pgRouting Manual

Release 2.0.0 (d4d49b7 master)

pgRouting Contributors

September 20, 2013

Contents

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

This is the manual for pgRouting 2.0.0 (d4d49b7 master).

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

CHAPTER 1

General

1.1 Introduction

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

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

1.1.1 License

The following licenses can be found in pgRouting:

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

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

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

1.1.2 Contributors

Individuals (in alphabetical order)

Akio Takubo, Anton Patrushev, Ashraf Hossain, Christian Gonzalez, Daniel Kastl, Dave Potts, David Techer, Ema Miyawaki, Florian Thurkow, Frederic Junod, Gerald Fenoy, Jay Mahadeokar, Jinfu Leng, Kai Behncke, Kishore Kumar, Ko Nagase, Mario Basa, Martin Wiesenhaan, Razequl Islam, 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

1.1.3 More Information

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

1.2 Installation

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

1.2.1 Windows

Winnie Bot Experimental Builds:

- PostgreSQL 9.2 32-bit, 64-bit[13]

1.2.2 Ubuntu/Debian

Ubuntu packages are available in Launchpad repositories:

- *stable* https://launchpad.net/~georepublic/+archive/pgrouting
- *unstable* https://launchpad.net/~georepublic/+archive/pgrouting-unstable

```
# Add pgRouting launchpad repository ("stable" or "unstable")
sudo add-apt-repository ppa:georepublic/pgrouting[-unstable]
sudo apt-get update

# Install pgRouting packages
sudo apt-get install postgresql-9.1-pgrouting
```

Use UbuntuGIS-unstable PPA[14] to install PostGIS 2.0.

[13] http://winnie.postgis.net/download/windows/pg92/buildbot/
[14] https://launchpad.net/ ubuntugis/+archive/ubuntugis-unstable

1.2.3 RHEL/CentOS/Fedora

- Fedora RPM's: https://admin.fedoraproject.org/pkgdb/acls/name/pgRouting

1.2.4 OS X

- Homebrew

```
brew install pgrouting
```

1.2.5 Source Package

Git 2.0.0-rc1 release	v2.0.0-rc1.tar.gz[15]	v2.0.0-rc1.zip[16]
Git 2.0.0-beta release	v2.0.0-beta.tar.gz[17]	v2.0.0-beta.zip[18]
Git 2.0.0-alpha release	v2.0.0-alpha.tar.gz[19]	v2.0.0-alpha.zip[20]
Git master branch	master.tar.gz[21]	master.zip[22]
Git develop branch	develop.tar.gz[23]	develop.zip[24]

1.2.6 Using Git

Git protocol (read-only):

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

HTTPS protocol (read-only): .. code-block:: bash

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

See *Build Guide* for notes on compiling from source.

1.3 Build Guide

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

- C and C++ compilers
- Postgresql version >= 8.4 (>= 9.1 recommended)
- PostGIS version >= 1.5 (>= 2.0 recommended)

[15] https://github.com/pgRouting/pgrouting/archive/v2.0.0-rc1.tar.gz
[16] https://github.com/pgRouting/pgrouting/archive/v2.0.0-rc1.zip
[17] https://github.com/pgRouting/pgrouting/archive/v2.0.0-beta.tar.gz
[18] https://github.com/pgRouting/pgrouting/archive/v2.0.0-beta.zip
[19] https://github.com/pgRouting/pgrouting/archive/v2.0.0-alpha.tar.gz
[20] https://github.com/pgRouting/pgrouting/archive/v2.0.0-alpha.zip
[21] https://github.com/pgRouting/pgrouting/archive/master.tar.gz
[22] https://github.com/pgRouting/pgrouting/archive/master.zip
[23] https://github.com/pgRouting/pgrouting/archive/develop.tar.gz
[24] https://github.com/pgRouting/pgrouting/archive/develop.zip

- The Boost Graph Library (BGL). Version >= [TBD]
- CMake >= 2.8.8
- (optional, for Driving Distance) CGAL >= [TBD]
- (optional, for Documentation) Sphinx >= 1.1
- (optional, for Documentation as PDF) Latex >= [TBD]

The cmake system has variables the can be configured via the command line options by setting them with -D<varable>=<value>. You can get a listing of these via:

```
mkdir build
cd build
cmake -L ..
```

Currently these are:

> Boost_DIR:PATH=Boost_DIR-NOTFOUND CMAKE_BUILD_TYPE:STRING=
> CMAKE_INSTALL_PREFIX:PATH=/usr/local POSTGRESQL_EXECUTABLE:FILEPATH=/usr/lib/postgresql/9.2/bin/post
> POSTGRESQL_PG_CONFIG:FILEPATH=/usr/bin/pg_config WITH_DD:BOOL=ON
> WITH_DOC:BOOL=OFF BUILD_HTML:BOOL=ON BUILD_LATEX:BOOL=OFF
> BUILD_MAN:BOOL=ON

These also show the current or default values based on our development system. So your values my be different. In general the ones that are of most interest are:

> WITH_DD:BOOL=ON – Turn on/off building driving distance code. 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=ON – If WITH_DOC=ON, turn on/off building MAN pages

To change any of these add `-D<variable>=<value>` to the cmake lines below. For example to turn on documentation, your cmake command might look like:

```
cmake -DWITH_DOC=ON ..    # Turn on the doc with default settings
cmake -DWITH_DOC=ON -DBUILD_LATEX ..  # Turn on doc and pdf
```

If you turn on the documentation, you also need to add the `doc` target to the make command.

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

1.3.1 For MinGW on Windows

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

1.3.2 For Linux

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

1.3.3 With Documentation

Build with documentation (requires Sphinx[25]):

```
    cmake -DWITH_DOC=ON ..
make all doc
```

Rebuild modified documentation only:

```
sphinx-build -b html -c build/doc/_build -d build/doc/_doctrees . build/html
```

1.4 Support

pgRouting community support is available through website[26], documentation[27], 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[28]. Please follow these steps:

1. Search the tickets to see if your problem has already been reported. If so, add any extra context you might have found, or at least indicate that you too are having the problem. This will help us prioritize common issues.
2. If your problem is unreported, create a new issue[29] 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.

[25] http://sphinx-doc.org/
[26] http://www.pgrouting.org
[27] http://docs.pgrouting.org
[28] https://github.com/pgrouting/pgrouting/issues
[29] https://github.com/pgRouting/pgrouting/issues/new

You can also ask at GIS StackExchange[30] 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[31].

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
Orkney Inc.	Japan	http://www.orkney.co.jp
Camptocamp	Switzerland, France	http://www.camptocamp.com

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

CHAPTER 2

Tutorial

2.1 Tutorial

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 to use for our project

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"
```

For older versions of postgresql

```
createdb -T template1 template_postgis
psql template_postgis -c "create language plpgsql"
psql template_postgis -f /usr/share/postgresql/9.0/contrib/postgis-1.5/postgis.sql
psql template_postgis -f /usr/share/postgresql/9.0/contrib/postgis-1.5/spatial_ref_sys.sql
psql template_postgis -f /usr/share/postgresql/9.0/contrib/postgis_comments.sql

createdb -T template_postgis template_pgrouting
psql template_pgrouting -f /usr/share/postgresql/9.0/contrib/pgrouting-2.0/pgrouting.sql

createdb -T template_pgrouting mydatabase
```

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:

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 and and can 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 you 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 tools the will help with this:

```
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 as some very specific requirments. One it that it is *NODED*, this means that except for some very specific use cases, each road segments 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 prep work done above, computing a route is fairly easy. We have a lot of different algorithms but they 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 on how to use them. What you get as a result from these queries will be a set of record of type *pgr_costResult[]* or *pgr_geomResult[]*. 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.

- See also *pgr_costResult[]* and *pgr_geomResult[]*.

How to use other tools to view your graph and route

TBD

How to create a web app

TBD

2.1.2 Routing Topology

Author Stephen Woodbridge <woodbri@swoodbridge.com[1]>

Overview

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

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

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

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

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

[1] woodbri@swoodbridge.com

```
    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[2]>

Overview

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

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

Analyze a Graph

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

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

In the vertices table "mytab_vertices_pgr":

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

[2] woodbri@swoodbridge.com

```
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 Custom Query

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

When the edge table has the mentioned columns, the following SQL queries can be used.

```
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 requiered
pgr_func('edge_table',0.001)   -- when tolerance is requiered
-- s_in_rule, s_out_rule, st_in_rules, t_out_rules are requiered
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

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.

2.1.6 User's wrapper contributions

How to contribute.

Use an issue tracker (see *Support*) with a title containing: *Proposing a wrapper: Mywrappername*. The body will contain:

- author: Required
- mail: if you are subscribed to the developers list this is not necessary
- date: Date posted
- comments and code: using reStructuredText format

Any contact with the author will be done using the developers mailing list. The pgRouting team will evaluate the wrapper and will be included it in this section when approved.

No contributions at this time

2.1.7 Use's Recipes contributions

How to contribute.

Use an issue tracker (see *Support*) with a title containing: *Proposing a Recipe: Myrecipename*. The body will contain:

- author: Required
- mail: if you are subscribed to the developers list this is not necessary
- date: Date posted
- comments and code: using reStructuredText format

Any contact with the author will be done using the developers mailing list. The pgRouting team will evaluate the recipe and will be included it in this section when approved.

Comparing topology of a unnoded network with a noded network

Author pgRouting team.

This recipe uses the *Sample Data* 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);
```

No more contributions

2.2 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 serial,
    dir character varying,
    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 network 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
                                 WHEN (cost<0 and reverse_cost>0) THEN 'TF'   -- reverse direction
                                 ELSE '' END;                                -- unknown
```

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

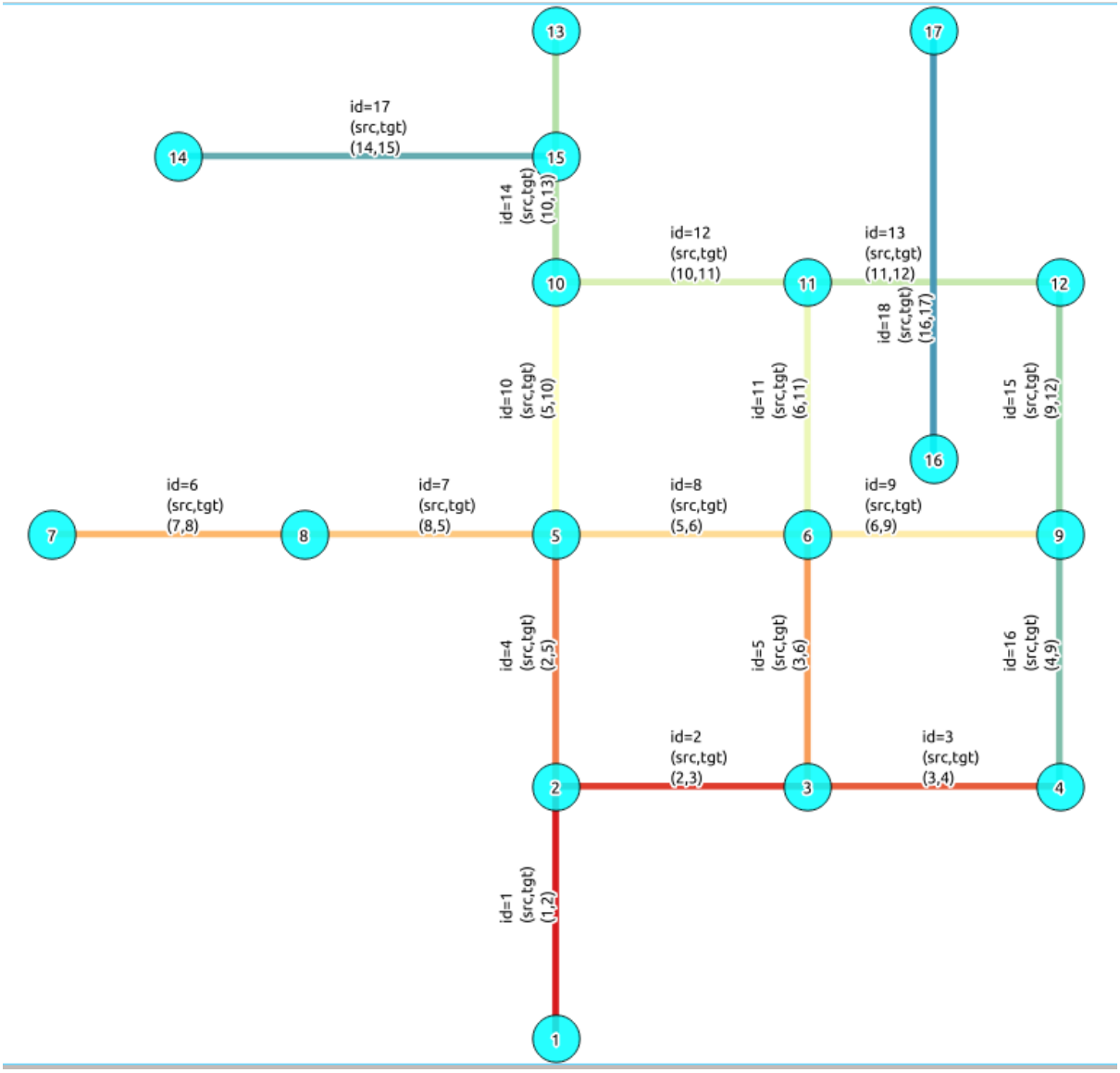
```
SELECT pgr_createTopology('edge_table',0.001);
```

This table is used in some of our examples

```
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);
```

The network created in *edge_table*



For a more complete introduction how to build a routing application read the pgRouting Workshop[3].

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

CHAPTER 3

Data Types

3.1 pgRouting Data Types

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

3.1.1 pgr_costResult[]

Name

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

Description

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

seq sequential ID indicating the path order

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

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

cost cost attribute

3.1.2 pgr_costResult3[] - Multiple Path Results with Cost

Name

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

Description

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

seq sequential ID indicating the path order

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

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

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

cost cost attribute

History

- New in version 2.0.0
- Replaces `path_result`

See Also

- *Introduction*

3.1.3 pgr_geomResult[]

Name

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

Description

```
CREATE TYPE pgr_geomResult AS
(
    seq integer,
    id1 integer,
    id2 integer,
    geom geometry
);
```

seq sequential ID indicating the path order

id1 generic name, to be specified by the function

id2 generic name, to be specified by the function

geom geometry attribute

History

- New in version 2.0.0
- Replaces `geoms`

See Also

- *Introduction*

Functions reference

4.1 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.

4.1.1 pgr_createTopology

Name

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

Synopsis

The function returns:

- `OK` after the network topology has been built and the vertices table created.
- `FAIL` when the network topology was not built due to an error.

```
varchar pgr_createTopology(text edge_table, double precision tolerance,
                  text the_geom:='the_geom', text id:='id',
                  text source:='source',text target:='target',text rows_where:='true')
```

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.

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);
```

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

```
SELECT  pgr_createTopology('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 `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','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_createTopology('edge_table',0.001,the_geom:='the_geom',id:='id',source:='source',targe
```

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

Parameters defined with a default value can be omited, AS long AS the value matches the default:

```
SELECT  pgr_createTopology('edge_table',0.001,source:='source');
```

Selecting rows using rows_where parameter

Selecting rows based on the id.

```
SELECT  pgr_createTopology('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_createTopology('edge_table',0.001,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_createTopology('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(other_geo
```

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

For the following table

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

Using positional notation:

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

```
SELECT  pgr_createTopology('mytable',0.001,'mygeom','gid','src','tgt');
```

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');
```

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

```
SELECT  pgr_createTopology('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_createTopology('mytable',0.001,'mygeom','gid','src','tgt',rows_where:='gid < 10');
```

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

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

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

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

Examples

```
SELECT pgr_createTopology('edge_table', 0.001,rows_where:='id<10');
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.0001,'the_geom','id','source','target','id<10')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  --------------> TOPOLOGY CREATED FOR  9 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr

 pgr_createtopology
--------------------
 OK
```

```
(1 row)

SELECT pgr_createTopology('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.0001,'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

 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.

4.1.2 pgr_createVerticesTable

Name

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

Synopsis

The function returns:

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

```
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_geo
```

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 ed
```

Using positional notation:

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

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

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

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

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.

4.1.3 pgr_analyzeGraph

Name

`pgr_anlizeGraph` — 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_create_topology('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_create_topology('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_analizeGraph
--------------------
 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_analizeGraph
    -------------------
     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_analizeGraph
    -------------------
     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_analizeGraph
    -------------------
     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_analizeGraph
    -------------------
     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_analizeGraph
    -------------------
     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.

4.1.4 pgr_analyzeOneway

Name

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

Synopsis

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

```
text pgr_analyzeOneway(geom_table text,
                       text[] s_in_rules, text[] s_out_rules,
                       text[] t_in_rules, text[] t_out_rules,
                       text oneway='oneway', text source='source', text target='target',
                       boolean two_way_if_null=true);
```

Description

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

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

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

Prerequisites

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

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

Parameters

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

s_in_rules `text[]` source node **in** rules

s_out_rules `text[]` source node **out** rules

t_in_rules `text[]` target node **in** rules

t_out_rules `text[]` target node **out** rules

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

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

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

two_way_if_null `boolean` flag to treat oneway NULL values as bi-directional. Default value is `true`.

Note: It is strongly recommended to use the named notation. See *pgr_createVerticesTable* or *pgr_createTopology* for examples.

The function returns:

- `OK` after the analysis has finished.
  - Uses the vertices table: <edge_table>_vertices_pgr.
  - Fills completely the `ein` and `eout` columns of the vertices table.
- `FAIL` when the analysis was not completed due to an error.
  - The vertices table is not found.
  - A required column of the Network table is not found or is not of the appropriate type.

- The names of source , target or oneway are the same.

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

The Vertices Table

The vertices table can be created with *pgr_createVerticesTable* or *pgr_createTopology*

The structure of the vertices table is:

id `bigint` Identifier of the vertex.

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

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

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

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

the_geom `geometry` Point geometry of the vertex.

History

- New in version 2.0.0

Examples

```
SELECT pgr_analyzeOneway('edge_table',
ARRAY['', 'B', 'TF'],
ARRAY['', 'B', 'FT'],
ARRAY['', 'B', 'FT'],
ARRAY['', 'B', 'TF'],
oneway:='dir');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table','{"",B,TF}','{"",B,FT}','{"",B,FT}','{"",B,TF}','dir','sou
NOTICE:  Analyzing graph for one way street errors.
NOTICE:  Analysis 25% complete ...
NOTICE:  Analysis 50% complete ...
NOTICE:  Analysis 75% complete ...
NOTICE:  Analysis 100% complete ...
NOTICE:  Found 0 potential problems in directionality

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

The queries use the *Sample Data* network.

See Also

- *Routing Topology* for an overview of a topology for routing algorithms.
- *Graph Analytics* for an overview of the analysis of a graph.
- *pgr_analyzeGraph* to analyze the edges and vertices of the edge table.
- *pgr_createVerticesTable* to reconstruct the vertices table based on the source and target information.

4.1.5 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')
```

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

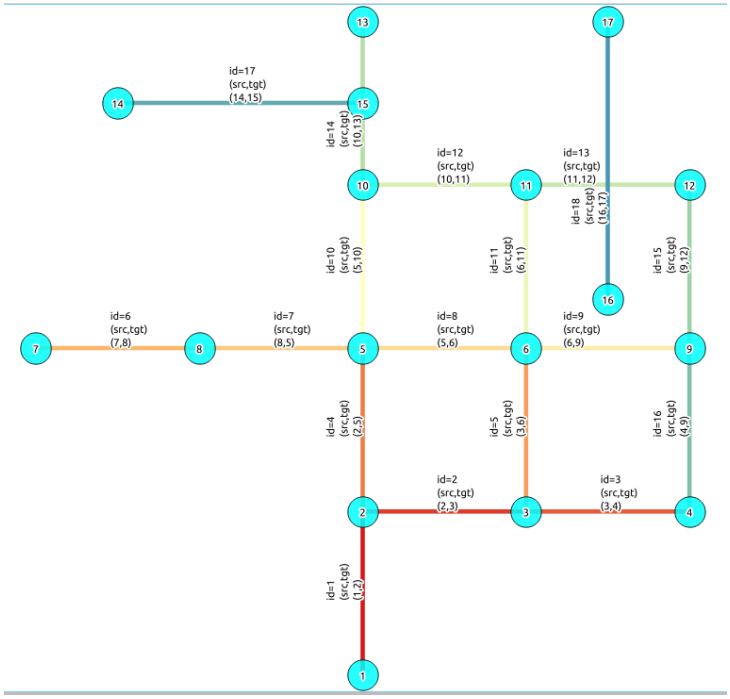
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  --------------> TOPOLOGY CREATED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

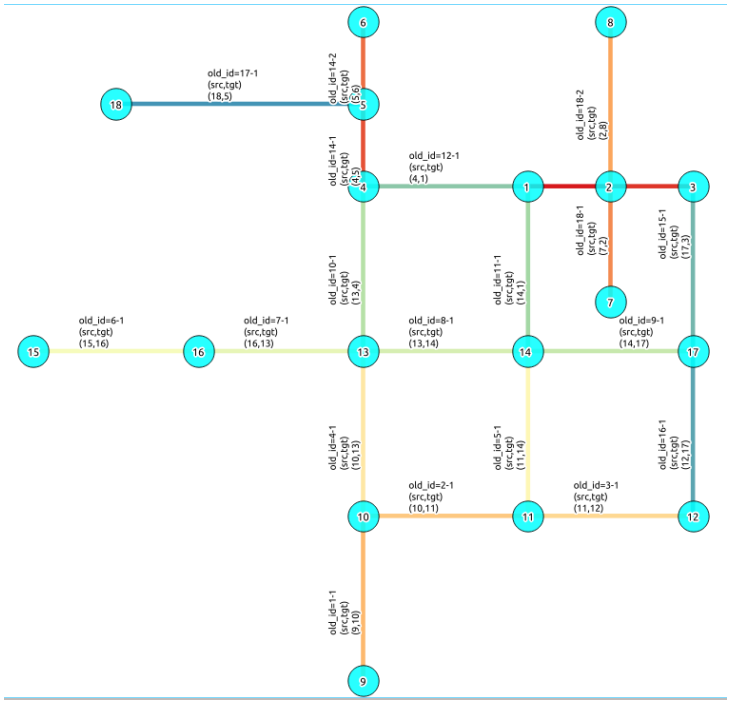
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...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

4.2 Routing Functions

4.2.1 pgr_apspJohnson - All Pairs Shortest Path, Johnson's Algorithm

Name

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

Synopsis

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

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

Description

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

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

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

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

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

Returns set of *pgr_costResult[]*:

seq row sequence

id1 source node ID

id2 target node ID

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

History

- New in version 2.0.0

Examples

```
SELECT seq, id1 AS from, id2 AS to, cost
    FROM pgr_apspJohnson(
        'SELECT source, target, cost FROM edge_table'
    );
```

```
 seq | from | to | cost
-----+------+----+------
   0 |    1 |  1 |    0
   1 |    1 |  2 |    1
   2 |    1 |  5 |    2
   3 |    1 |  6 |    3
[...]
```

The query uses the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *pgr_apspWarshall - All Pairs Shortest Path, Floyd-Warshall Algorithm*
- http://en.wikipedia.org/wiki/Johnson%27s_algorithm

4.2.2 pgr_apspWarshall - All Pairs Shortest Path, Floyd-Warshall Algorithm

Name

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

Synopsis

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

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

Description

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

```
SELECT 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

directed `true` if the graph is directed

reverse_cost 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

- New in version 2.0.0

Examples

```
SELECT seq, id1 AS from, id2 AS to, cost
    FROM pgr_apspWarshall(
        'SELECT id, source, target, cost FROM edge_table',
        false, false
    );
```

```
 seq | from | to | cost
-----+------+----+------
   0 |    1 |  1 |    0
   1 |    1 |  2 |    1
   2 |    1 |  3 |    0
   3 |    1 |  4 |   -1
[...]
```

The query uses the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *pgr_apspJohnson - All Pairs Shortest Path, Johnson's Algorithm*
- http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

4.2.3 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:

```
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 seq, id1 AS node, id2 AS edge, cost
        FROM pgr_astar(
                'SELECT id, source, target, cost, x1, y1, x2, y2 FROM edge_table',
                4, 1, false, false
        );
```

```
 seq | node | edge | 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

(4 rows)
```

- With `reverse_cost`

```
SELECT seq, id1 AS node, id2 AS edge, cost
        FROM pgr_astar(
                'SELECT id, source, target, cost, x1, y1, x2, y2, reverse_cost FROM edge_table',
                4, 1, true, true
        );

 seq | node | edge | 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

4.2.4 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 seq, id1 AS node, id2 AS edge, cost
    FROM pgr_bdAstar(
        'SELECT id, source, target, cost, x1, y1, x2, y2 FROM edge_table',
        4, 10, false, false
    );
```

```
 seq | node | edge | 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 seq, id1 AS node, id2 AS edge, cost
    FROM pgr_bdAstar(
        'SELECT id, source, target, cost, x1, y1, x2, y2, reverse_cost FROM edge_table',
        4, 10, true, true
    );
```

```
 seq | node | edge | 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

4.2.5 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 seq, id1 AS node, id2 AS edge, cost
    FROM pgr_bdDijkstra(
        'SELECT id, source, target, cost FROM edge_table',
        4, 10, false, false
    );

 seq | node | edge | 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 seq, id1 AS node, id2 AS edge, cost
    FROM pgr_bdDijkstra(
        'SELECT id, source, target, cost, reverse_cost FROM edge_table',
        4, 10, true, true
    );

 seq | node | edge | 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

4.2.6 pgr_dijkstra - 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);
```

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

- Without `reverse_cost`

```
SELECT seq, id1 AS node, id2 AS edge, cost
        FROM pgr_dijkstra(
                'SELECT id, source, target, cost FROM edge_table',
                7, 12, false, false
        );
```

```
 seq | node | edge | cost
-----+------+------+------
   0 |    7 |    8 |    1
```

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

- With `reverse_cost`

```
SELECT seq, id1 AS node, id2 AS edge, cost
        FROM pgr_dijkstra(
                'SELECT id, source, target, cost, reverse_cost FROM edge_table',
                7, 12, true, true
        );

 seq | node | edge | cost
-----+------+------+------
   0 |    7 |    8 |    1
   1 |    8 |    9 |    1
   2 |    9 |   15 |    1
   3 |   12 |   -1 |    0
(4 rows)
```

The queries use the *Sample Data* network.

See Also

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

4.2.7 pgr_kDijkstra - Mutliple destination Shortest Path Dijkstra

Name

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

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_costResult3* 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

- New in version 2.0.0

Examples

- Returning a `cost` result

```
SELECT seq, id1 AS source, id2 AS target, cost FROM pgr_kdijkstraCost(
    'SELECT id, source, target, cost FROM edge_table',
    10, array[4,12], false, false
);

 seq | source | target | cost
-----+--------+--------+------
   0 |     10 |      4 |    4
   1 |     10 |     12 |    2

SELECT seq, id1 AS path, id2 AS node, id3 AS edge, cost
  FROM pgr_kdijkstraPath(
      'SELECT id, source, target, cost FROM edge_table',
      10, array[4,12], false, false
);
```

```
 seq | path | node | edge | 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, source, target, 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;

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

4.2.8 pgr_ksp - K-Shortest Path

Name

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

Synopsis

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

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

Description

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

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

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

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

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

source `int4` id of the start point

target `int4` id of the end point

paths `int4` number of alternative routes

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

Returns set of *pgr_costResult[]*:

seq sequence for ording the results

id1 route ID

id2 node ID

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

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

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

History

- New in version 2.0.0

Examples

- Without `reverse_cost`

```
SELECT seq, id1 AS route, id2 AS node, id3 AS edge, cost
 FROM pgr_ksp(
   'SELECT id, source, target, cost FROM edge_table',
   7, 12, 2, false
 );

 seq | route | node | edge | cost
-----+-------+------+------+------
   0 |     0 |    7 |    6 |    1
   1 |     0 |    8 |    7 |    1
   2 |     0 |    5 |    8 |    1
   3 |     0 |    6 |   11 |    1
   4 |     0 |   11 |   13 |    1
   5 |     0 |   12 |    0 |    0
   6 |     1 |    7 |    6 |    1
   7 |     1 |    8 |    7 |    1
   8 |     1 |    5 |    8 |    1
   9 |     1 |    6 |    9 |    1
  10 |     1 |    9 |   15 |    1
```

```
  11 |     1 |   12 |    0 |    0
(12 rows)
```

- With `reverse_cost`

```
SELECT seq, id1 AS route, id2 AS node, id3 AS edge, cost
 FROM pgr_ksp(
   'SELECT id, source, target, cost, reverse_cost FROM edge_table',
   7, 12, 2, true
 );

 seq | route | node | edge | cost
-----+-------+------+------+------
   0 |     0 |    7 |    6 |    1
   1 |     0 |    8 |    7 |    1
   2 |     0 |    5 |    8 |    1
   3 |     0 |    6 |   11 |    1
   4 |     0 |   11 |   13 |    1
   5 |     0 |   12 |    0 |    0
   6 |     1 |    7 |    6 |    1
   7 |     1 |    8 |    7 |    1
   8 |     1 |    5 |    8 |    1
   9 |     1 |    6 |    9 |    1
  10 |     1 |    9 |   15 |    1
  11 |     1 |   12 |    0 |    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

4.2.9 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.

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

[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.

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[2] 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.

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

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

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

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

- Using distance matrix (A loop starting from 1)

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

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

```
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

4.2.10 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 double precision,
                          target_edge integer, target_pos double precision, directed boolean,
                     has_rcost boolean [,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 makes adding turn restrictions to a road network much easier than trying to add them to Shooting Star where you had to ad the same edges multiple times if it was involved in a restriction.

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

Examples

- Without turn restrictions

```
SELECT seq, id1 AS node, id2 AS edge, cost
        FROM pgr_trsp(
                'SELECT id, source, target, cost FROM edge_table',
                7, 12, false, false
        );
```

```
seq | node | edge | cost
----+------+------+------
  0 |    7 |    6 |    1
  1 |    8 |    7 |    1
  2 |    5 |    8 |    1
  3 |    6 |   11 |    1
  4 |   11 |   13 |    1
  5 |   12 |   -1 |    0
(6 rows)
```

- With turn restrictions

Turn restrictions require additional information, which can be stored in a separate table:

```
CREATE TABLE restrictions (
    rid serial,
    to_cost double precision,
    to_edge integer,
    from_edge integer,
    via text
);

INSERT INTO restrictions VALUES (1,100,7,4,null);
INSERT INTO restrictions VALUES (2,4,8,3,5);
INSERT INTO restrictions VALUES (3,100,9,16,null);
```

Then a query with turn restrictions is created as:

```
SELECT seq, id1 AS node, id2 AS edge, cost
        FROM pgr_trsp(
                'SELECT id, source, target, cost FROM edge_table',
                7, 12, false, false,
                'SELECT to_cost, to_edge AS target_id,
           from_edge || coalesce('','' || via, '''') AS via_path
       FROM restrictions'
        );
```

```
 seq | node | edge | cost
-----+------+------+------
   0 |    7 |    6 |    1
   1 |    8 |    7 |    1
   2 |    5 |    8 |    1
   3 |    6 |   11 |    1
   4 |   11 |   13 |    1
   5 |   12 |   -1 |    0
(6 rows)
```

The queries use the *Sample Data* network.

See Also

- *pgr_costResult[]*
- *genindex*
- *search*

4.3 With Driving Distance Enabled

Driving distance related Functions

4.3.1 pgr_drivingDistance

Name

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

Note: Requires *to build pgRouting* with support for Driving Distance.

Synopsis

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

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

Description

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

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

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

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

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

source `int4` id of the start point

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

directed `true` if the graph is directed

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

Returns set of *pgr_costResult[]*:

seq row sequence

id1 node ID

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

cost cost to get to this node ID

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

History

- Renamed in version 2.0.0

Examples

- Without `reverse_cost`

```
SELECT seq, id1 AS node, cost
        FROM pgr_drivingDistance(
                'SELECT id, source, target, cost FROM edge_table',
                7, 1.5, false, false
        );
```

```
 seq | node | cost
-----+------+------
   0 |    2 |    1
   1 |    6 |    1
   2 |    7 |    0
   3 |    8 |    1
   4 |   10 |    1
(5 rows)
```

- With `reverse_cost`

```
SELECT seq, id1 AS node, cost
        FROM pgr_drivingDistance(
                'SELECT id, source, target, cost, reverse_cost FROM edge_table',
                7, 1.5, true, true
        );

 seq | node | cost
-----+------+------
   0 |    2 |    1
   1 |    6 |    1
   2 |    7 |    0
   3 |    8 |    1
   4 |   10 |    1
(5 rows)
```

The queries use the *Sample Data* network.

See Also

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

4.3.2 pgr_alphaShape

Name

`pgr_alphashape` — Core function for alpha shape computation.

Note: This function should not be used directly. Use *pgr_drivingDistance* instead.

Synopsis

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

```
table() pgr_alphashape(text sql);
```

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

Returns a vertex record for each row :

x x-coordinate

y y-coordinate

History

- Renamed in version 2.0.0

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 | 0
 4 | 1
 4 | 2
 4 | 3
 2 | 4
 0 | 2
(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
------+-----
 0.5  | 3.5
   0  |   2
   2  |   0
   4  |   1
   4  |   2
   4  |   3
 3.5  |   4
   2  |   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
) as a) as foo;

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

The queries use the *Sample Data* network.

See Also

- *pgr_drivingDistance* - Driving Distance

- *pgr_pointsAsPolygon* - Polygon around set of points

4.3.3 pgr_pointsAsPolygon

Name

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

Synopsis

Returns the alpha shape as polygon geometry.

```
geometry pgr_pointsAsPolygon(text sql);
```

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

Returns a polygon geometry.

History

- Renamed in version 2.0.0

Examples

In the following query there is not 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. Each of the pgr_pointsAsPolygon queries below is followed by one the compute the area of the polygon. This area should remain constant regardles of the order of the points making up the polygon.

```
SELECT ST_AsText(pgr_pointsAsPolygon('SELECT id, x, y FROM vertex_table'));

               st_astext
----------------------------------------
 POLYGON((2 0,4 1,4 2,4 3,2 4,0 2,2 0))
(1 row)

SELECT round(ST_Area(pgr_pointsAsPolygon('SELECT id, x, y FROM vertex_table'))::numeric, 2) as st_

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


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((0.5 3.5,0 2,2 0,4 1,4 2,4 3,3.5 4,2 4,0.5 3.5))
(1 row)

SELECT round(ST_Area(pgr_pointsASPolygon('SELECT id::integer, st_x(the_geom)::float as x, st_y(the
     FROM edge_table_vertices_pgr'))::numeric, 2) as st_area;

 st_area
---------
   11.75
```

The queries use the *Sample Data* network.

See Also

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

Indices and tables

- *genindex*
- *search*

4.4 Developers's Functions

4.4.1 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.

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.

4.4.2 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.

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.

4.4.3 pgr_isColumnIndexed

Name

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

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

Synopsis

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

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

Description

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

col `text` Column name to be checked for.

Returns:

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

History

- New in version 2.0.0

Examples

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

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

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

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

The example use the *Sample Data* network.

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_isColumnInTable* 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.

4.4.4 pgr_isColumnInTable

Name

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

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

Synopsis

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

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

Description

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

col `text` Column name to be checked for.

Returns:

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

History

- New in version 2.0.0

Examples

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

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

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

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

The example use the *Sample Data* network.

See Also

- *Developer's Guide* for the tree layout of the project.
- *pgr_isColumnIndexed* 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.

4.4.5 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.

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.

4.4.6 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.

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.

4.4.7 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
-------------------------------------------------------
 (2.0.0-dev,v2.0dev,290,c64bcb9,sew-devel-2_0,1.49.0)
(1 row)
```

- Query for `version` and `boost` attribute

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

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

See Also

- *pgr_versionless* to compare two version numbers

4.4.8 pgr_versionless

Name

`pgr_versionless` — Compare two version numbers.

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

Synopsis

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

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

Description

v1 `text` first version number

v2 `text` second version number

History

- New in version 2.0.0

Examples

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

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

See Also

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

4.4.9 pgr_startPoint

Name

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

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

Synopsis

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

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

Description

Parameters

geom `geometry` Geometry of a MULTILINESTRING or LINESTRING.

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

History

- New in version 2.0.0

See Also

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

4.4.10 pgr_endPoint

Name

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

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

Synopsis

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

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

Description

Parameters

geom `geometry` Geometry of a MULTILINESTRING or LINESTRING.

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

History

- New in version 2.0.0

See Also

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

4.5 Legacy Functions

pgRouting 2.0 release has total restructured the function naming and obsoleted many of the functions that were available in the 1.x releases. While we realize that this may inconvenience our existing users, we felt this was needed for the long term viability of the project to be more response to our users and to be able to add new functionality and test existing functionality.

We have made a minimal effort to save most of these function and distribute with the release in a file `pgrouting_legacy.sql` that is not part of the pgrouting extension and is not supported. If you can use these functions that is great. We have not tested any of these functions so if you find issues and want to post a pull request or a patch to help other users that is fine, but it is likely this file will be removed in a future release and we strongly recommend that you convert your existing code to use the new documented and supported functions.

The follow is a list of TYPEs, CASTs and FUNCTION included in the `pgrouting_legacy.sql` file. The list is provide as a convenience but these functions are deprecated, not supported, and probably will need some changes to get them to work.

4.5.1 TYPEs & CASTs

```
TYPE vertex_result AS ( x float8, y float8 ):
CAST (pgr_pathResult AS path_result) WITHOUT FUNCTION AS IMPLICIT;
CAST (pgr_geoms AS geoms) WITHOUT FUNCTION AS IMPLICIT;
CAST (pgr_linkPoint AS link_point) WITHOUT FUNCTION AS IMPLICIT;
```

4.5.2 FUNCTIONs

```
FUNCTION text(boolean)
FUNCTION add_vertices_geometry(geom_table varchar)
FUNCTION update_cost_from_distance(geom_table varchar)
FUNCTION insert_vertex(vertices_table varchar, geom_id anyelement)
FUNCTION pgr_shootingStar(sql text, source_id integer, target_id integer,
                    directed boolean, has_reverse_cost boolean)
FUNCTION shootingstar_sp( varchar,int4, int4, float8, varchar, boolean, boolean)
FUNCTION astar_sp_delta( varchar,int4, int4, float8)
FUNCTION astar_sp_delta_directed( varchar,int4, int4, float8, boolean, boolean)
FUNCTION astar_sp_delta_cc( varchar,int4, int4, float8, varchar)
FUNCTION astar_sp_delta_cc_directed( varchar,int4, int4, float8, varchar, boolean, boolean)
FUNCTION astar_sp_bbox( varchar,int4, int4, float8, float8, float8, float8)
FUNCTION astar_sp_bbox_directed( varchar,int4, int4, float8, float8, float8,
                            float8, boolean, boolean)
FUNCTION astar_sp( geom_table varchar, source int4, target int4)
FUNCTION astar_sp_directed( geom_table varchar, source int4, target int4,
                        dir boolean, rc boolean)
FUNCTION dijkstra_sp( geom_table varchar, source int4, target int4)
FUNCTION dijkstra_sp_directed( geom_table varchar, source int4, target int4,
                           dir boolean, rc boolean)
FUNCTION dijkstra_sp_delta( varchar,int4, int4, float8)
FUNCTION dijkstra_sp_delta_directed( varchar,int4, int4, float8, boolean, boolean)
FUNCTION tsp_astar( geom_table varchar,ids varchar, source integer, delta double precision)
FUNCTION tsp_astar_directed( geom_table varchar,ids varchar, source integer, delta float8, dir boo
FUNCTION tsp_dijkstra( geom_table varchar,ids varchar, source integer)
FUNCTION tsp_dijkstra_directed( geom_table varchar,ids varchar, source integer,
                           delta float8, dir boolean, rc boolean)
```

4.6 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.

4.6.1 Shooting Star algorithm

Version Removed with 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. The function signature is still available in *Legacy Functions* but it is just a wrapper that throws an error. We have not included any of the old code for this in this release.

CHAPTER 5

Developer

5.1 Developer's Guide

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

5.1.1 Source Tree Layout

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

core/ This is the algorithm source tree. Each algorithm should be contained in its on sub-tree with doc, sql, src, and test sub-directories. This might get renamed to "algorithms" at some point.

core/astar/ This is an implementation of A* Search based on using Boost Graph libraries for its implementation. This is a Dijkstra shortest path implementation with a Euclidean Heuristic.

core/common/ At the moment this does not have an core in "src", but does have a lot of SQL wrapper code and topology code in the "sql" directory. *Algorithm specific wrappers should get move to the algorithm tree and appropriate tests should get added to validate the wrappers.*

core/dijkstra/ This is an implementation of Dikjstra's shortest path solution using Boost Graph libraries for the implementation.

core/driving_distance/ This optional package creates driving distance polygons based on solving a Dijkstra shortest path solution, then creating polygons based on equal cost distances from the start point. This optional package requires CGAL libraries to be installed.

core/shooting_star/ *DEPRECATED and DOES NOT WORK and IS BEING REMOVED* This is an edge based shortest path algorithm that supports turn restrictions. It is based on Boost Graph. Do *NOT* use this algorithm as it is broken, instead use *trsp* which has the same functionality and is faster and give correct results.

core/trsp/ This is a turn restricted shortest path algorithm. It has some nice features like you can specify the start and end points as a percentage along an edge. Restrictions are stored in a separate table from the graph edges and this make it easier to manage the data.

core/tsp/ This optional package provides the ability to compute traveling salesman problem solutions and compute the resulting route. This optional package requires GAUL libaries to be installed.

tools/ Miscellaneous scripts and tools.

lib/ This is the output directory where compiled libraries and installation targets are staged before installation.

5.1.2 Documentation Layout

As noted above all documentation should be done using reStructuredText formated files.

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

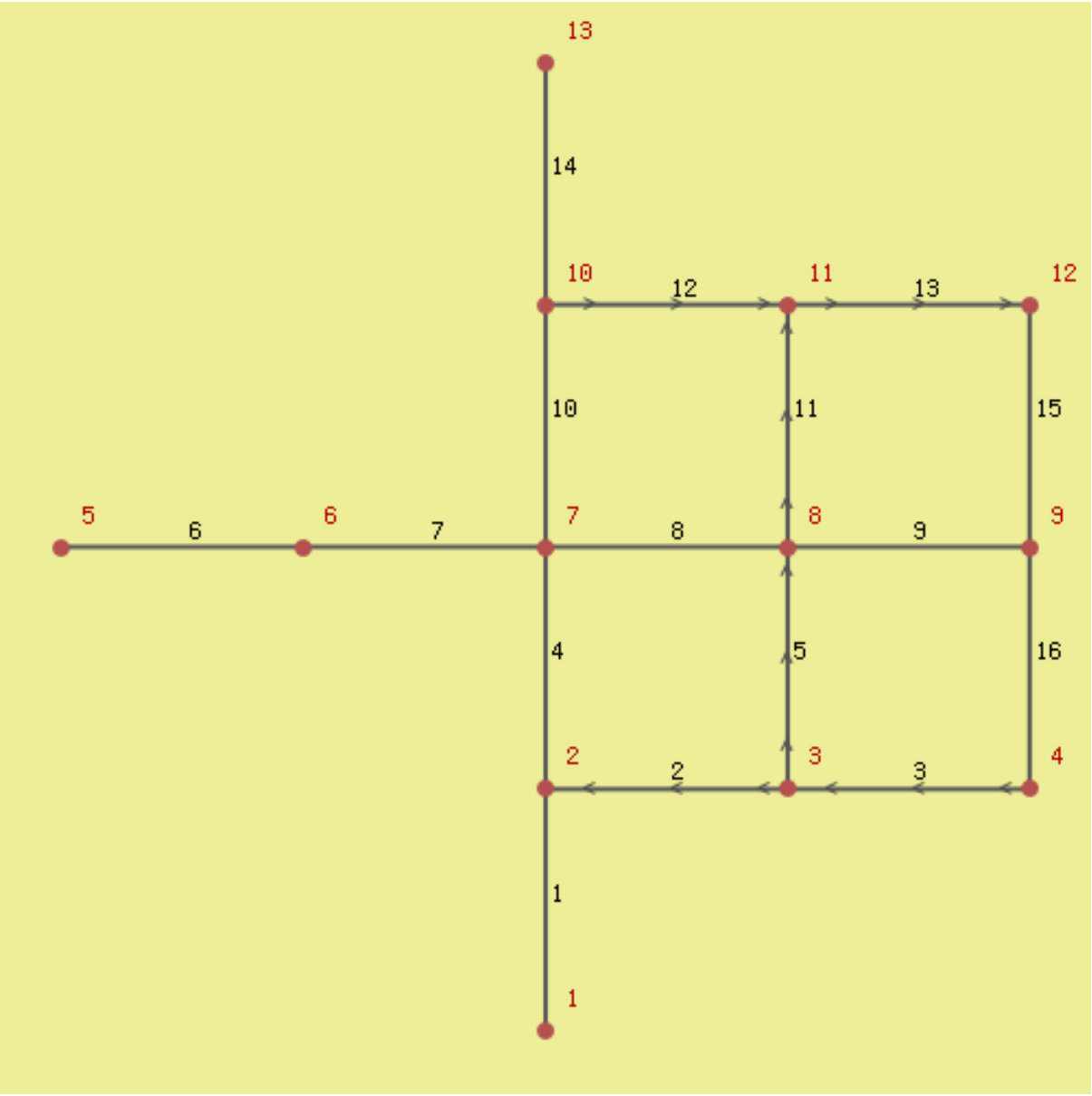
Since the algorithm specific documentation is contained in the source tree with the algorithm specific files, the process of building the documentation and publishing it will need to assemble the details with the front material as needed.

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.

5.1.3 Testing Infrastructure

There is a very basic testing infrastructure put in place. Here are the basics of how it works. We need more test cases. Longer term we should probably get someone to setup travis-ci or jenkins testing frameworks.

Here is the graph for the TRSP tests.



Tests are run via the script at the top level tools/test-runner.pl and it runs all the test configured tests and at the moment just dumps the results structure of the test. This can be prettied up later.

> It also assumes that you have installed the libraries as it tests using the installed postgresql. This probably needs to be made smarter so we can test out of the build tree. I'll need to think about that.

Basically each .../test/ directory should include one *test.conf* file that is a perl script fragment that defines what data files to load and what tests to run. I have built in some mechanisms to allow test and data to be pg version and postgis version specific, but I'm not using that yet. So for example, *core/trsp/test/test-any-00.data* is a sql plain text dump that will load and needed data for a set of tests. This is also the graph in the image above. You can specify multiple files to load, but as a group they need to have unique names.

core/trsp/test/test-any-00.test is a sql command to be run. It will get run as:

```
psql ... -A -t -q -f file.test dbname > tmpfile
diff -w file.rest tmpfile
```

Then if there is a difference then an test failure is reported.

5.2 Release Notes

5.2.1 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[1] 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

5.2.2 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.

[1] 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*