoid confusion when there is a vertex with the same number as identifier as the points identifier.

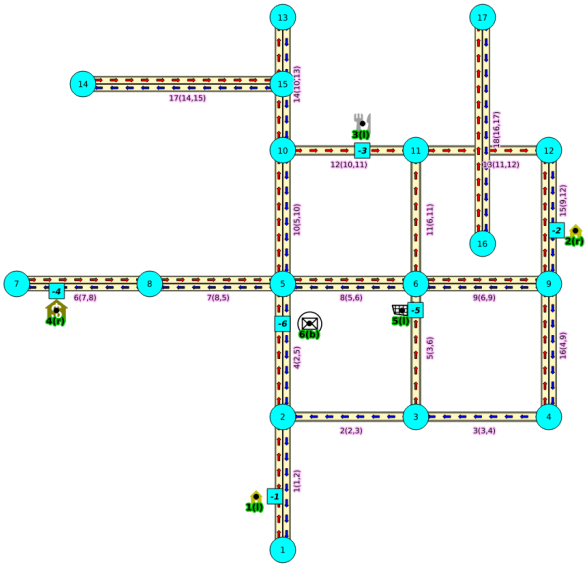
Images

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.

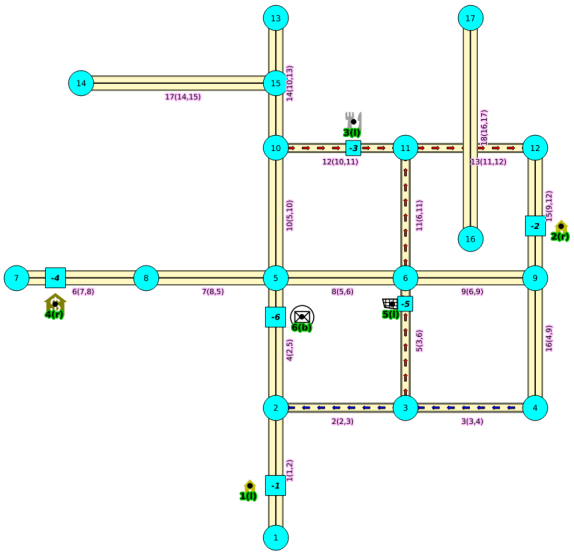
Right driving side



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 thru a web application

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

pgr_withPoints

Name `pgr_withPoints` - Returns the shortest path in a graph with additional temporary vertices.

Note: This is a proposed function for version 2.3.

- Is not officially in the version 2.2 release.

[2]

Fig. 5.2: 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

- Runing 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 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.
- 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	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)*, 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.

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.** • 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

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.

Note: This is a proposed function for version 2.3.

- Is not officially in the version 2.2 release.

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.

Fig. 5.3: Boost Graph Inside

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 family of functions.

Signatures

Minimal Usage

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	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-NUMERIC	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-NUMERIC	(optional) Weight of the edge *(target, source)*, If negative: edge *(target, source)* does not exist, therefore it's not part of the graph.

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.** • 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_withPointsKSP

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

Note: This is a proposed function for version 2.3.

- Is not officially in the version 2.2 release.

[4]

Fig. 5.4: 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_side
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	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)*, 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.

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.** • 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

Name `pgr_withPointsDD` - Returns the driving distance from a starting point.

Note: This is a proposed function for version 2.3.

- Is not officially in the version 2.2 release.

[5]

Fig. 5.5: 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 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.
- 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	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)*, 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.

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.** • 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

5.2 Experimental and Proposed functions

Experimental and Proposed functions

These are proposed functions

- They are not officially of the current release.
- They likely will not be officialy be part of the version 2.3 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 not being done.
  - 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 need a non official function of pgRouting
  - Might need a deprecated function of pgRouting

5.2.1 Proposed functions: Proposed by Steve Woodbridge

Convenience Functions

- *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`.

5.2.2 Proposed functions: Proposed by Zia Mohammed

- *pgr_labelGraph* - Analyze / label subgraphs within a network

pgr_labelGraph

Name

`pgr_labelGraph` — Locates and labels sub-networks within a network which are not topologically connected. Must be run after `pgr_createTopology()`. No use of `geometry` column. Only `id`, `source` and `target` columns are required.

Synopsis

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

Convenience Functions

Warning: This are proposed function
- Is not officially in the release.
- Name could change.
- Signature could change.
- Needs testing.
- Functionality could change.

The following functions are general purpose convenience functions that might be useful when building a larger application or handling input from say an Ajax handler.

- *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.

Distance Matrix Functions

These function my be helpful when you need to create or manipulate distance matricies, like for TSP or VRP related problems.

- *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`.

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

See also

Indices and tables

- genindex
- search

pgr_pointToEdgeNode

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

> **Warning:** This is a proposed function
> - Is not officially in the release.
> - Name could change.
> - Signature could change.
> - Needs testing.
> - Functionality could change.

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

Indices and tables

- genindex
- search

pgr_flipEdges

Name `pgr_flipEdges` -

> **Warning:** This is a proposed function
> - Is not officially in the release.
> - Name could change.
> - Signature could change.
> - Needs testing.
> - Functionality could change.

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

- 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;
      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_textToPoints

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

Warning: This is a proposed function
- Is not officially in the release.
- Name could change.
- Signature could change.
- Needs testing.
- Functionality could change.

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

Paramters

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;
 st_astext
------------
 POINT(2 0)
 POINT(2 1)
 POINT(3 1)
 POINT(2 2)
(4 rows)
```

See Also

- *pgr_pointToEdgeNode* - convert a point geometry to a `node_id` based on closest edge.
- *pgr_pointsToVids* - convert an array of point geometries into vertex ids.

Indices and tables

- genindex
- search

pgr_pointsToVids

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

Warning: This is a proposed function
- Is not officially in the release.
- Name could change.
- Signature could change.
- Needs testing.
- Functionality could change.

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

Paramters

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'
);
 pgr_pointstovids
------------------
 {1,2,3,5}
(1 row)
```

This example uses the *Sample Data* network.

See Also

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

Indices and tables

- genindex
- search

pgr_pointsToDMatrix

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

Warning: This is a proposed function
- Is not officially in the release.
- Name could change.
- Signature could change.
- Needs testing.
- Functionality could change.

Synopsis Create a distance symetric 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

Paramters

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));
                                                  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
);
 seq | id
-----+----
   0 |  1
   1 |  3
```

```
   2 |  2
   3 |  0
(4 rows)
```

See Also

- *pgr_vidsToDMatrix* - convert a point geometry to the closest vertex_id of an edge..
- *pgr_tsp* - Traveling Sales Person

Indices and tables

- genindex
- search

pgr_vidsToDMatrix

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

Warning: This is a proposed function
- Is not officially in the release.
- Name could change.
- Signature could change.
- Needs testing.
- Functionality could change.

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

Paramters

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 shoule 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'
);
                  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* - is used to convert the locations into point geometries.
- *pgr_pointsToVids* - 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');
                  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
);
 seq | id
-----+----
   0 |  1
   1 |  2
   2 |  3
   3 |  0
(4 rows)
```

This example uses the *Sample Data* network.

See Also

- *pgr_textToPoints* - Create an array of points from a text string.
- *pgr_tsp* - Traveling Sales Person

Indices and tables

- genindex
- search

pgr_vidsToDMatrix

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

Warning: This is a proposed function
- Is not officially in the release.
- Name could change.
- Signature could change.
- Needs testing.
- Functionality could change.

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

Paramters

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 shoule 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);
             pgr_vidstodmatrix
-------------------------------------------
 {{0,1,6,2},{1,0,5,1},{2,1,0,2},{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);
             pgr_vidstodmatrix
-------------------------------------------
 {{0,1,4,2},{1,0,3,1},{4,3,0,3},{2,1,3,0}}
(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
);
 seq | id
-----+----
   0 |  1
   1 |  2
   2 |  3
   3 |  0
(4 rows)
```

This example uses the *Sample Data* network.

See Also

- *pgr_vidsToDMatrix* - - Creates a distances matrix from an array of `vertex_id`.
- *pgr_textToPoints* - Create an array of points from a text string.
- *pgr_tsp* - Traveling Sales Person

Indices and tables

- genindex
- search

5.2.3 Proposed functions: Proposed by Rohith Reddy

- *Contraction* - Reduce network size using contraction techniques

Contraction

Contracting a graph becomes a crucial operation when talking about big graphs like the graphs involved in routing across cities, countries, continents or the whole world.

The contraction level and contraction operations can become very complex, as the complexity of the graphs grows.

For this proposal, we are making our contraction algorithm simple as possible so that more contraction operations can be added in the future.

We are not aiming with this work to implement all the possible contraction operations but to give a framework such that adding a contraction operation can be easily achieved.

For this contraction proposal I am only making 2 operations:

1. dead end contraction: vertices have one incoming edge
2. linear contraction: vertices have one incomming and one outgoing edge

And with the additional characteristics:

- The user can forbid to contract a particular set of nodes or edges.
- The user can decide how many times the cycle can be done.
- If possible, the user can decide the order of the operations on a cycle.

Note: Work on progress in contraction branch

The contraction skeleton

In general we have an initial set up that may involve analizing the graph given as input and setting the non contractable nodes or edges. We have a cycle that will go and perform a contraction operation until while possible, and then move to the next contraction operation. Adding a new operation then becomes an "easy" task; more things might be involved, because the charachteristics of the graph change each time its contracted, so some interaction between contractions has to be implemented also.

Procedure

- For contracting, we are going to cycle as follows

```
input: G(V,E);
removed_vertices = {};


<initial set up>
do N times {

    while ( <conditions for 1> ) {
        < contraction operation 1 >
    }

    while ( <conditions for 2> ) {
        < contraction operation 2>
    }
    .....
}
```

```
output: G'(V',E'), removed_vertices
```

Contraction operations for this implementation

Dead end contraction Characteristics:

- $V1$: set of vertices with 1 incoming edge in increasing order of id:
  - Edges with the same identifier are considered the same edge and if it has the *reverse_cost* valid the outgoing edge is ignored

```
while ( V1 is not empty ) {

    delete vertex of V1
    the deleted vertex add it to removed_vertices
    vertex that leads to removed vertex, inherits the removed vertex

    <adjust any conditions that might affect other contraction operation>
}
```

Linear contraction Characteristics:

- $V2$: vertex with 1 incoming edge and 1 outgoing edge:
  - The outgoing edge must have different identifier of the incomming edge

```
while ( V2 is not empty ) {

    delete vertex of V2
    create edge (shortcut)
    the deleted vertex add it to removed_vertices
    inewly created edge, inherits the removed vertex

    <adjust any conditions that might affect other contraction operations>
}
```

Notation

- V: is the set of vertices
- E: is the set of edges
- G: is the graph
- $V1$: is the set of *dead end* vertices
- $V2$: is the set of *linear* vertices
- removed_vertices: is the set of removed vertices

The contracted graph will be represented with two parameters, the modified Graph, and the removed_vertices set.

removed_vertices = {(v,1):{2}, (e,-1):{3}}.

The above notation indicates:

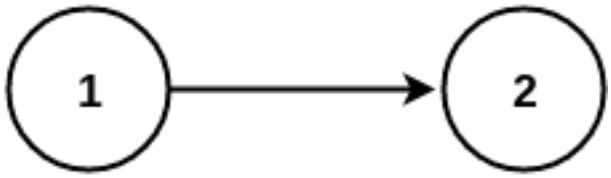
- Vertex 2 is removed, and belongs to vertex 1 subgraph
- Vertex 3 is removed, and belongs to edge -1 subgraph

Examples

For simplicity all the edges in the examples have unit weight.

Dead End

- Perform dead end contraction operation first and then linear contraction
- 1 cycle of contraction.



Input G = {V:{1, 2}, E:{(1, 2)}}

initial set up

```
removed_vertices={}
V1 = {2}
V2 = {}
```

procedure

```
V1 = {2} is not empty

    V1 = {}
    V2 = {}
    G = {V:{1}, E:{}}
    removed_vertices = {(v, 1):{2}}.

V1 is empty
```

Since V1 is empty we go on to the next contraction operation

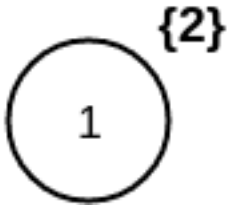
```
V2 is empty
```

So we do not perform any linear contraction operation.

Results

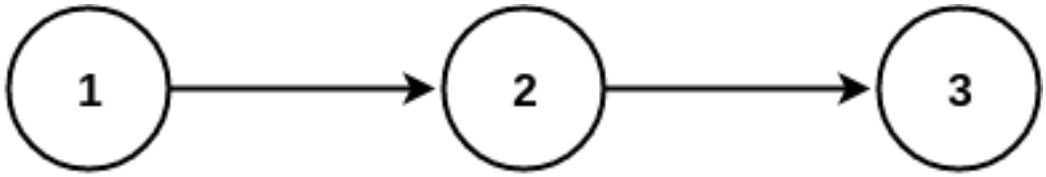
```
G = {V:{1}, E:{}}
removed_vertices = {(v, 1):{2}}
```

Visualy the results are



Linear contraction

- Perform linear contraction operation first and then dead end contraction
- 1 cycle of contraction.



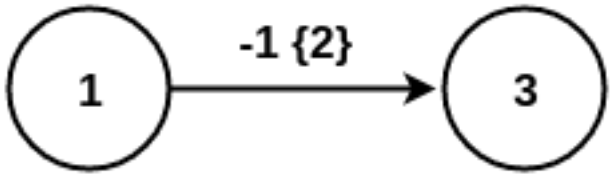
Input G = {V:{1, 2, 3}, E:{(1, 2), (2, 3)}}

initial set up

```
removed_vertices={}
V1 = {3}
V2 = {2}
```

procedure

```
V2 = {2} is not empty

    V1 = {3}
    removed_vertices = {(e, -1):{2}}
    V2 = {}
    G = {V:{1, 3}, E:{-1(1,3,c=2)}}

V2 is empty
```



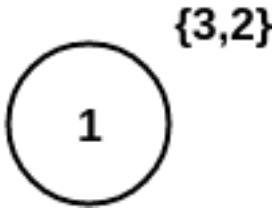
Since V2 is empty we go on to the next contraction operation

```
V1 = {3} is not empty

    V1 = {}
    V2 = {}
    removed_vertices = {(v, 1):{3, 2}}.
    G = {V:{1}, E:{}}

V1 is empty
```

Results

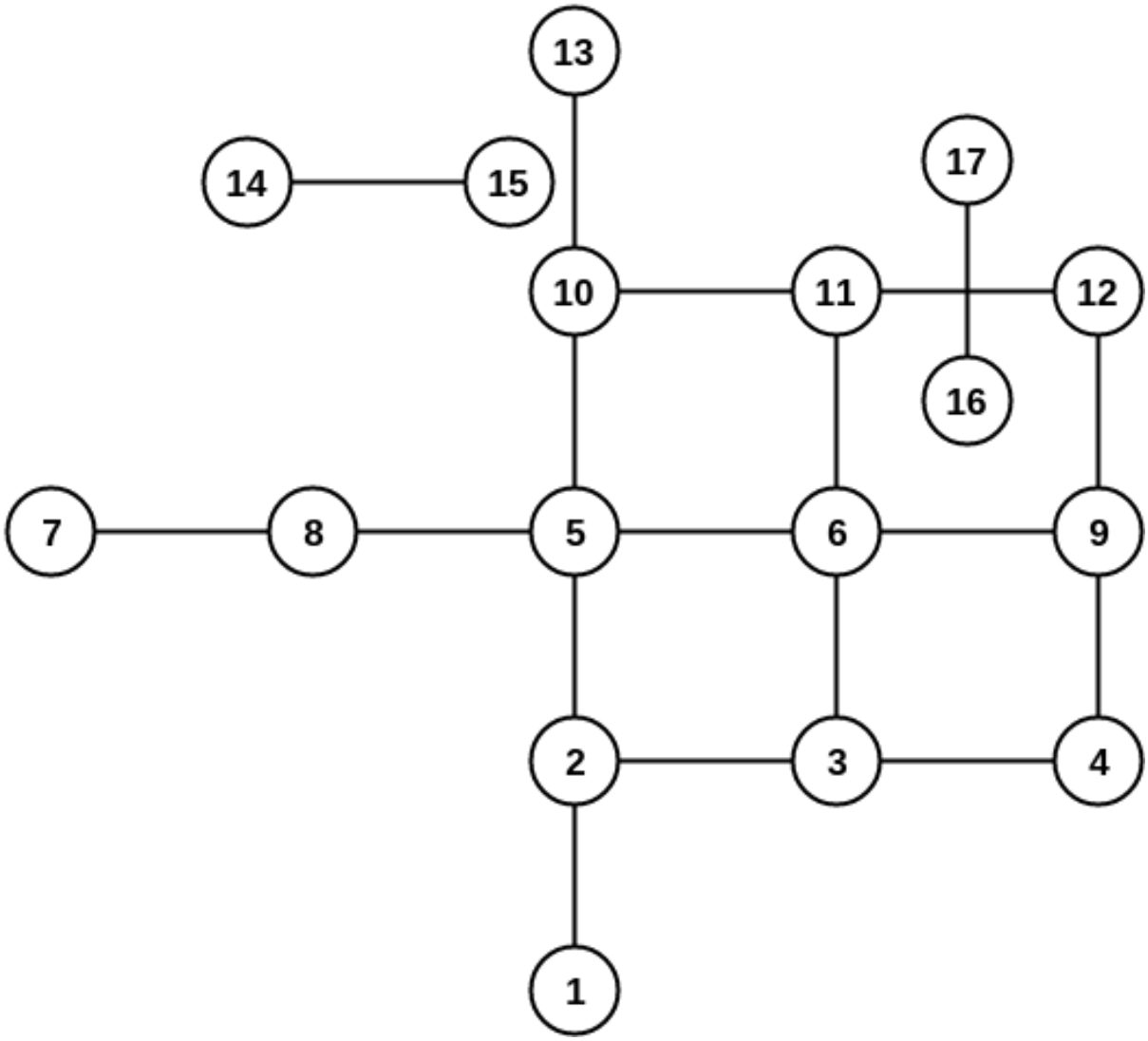
```
removed_vertices = {(v, 1):{3, 2}}.
G = {V:{1}, E:{}}
```

Visualy the results are



Sample Data

- Perform dead end contraction operation first and then linear contraction
- 1 cycle of contraction.



Input G = {V:{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}, E:{1(1, 2), 2(2,3), 3(3,4), 4(2,5), 5(3,6), 6(7,8), 7(8,5), 8(5,6), 9(6,9), 10(5,10), 11(6,11), 12(10,11), 13(11,12), 14(10,13), 15(9,12), 16(4,9), 17(14,15), 18(16,17)}}

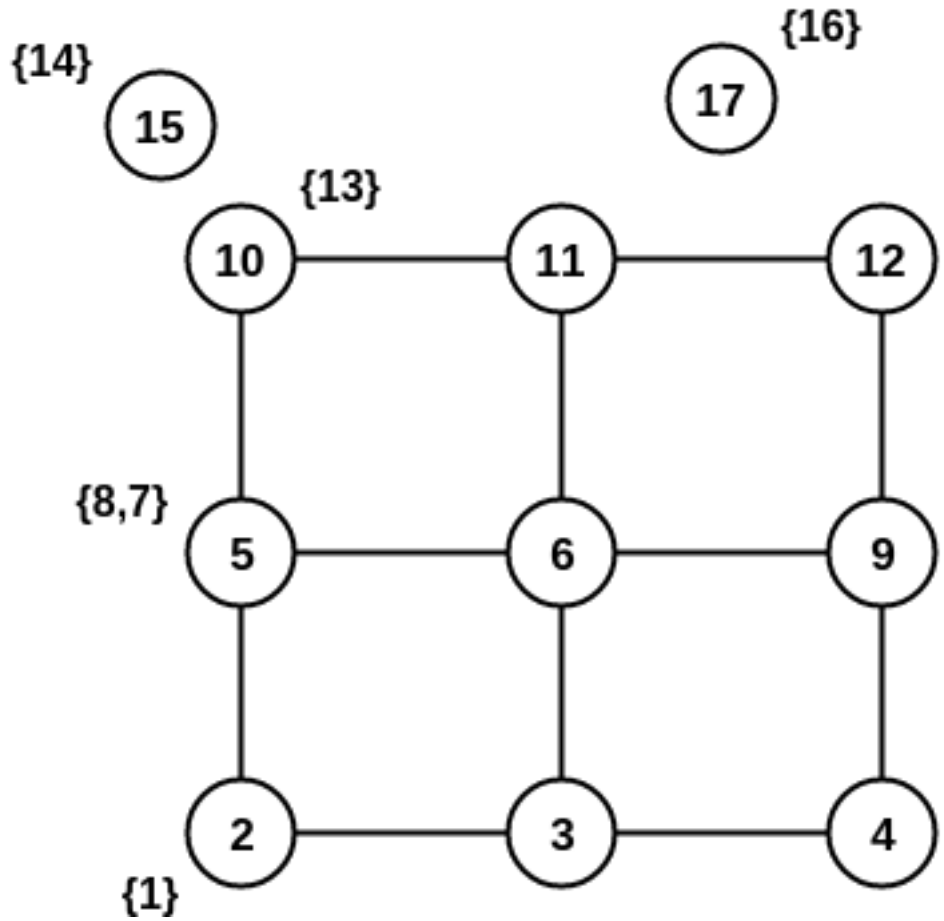
initial set up

```
removed_vertices={}
V1 = {1,7,13,14,15,16,17}
V2 = {4,8,12}
```

procedure

```
V1 = {1,7,13,14,15,16,17} is not empty

    V1 = {7,13,14,15,16,17}
    V2 = {2,4,8,12}
    G = {V:{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17},
    E:{2(2,3), 3(3,4), 4(2,5), 5(3,6), 6(7,8), 7(8,5), 8(5,6), 9(6,9),
        10(5,10), 11(6,11), 12(10,11), 13(11,12), 14(10,13), 15(9,12), 16(4,9), 17(14,15), 18(16,
    removed_vertices = {(v, 2):{1}}.


V1 = {7,13,14,15,16,17} is not empty

    V1 = {8,13,14,15,16,17}
    V2 = {2,4,12}
```

```
    G = {V:{2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17},
    E:{2(2,3), 3(3,4), 4(2,5), 5(3,6), 7(8,5), 8(5,6), 9(6,9), 10(5,10),
        11(6,11), 12(10,11), 13(11,12), 14(10,13), 15(9,12), 16(4,9), 17(14,15), 18(16,17)}}
    removed_vertices = {(v, 2):{1}, (v,8):{7}}.

V1 = {8,13,14,15,16,17} is not empty

    V1 = {13,14,15,16,17}
    V2 = {2,4,12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17},
    E:{2(2,3), 3(3,4), 4(2,5), 5(3,6), 8(5,6), 9(6,9), 10(5,10),
         11(6,11), 12(10,11), 13(11,12), 14(10,13), 15(9,12), 16(4,9), 17(14,15), 18(16,17)}}
    removed_vertices = {(v, 2):{1}, (v,5):{8,7}}.

V1 = {13,14,15,16,17} is not empty

    V1 = {14,15,16,17}
    V2 = {2,4,10,12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 14, 15, 16, 17},
    E:{2(2,3), 3(3,4), 4(2,5), 5(3,6), 8(5,6), 9(6,9), 10(5,10),
         11(6,11), 12(10,11), 13(11,12), 15(9,12), 16(4,9), 17(14,15), 18(16,17)}}
    removed_vertices = {(v, 2):{1}, (v,5):{8,7}, (v,10):{13}}.

V1 = {14,15,16,17} is not empty

    V1 = {16,17}
    V2 = {2,4,10,12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 15, 16, 17},
    E:{2(2,3), 3(3,4), 4(2,5), 5(3,6), 8(5,6), 9(6,9), 10(5,10),
         11(6,11), 12(10,11), 13(11,12), 15(9,12), 16(4,9)}, 18(16,17)}
    removed_vertices = {(v, 2):{1}, (v,5):{8,7}, (v,10):{13}, (v,15):{14}}.

V1 = {16,17} is not empty

    V1 = {}
    V2 = {2,4,10,12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 15, 17},
    E:{2(2,3), 3(3,4), 4(2,5), 5(3,6), 8(5,6), 9(6,9), 10(5,10),
         11(6,11), 12(10,11), 13(11,12), 15(9,12), 16(4,9)}}
    removed_vertices = {(v, 2):{1}, (v,5):{8,7}, (v,10):{13}, (v,15):{14}, (v,17):{16}}.

Since V1 is empty we go on to the next contraction operation
```




```
V2 = {2,4,10,12} is not empty

    V1 = {}
    V2 = {4,10,12}
    G = {V:{3, 4, 5, 6, 9, 10, 11, 12, 15, 17},
    E:{-1(3,5), 3(3,4), 5(3,6), 8(5,6), 9(6,9), 10(5,10),
         11(6,11), 12(10,11), 13(11,12), 15(9,12), 16(4,9)}}
    removed_vertices = {(e, -1):{1,2}, (v, 2):{1}, (v,5):{8,7}, (v,10):{13}, (v,15):{14}, (v,17):

V2 = {4,10,12} is not empty

    V1 = {}
    V2 = {10,12}
    G = {V:{3, 5, 6, 9, 10, 11, 12, 15, 17},
    E:{-1(3,5),-2(3,9), 5(3,6), 8(5,6), 9(6,9), 10(5,10),
         11(6,11), 12(10,11), 13(11,12), 15(9,12)}}
    removed_vertices = {(e, -1):{1,2}, (e, -2):{4}, (v, 2):{1}, (v,5):{8,7}, (v,10):{13}, (v,15):

V2 = {10,12} is not empty

    V1 = {}
    V2 = {12}
    G = {V:{3, 5, 6, 9, 11, 12, 15, 17},
    E:{-1(3,5),-2(3,9), -3(5,11), 5(3,6), 8(5,6), 9(6,9),
         11(6,11), 13(11,12), 15(9,12)}}
    removed_vertices = {(e, -1):{1,2}, (e, -2):{4}, (e,-3):{10,13}, (v, 2):{1}, (v,5):{8,7}, (v,15

V2 = {12} is not empty

    V1 = {}
    V2 = {}
```

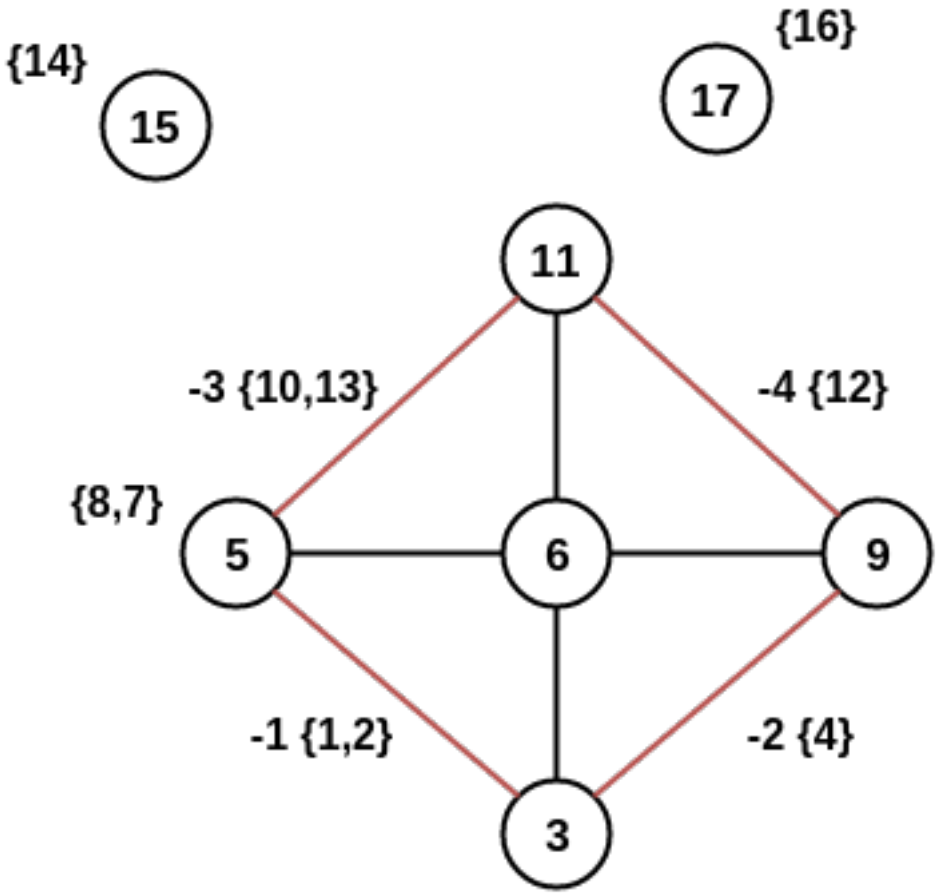
```
    G = {V:{3, 5, 6, 9, 11, 15, 17},
    E:{-1(3,5),-2(3,9), -3(5,11), -4(9,11), 5(3,6), 8(5,6), 9(6,9), 11(6,11)}}
    removed_vertices = {(e, -1):{1,2}, (e, -2):{4}, (e,-3):{10,13}, (e, -4):{12},
         (v, 2):{1}, (v,5):{8,7}, (v,15):{14}, (v,17):{16}}.
```

Since V1 and V2 are empty we stop our contraction here.

Results

```
G = {V:{3, 5, 6, 9, 11, 15, 17},
E:{-1(3,5),-2(3,9), -3(5,11), -4(9,11), 5(3,6), 8(5,6), 9(6,9), 11(6,11)}}
removed_vertices = {(e, -1):{1,2}, (e, -2):{4}, (e,-3):{10,13}, (e, -4):{12},
     (v, 2):{1}, (v,5):{8,7}, (v,15):{14}, (v,17):{16}}.
```

Visualy the results are



References

- 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

5.2.4 Experimental functions: by GSoC Students

- *pgr_vrpOneDepot* - VRP One Depot
- *pgr_gsoc_vrppdtw* - VRP Pickup & Delivery

pgr_vrpOneDepot

No documentation available from the original developer

Warning: this function is experimental and there is no assurance that it will work

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

pgr_gsoc_vrppdtw

Name

`pgr_gsoc_vrppdtw` — Returns optimized solution

Synopsis

Vehicle Routing Problem with Pickup and Delivery (VRPPD): A number of goods need to be moved from certain pickup locations to other delivery locations. The goal is to find optimal routes for a fleet of vehicles to visit the pickup and drop-off locations.

```
pgr_gsoc_vrppdtw(text sql, integer , integer;
```

Description

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

```
        select * from pgr_gsoc_vrppdtw(
                                        'select * from customer order by id'::text, 25,200
                                                );
```

Returns set of *pgr_costResult[]*:

seq row sequence

rid route ID

nid node ID (`-1` for the last row)

cost cost to traverse to `seq`

Examples

```
        SELECT  * from pgr_gsoc_vrppdtw(
                                        'select * from customer order by id'::text, 25,200
                                                );
```

```
 seq | rid  | nid  | cost
-----+------+------+------
   0 |    7 |    8 |    1
   1 |    8 |    9 |    1
   2 |    9 |   15 |    1
   3 |   12 |   -1 |    0
   . |    . |    . |    .
   . |    . |    . |    .
```

See Also

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

CHAPTER 6

Discontinued & Deprecated Functions

- *Discontinued Functions*
- *Deprecated Functions*

6.1 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.

6.1.1 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.

6.2 Deprecated Functions

Warning: This functions have being deprecated

6.2.1 Deprecated Routing Functions

Deprecated on version 2.2

- *pgr_apspJohnson* - Replaced with *pgr_johnson*
- *pgr_apspWarshall* - Replaced with *pgr_floydWarshall*
- *pgr_kDijkstra* - Replaced with *pgr_dijkstraCost*
- *pgr_drivingDistance* - Driving Distance (V2.0 signature). See new signature *pgr_drivingDistance*

pgr_apspJohnson

Name

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

Warning: This function is deprecated in version 2.2 Use *pgr_johnson* instead

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

Name

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

Warning: This function is deprecated in version 2.2 Use *pgr_floydWarshall* instead

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 - Mutliple destination Shortest Path Dijkstra

Name

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

Warning: This functions is deprecated in 2.2. Use *pgr_dijkstraCost* instead.

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

Warning: This function is deprecated in 2.2. Use *pgr_dijkstra* 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_costResult[]*
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

pgr_drivingDistance (V2.0)

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);
```

Warning: This signature is being deprecated on version 2.1, Please use it without the `has_rcost` flag instead:
`pgr_drivingDistance(sql, start_v, distance, directed)`
See *pgr_drivingDistance*

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_alphaShape* - Alpha shape computation
- *pgr_pointsAsPolygon* - Polygon around set of points

Deprecated on version 2.1

- *pgr_dijkstra* - Mutliple destination Shortest Path Dijkstra (V2.0 signature)
- *pgr_ksp* - K shortest paths (V2.0 signature)

pgr_dijkstra (V 2.0)- Shortest Path Dijkstra

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);
```

Warning: This signature is being deprecated in version 2.1, Please use it without the `has_rcost` flag instead:
`pgr_dijkstra(sql, source, target, directed)`
See *pgr_dijkstra*

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

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

pgr_ksp (V 2.0)

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);
```

Warning: This signature is being deprecated in version 2.1, Please use it without the `has_rcost` flag instead.

- for undirected graph. `pgr_ksp(sql, source, target, distance, directed:=false)`
- for directed graph. `pgr_ksp(sql, source, target, distance, directed:=true)`

See *pgr_ksp*

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_costResult3[] - Multiple Path Results with Cost*
- http://en.wikipedia.org/wiki/K_shortest_path_routing

6.2.2 Deprecated Developer's Auxiliary Functions

Deprecated on version 2.1

- *pgr_getColumnName* - to get the name of the column as is stored in the postgres administration tables.
- *pgr_getTableName* - to retrieve the name of the table as is stored in the postgres administration tables.
- *pgr_isColumnIndexed* - to check if the column is indexed.
- *pgr_isColumnInTable* - to check only for the existance of the column.
- *pgr_quote_ident* - to quotes the input text to be used as an identifier in an SQL statement string.
- *pgr_versionless* - to compare two version numbers.
- *pgr_startPoint* - to get the start point of a (multi)linestring.
- *pgr_endPoint* - to get the end point of a (multi)linestring.

Developers's Functions

Warning: This functions have being deprecated in version 2.1

The functions have being renamed but documentation will not be generated for any of the developer's functions.

- *pgr_getColumnName*
- *pgr_getTableName*
- *pgr_isColumnIndexed*
- *pgr_isColumnInTable*
- *pgr_pointToId*
- *pgr_quote_ident*
- *pgr_versionless*
- *pgr_startPoint*
- *pgr_endPoint*

pgr_getColumnName

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.

Warning: This function is deprecated in 2.1. Use *_pgr_getColumnName* instead

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 retrived.

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* to check only for the existance of the column.
- *pgr_getTableName* to retrieve the name of the table as is stored in the postgres administration tables.

pgr_getTableName

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.

Warning: This function is deprecated in 2.1 Use *_pgr_getTableName* instead

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* to check only for the existance of the column.
- *pgr_getTableName* to retrieve the name of the table as is stored in the postgres administration tables.

pgr_isColumnIndexed

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

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

Warning: This function is deprecated in 2.1 Use *_pgr_isColumnIndexed* instead

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* to check only for the existance of the column in the table.
- *pgr_getColumnName* to get the name of the column as is stored in the postgres administration tables.
- *pgr_getTableName* to get the name of the table as is stored in the postgres administration tables.

pgr_isColumnInTable

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

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

Warning: This function is deprecated in 2.1 Use *_pgr_isColumnInTable* instead

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* to check if the column is indexed.
- *pgr_getColumnName* to get the name of the column as is stored in the postgres administration tables.
- *pgr_getTableName* to get the name of the table as is stored in the postgres administration tables.

pgr_pointToId

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.

Use *pgr_createTopology* or *pgr_createVerticesTable* instead.

Warning: This function is deprecated in 2.1

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 corresponing `id`.
- When the `point` geometry is not found in the vertices table `vertname`, the funcion inserts the `point` and returns the corresponing `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

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.

Warning: This function is deprecated in 2.1 Use *_pgr_quote_ident* instead

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* to get the name of the table as is stored in the postgres administration tables.

pgr_versionless

Name `pgr_versionless` — Compare two version numbers.

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

Warning: This function is deprecated in 2.1. Use *_pgr_versionless* instead

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

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

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

Warning: This function is deprecated in 2.1 Use *_pgr_startPoint* instead

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* to get the end point of a (multi)linestring.

pgr_endPoint

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

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

Warning: This function is being deprecated on 2.1. Use *_pgr_endPoint* instead

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* to get the start point of a (multi)linestring.

CHAPTER 7

Change Log

Release Notes

- *pgRouting 2.2.4 Release Notes*
- *pgRouting 2.2.3 Release Notes*
- *pgRouting 2.2.2 Release Notes*
- *pgRouting 2.2.1 Release Notes*
- *pgRouting 2.2.0 Release Notes*
- *pgRouting 2.1.0 Release Notes*
- *pgRouting 2.0 Release Notes*
- *pgRouting 1.x Release Notes*

7.1 Release Notes

- *pgRouting 2.2.4 Release Notes*
- *pgRouting 2.2.3 Release Notes*
- *pgRouting 2.2.2 Release Notes*
- *pgRouting 2.2.1 Release Notes*
- *pgRouting 2.2.0 Release Notes*
- *pgRouting 2.1.0 Release Notes*
- *pgRouting 2.0 Release Notes*
- *pgRouting 1.x Release Notes*

7.1.1 pgRouting 2.2.4 Release Notes

With the release of pgRouting 2.2.4 fixes compatibility errors with Fedora

- To see the full list of changes check the list of Git commits[1] on Github.
- To see the issues closed by this release see the Git closed issues[2] on Github.
- For important changes see the following release notes.

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

Release Notes

Changes for release 2.2.4

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

7.1.2 pgRouting 2.2.3 Release Notes

With the release of pgRouting 2.2.e fixes compatibility issues with PostgreSQL 9.6.

- To see the full list of changes check the list of Git commits[3] on Github.
- To see the issues closed by this release see the Git closed issues[4] on Github.
- For important changes see the following release notes.

Release Notes

Changes for release 2.2.3

- Fixed compatibility issues with PostgreSQL 9.6.

7.1.3 pgRouting 2.2.2 Release Notes

With the release of pgRouting 2.2.2 fixes a regression bug.

- To see the full list of changes check the list of Git commits[5] on Github.
- To see the issues closed by this release see the Git closed issues[6] on Github.
- For important changes see the following release notes.

Release Notes

Changes for release 2.2.2

- Fixed regression error on pgr_drivingDistance

7.1.4 pgRouting 2.2.1 Release Notes

With the release of pgRouting 2.2.1 fixes some bugs and issues.

- To see the full list of changes check the list of Git commits[7] on Github.
- To see the issues closed by this release see the Git closed issues[8] on Github.
- For important changes see the following release notes.

[3] https://github.com/pgRouting/pgrouting/commits
[4] https://github.com/pgRouting/pgrouting/issues?q=milestone%3ARelease-2.2.3+is%3Aclosed
[5] https://github.com/pgRouting/pgrouting/commits
[6] https://github.com/pgRouting/pgrouting/issues?q=milestone%3ARelease-2.2.2+is%3Aclosed
[7] https://github.com/pgRouting/pgrouting/commits
[8] https://github.com/pgRouting/pgrouting/issues?q=milestone%3A2.2.1+is%3Aclosed

Release Notes

Changes for release 2.2.1

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

7.1.5 pgRouting 2.2.0 Release Notes

With the release of pgRouting 2.2.0 fixes some bugs and issues.

- To see the full list of changes check the list of Git commits[9] on Github.
- To see the issues closed by this release see the Git closed issues[10] on Github.
- For important changes see the following release notes.

Release Notes

Changes for release 2.2.0

- Improved:
  - pgr_nodeNetwork
  - Adding a row_where and outall optional parameters
- Signature fix
  - pgr_dijkstra – to match what was documented
- New functions
  - pgr_floydWarshall
  - pgr_Johnson
  - pgr_DijkstraCost
- New Proposed functions
  - pgr_withPoints
  - pgr_withPointsCost
  - pgr_withPointsDD
  - pgr_withPointsKSP
  - pgr_dijkstraVia
- Deprecated functions:
  - pgr_apspWarshall use pgr_floydWarshall instead
  - pgr_apspJohnson use pgr_Johnson instead
  - pgr_kDijkstraCost use pgr_dijkstraCost instead
  - pgr_kDijkstraPath use pgr_dijkstra instead

[9] https://github.com/pgRouting/pgrouting/commits
[10] https://github.com/pgRouting/pgrouting/issues?utf8=%E2%9C%93&q=is%3Aissue+milestone%3A%22Release+2.2.0%22+is%3Aclosed

7.1.6 pgRouting 2.1.0 Release Notes

With the release of pgRouting 2.1.0 fixes some bugs and issues.

- To see the full list of changes check the list of Git commits[11] on Github.
- To see the issues closed by this release see the Git closed issues[12] on Github.
- For important changes see the following release notes.

Release Notes

- 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
- Three Functions where completly re-factored using the developers library and functionality was added.
  - pgr_dijkstra
  - pgr_ksp
  - pgr_drivingDistance
- Improved - Alphashape function now can generate better (multi)polygon with holes and alpha parameter.
- Instead of generating many libraries: - All functions are encapsulated in one library - The library has a the prefix 2-1-0
- Added proposed functions to be evaluated:
  - Proposed functions from Steve Woodbridge, (Classified as Convinience 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
- Removed the 1.x legacy functions
- Some bug fixes in other functions

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

7.1.7 pgRouting 2.0 Release Notes

With the release of pgRouting 2.0 the library has abandoned backwards compatibility to *pgRouting 1.x* releases. We did this so we could restructure pgRouting, standardize the function naming, and prepare the project for future development. As a result of this effort, we have been able to simplify pgRouting, add significant new functionality, integrate documentation and testing into the source tree and make it easier for multiple developers to make contribution.

For important changes see the following release notes. To see the full list of changes check the list of Git commits[13] on Github.

Changes for 2.0.0

- Graph Analytics - tools for detecting and fixing connection some problems in a graph
- A collection of useful utility functions
- Two new All Pairs Short Path algorithms (pgr_apspJohnson, pgr_apspWarshall)
- Bi-directional Dijkstra and A-star search algorithms (pgr_bdAstar, pgr_bdDijkstra)
- One to many nodes search (pgr_kDijkstra)
- K alternate paths shortest path (pgr_ksp)
- New TSP solver that simplifies the code and the build process (pgr_tsp), dropped "Gaul Library" dependency
- Turn Restricted shortest path (pgr_trsp) that replaces Shooting Star
- Dropped support for Shooting Star
- Built a test infrastructure that is run before major code changes are checked in
- Tested and fixed most all of the outstanding bugs reported against 1.x that existing in the 2.0-dev code base.
- Improved build process for Windows
- Automated testing on Linux and Windows platforms trigger by every commit
- Modular library design
- Compatibility with PostgreSQL 9.1 or newer
- Compatibility with PostGIS 2.0 or newer
- Installs as PostgreSQL EXTENSION
- Return types 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

7.1.8 pgRouting 1.x Release Notes

The following release notes have been copied from the previous `RELEASE_NOTES` file and are kept as a reference. Release notes starting with *version 2.0.0* will follow a different schema.

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

Changes for release 1.05

- Bugfixes

Changes for release 1.03

- Much faster topology creation
- Bugfixes

Changes for release 1.02

- Shooting* bugfixes
- Compilation problems solved

Changes for release 1.01

- Shooting* bugfixes

Changes for release 1.0

- Core and extra functions are separated
- Cmake build process
- Bugfixes

Changes for release 1.0.0b

- Additional SQL file with more simple names for wrapper functions
- Bugfixes

Changes for release 1.0.0a

- Shooting* shortest path algorithm for real road networks
- Several SQL bugs were fixed

Changes for release 0.9.9

- PostgreSQL 8.2 support
- Shortest path functions return empty result if they couldn't find any path

Changes for release 0.9.8

- Renumbering scheme was added to shortest path functions
- Directed shortest path functions were added
- routing_postgis.sql was modified to use dijkstra in TSP search

Indices and tables

- genindex
- search