pgRouting Manual

Release 2.0.0-dev (6a63bc1 develop)

pgRouting Contributors

July 17, 2013

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-dev (6a63bc1 develop).

The pgRouting Manual is licensed under a Creative Commons Attribution-Share Alike 3.0 License[3]. Feel free to use this material any way you like, but we ask that you attribute credit to the pgRouting Project and wherever possible, a link back to http://pgrouting.org. For other licenses used in pgRouting see the *License* page.

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

CHAPTER

ONE

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.

1.1.2 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

[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/

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 Support

pgRouting community support is available through website[13], documentation[14], tutorials, mailing lists and others. If you're looking for *commercial support*, find below a list of companies providing pgRouting development and consulting services.

1.2.1 Reporting Problems

Bugs are reported and managed in an issue tracker[15]. 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[16] 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.

1.2.2 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

[13] http://www.pgrouting.org
[14] http://docs.pgrouting.org
[15] https://github.com/pgrouting/pgrouting/issues
[16] https://github.com/pgRouting/pgrouting/issues/new

For general questions and topics about how to use pgRouting, please write to the user mailing list.

You can also ask at GIS StackExchange[17] 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[18].

1.2.3 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

1.3 Installation

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

1.3.1 Windows

Winnie Bot Experimental Builds:

- PostgreSQL 9.2 32-bit, 64-bit[19]

1.3.2 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[20] to install PostGIS 2.0.

1.3.3 RHEL/CentOS/Fedora

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

1.3.4 OSX

See builds from KingChaos[21].

[17]http://gis.stackexchange.com/
[18]http://gis.stackexchange.com/feeds/tag?tagnames=pgrouting&sort=newest
[19]http://winnie.postgis.net/download/windows/pg92/buildbot/
[20]https://launchpad.net/ ubuntugis/+archive/ubuntugis-unstable
[21]http://www.kyngchaos.com/software/postgres

1.4 Build from Source

1.4.1 Source Package

Git 2.0.0-rc1 release	v2.0.0-rc1.tar.gz[22]	v2.0.0-rc1.zip[23]
Git 2.0.0-beta release	v2.0.0-beta.tar.gz[24]	v2.0.0-beta.zip[25]
Git 2.0.0-alpha release	v2.0.0-alpha.tar.gz[26]	v2.0.0-alpha.zip[27]
Git master branch	master.tar.gz[28]	master.zip[29]
Git develop branch	develop.tar.gz[30]	develop.zip[31]

1.4.2 Using Git

Git protocol (read-only):

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

HTTPS protocol (read-only):

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

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

[22]https://github.com/pgRouting/pgrouting/archive/v2.0.0-rc1.tar.gz
[23]https://github.com/pgRouting/pgrouting/archive/v2.0.0-rc1.zip
[24]https://github.com/pgRouting/pgrouting/archive/v2.0.0-beta.tar.gz
[25]https://github.com/pgRouting/pgrouting/archive/v2.0.0-beta.zip
[26]https://github.com/pgRouting/pgrouting/archive/v2.0.0-alpha.tar.gz
[27]https://github.com/pgRouting/pgrouting/archive/v2.0.0-alpha.zip
[28]https://github.com/pgRouting/pgrouting/archive/master.tar.gz
[29]https://github.com/pgRouting/pgrouting/archive/master.zip
[30]https://github.com/pgRouting/pgrouting/archive/develop.tar.gz
[31]https://github.com/pgRouting/pgrouting/archive/develop.zip

CHAPTER

TWO

TUTORIAL

2.1 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

2.1.1 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
```

2.1.2 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.

2.1.3 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, 'the_geom', 'gid');
```

See *pgr_createTopology - Building a Network Topology* for more information.

2.1.4 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', 'the_geom', 0.000001);
select pgr_analyzeoneway('myroads', 'direction', s_in_rules, s_out_rules,
                         t_in_rules, t_out_rules)
```

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 - Nodes an network edge table.* for more information.

2.1.5 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[] - Path Result with Cost* and *pgr_geomResult[] - Path Result with Geometry*.

2.1.6 How to use other tools to view your graph and route

TBD

2.1.7 How to create a web app

TBD

2.2 Routing Topology

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

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

2.2.1 Overview

Typically when GIS files are loaded into the data database for use with pgRouting they do not have topology information assocated 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.

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

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

[1] woodbri@swoodbridge.com

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.

2.2.2 See Also

- *pgr_createTopology - Building a Network Topology*
- *pgr_nodeNetwork - Nodes an network edge table.*
- *pgr_pointToId - Inserts point into a vertices table*

2.3 Graph Analytics

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

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

2.3.1 Overview

It is common to find problems with graphs that have not been constructed fully noded or in graphs with zlevels at intersection that have been entered incorrectly. An other problem is oneway 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 familar with graphviz might contribute tools for generating images with that.

2.3.2 Analyze a Graph

With *pgr_analyzeGraph - Analyze a Graph* the graph can be checked for errors.

```
SELECT pgr_analyzeGraph('mytab', 'the_geom', 0.000002);
```

After the analyzing the graph, deadends are indentified by `cnt=1` in the "vertices_tmp" table and potential problems are identified with `chk=1`.

```
SELECT * FROM vertices_tmp WHERE chk = 1;
```

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

```
SELECT *
    FROM edge_tab a, vertices_tmp b, vertices_tmp 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.

2.3.3 Analyze One Way Streets

pgr_analyzeOneway - Analyze One Way Streets analyzes oneway 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.

[2] woodbri@swoodbridge.com

```
SELECT * FROM vertices_tmp WHERE in=0 OR out=0;
```

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.

2.3.4 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('st', 'one_way',
            ARRAY['', 'B', 'TF'],
            ARRAY['', 'B', 'FT'],
            ARRAY['', 'B', 'FT'],
            ARRAY['', 'B', 'TF'],
            true);

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

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

Typically these problems are generated by a break in the network, the oneway direction set wrong, maybe an error releted to zlevels 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 ot ferry routes are missing.

2.3.5 See Also

- *pgr_analyzeGraph - Analyze a Graph*
- *pgr_analyzeOneway - Analyze One Way Streets*
- *pgr_nodeNetwork - Nodes an network edge table.*

2.4 Custom Query

2.5 Custom Wrapper

2.6 Useful Recipes

2.7 Performance Tips

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

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

CHAPTER

THREE

REFERENCE

pgRouting provides several *common functions*:

3.1 Common Functions

3.1.1 pgr_analyzeGraph - Analyze a Graph

Name

`pgr_analyzeGraph` — Analyzes the "network" and "vertices_tmp" tables.

Synopsis

Analyzes the edge table and "vertices_tmp" tables and flags if nodes are deadends, ie. `vertices_tmp.cnt=1` and identifies nodes that might be disconnected because of gaps < `tolerance` or because of zlevel errors in the data.

```
character varying pgr_analyzeGraph(geom_table text, geo_cname text,
                                   tolerance double precision);
```

Description

geom_table `text` network table name (may contain the schema name as well)

geo_cname `text` geometry column name of the network table

tolerance `float8` snapping tolerance of disconnected edges (in projection unit)

History

- New in version 2.0.0

Examples

```
SELECT pgr_analyzeGraph('edge_table','the_geom',0.0001);

NOTICE:  Adding "cnt" and "chk" columns to vertices_tmp
NOTICE:  Adding unique index "vertices_tmp_id_idx".
NOTICE:  Adding index on "source" for "edge_table".
NOTICE:  Adding index on "target" for "edge_table".
NOTICE:  Adding index on "edge_table" for "the_geom".
NOTICE:  Populating vertices_tmp.cnt
NOTICE:  Analyzing graph for gaps and zlev errors.
```

```
NOTICE:  Found 1 potential problems at 'SELECT * FROM vertices_tmp WHERE chk=1'

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

The queries use the *Sample Data* network.

See Also

- *pgr_analyzeOneway - Analyze One Way Streets*
- *Graph Analytics*

3.1.2 pgr_analyzeOneway - Analyze One Way Streets

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, 1way_cname text, s_in_rules text[], s_out_rules text[],
                       t_in_rules text[], t_out_rules text[], 2way_if_null boolean);
```

Description

geom_table `text` network table name (may contain the schema name as well)

1way_cname `text` oneway column name name of the network table

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

2way_if_null `boolean` flag to treat oneway NULL values as bi-directional

This query will add two columns to the vertices_tmp table `ein int` and `eout int` and populate it with the appropriate counts.

The rules are defined as an array of text strings that if match the `1way_cname` value would be counted as `true` for the source or target **in** or **out** condition.

History

- New in version 2.0.0

Examples

```
SELECT pgr_analyzeOneway('edge_table', 'dir',
    ARRAY['', 'B', 'TF'],
    ARRAY['', 'B', 'FT'],
    ARRAY['', 'B', 'FT'],
    ARRAY['', 'B', 'TF'],
    true);

NOTICE:  Adding "ein" and "eout" columns to vertices_tmp
NOTICE:  Zeroing columns "ein" and "eout" on "vertices_tmp".
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 at 'SELECT * FROM vertices_tmp WHERE ein=0 or eout=0'

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

The queries use the *Sample Data* network.

See Also

- *pgr_analyzeGraph - Analyze a Graph*
- *Graph Analytics*

3.1.3 pgr_createTopology - Building a Network Topology

Name

`pgr_createTopology` — Builds a network topology with source and target information.

Synopsis

The function returns `OK` after the network topology has been built.

```
varchar pgr_createTopology(varchar geom_table, double precision tolerance,
                           varchar geo_cname, cvarchar gid_cname);
```

Description

This function assumes the `source` and `target` columns exist on table `geom_table` and are of type integer or bigint.

```
CREATE TABLE edge_table (
            gid_cname integer,
            source integer, -- or bigint
            target integer, -- or bigint
            geo_cname geometry
);
```

It fills the `source` and `target` id column for all edges. All line ends with a distance less than tolerance, are assigned the same id.

The topology creation function accepts the following parameters:

geom_table `varchar` network table name (may contain the schema name as well)

tolerance `float8` snapping tolerance of disconnected edges (in projection unit)

geo_cname `varchar` geometry column name of the network table

gid_cname `varchar` primary key column name of the network table

The function `OK` after the network topology has been built.

History

- Renamed in version 2.0.0

Examples

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

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

The queries use the *Sample Data* network.

See Also

- *Routing Topology*
- *pgr_nodeNetwork - Nodes an network edge table.*
- *pgr_pointToId - Inserts point into a vertices table*

3.1.4 pgr_isColumnInTable - Check if column exists

Name

`pgr_isColumnInTable` — Check if a column exists in a table.

Synopsis

Returns `true` or `false` if column "col" exists in table "tab".

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

Description

tab `text` table name with or without schema component

col `text` column name to be checked for

History

- New in version 2.0.0

Examples

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

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

The queries use the *Sample Data* network.

See Also

- *pgr_isColumnIndexed - Check if column is indexed*

3.1.5 pgr_isColumnIndexed - Check if column is indexed

Name

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

Synopsis

Returns `true` or `false` if column "col" in table "tab" is indexed.

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

Description

tab `text` table name with or without schema component

col `text` column name to be checked for

History

- New in version 2.0.0

Examples

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

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

The queries use the *Sample Data* network.

See Also

- *pgr_isColumnInTable - Check if column exists*

3.1.6 pgr_nodeNetwork - Nodes an network edge table.

Name

`pgr_nodeNetwork` - Nodes an network edge table.

Author Nicolas Ribot

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

Synopsis

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

```
text pgr_nodeNetwork(text table_in, text gid_cname, text geo_cname,
                     text table_out, double precision tolerance)
```

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 accomodate those situations.

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

table_in `text` name of the input table to be noded

gid_cname `text` name of the primary key column for the input table

geo_cname `text` name of the geometry column that should be noded

table_out `text` name of the output table that will be created

tolerance `float8` tolerance for coinicident points (in projection unit)

The `table_out` table has a structure like:

id `integer` is a reference to the record in `table_in`

sub_id `bigint` is a sequence number that identifies all the new segments generated from the original record.

geom `geometry` attribute

History

- New in version 2.0.0

Examples

```
SELECT * FROM pgr_nodeNetwork('edge_table', 'id', 'the_geom', 'edge_table_noded', 0.000001);

              pgr_nodenetwork
----------------------------------------------
```

```
 edge_table_noded generated with: 16 segments
(1 row)
```

See Also

- *pgr_createTopology - Building a Network Topology*

3.1.7 pgr_pointToId - Inserts point into a vertices table

Name

`pgr_pointToId` — Inserts a point into a temporary vertices table.

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

Synopsis

Inserts a point into a temporary vertices table, and returns an id of a new point or an existing point. Tolerance is the minimal distance between existing points and the new point to create a new point.

```
bigint pgr_pointToId(geometry point, double precision tolerance);
```

Description

point `geometry` of the existing point

tolerance `float8` snapping tolerance of disconnected edges (in projection unit)

Returns point id (`bigint`) of a new or existing point.

History

- Renamed in version 2.0.0

See Also

- *pgr_createTopology - Building a Network Topology*
- *Routing Topology*
- *pgr_nodeNetwork - Nodes an network edge table.*

3.1.8 pgr_quote_ident - Quote table name with Schema Component

Name

`pgr_quote_ident` — Quote table name with or without schema component.

Synopsis

Function to split a string on `.` characters and then quote the components as postgres identifiers and then join them back together with `.` characters. Multile `.` will get collapsed into a single `.`, so `schema...table` till get returned as `schema."table"` and `Schema.table` becomes `"Schema"."table"`.

```
text pgr_quote_ident(text tab);
```

Description

tab `text` table name with or without schema component

Returns table name with or without schema as `text`.

History

- New in version 2.0.0

Examples

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

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


SELECT pgr_quote_ident('edge_table');
 pgr_quote_ident
-----------------
 edge_table
(1 row)


SELECT pgr_quote_ident('Public...edge_table');
   pgr_quote_ident
---------------------
 "Public".edge_table
(1 row)
```

See Also

- [TBD]

3.1.9 pgr_version - Get version information

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 - Compare version numbers*

3.1.10 pgr_versionless - Compare version numbers

Name

`pgr_version` — Compare two version numbers and return if smaller.

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

- *pgr_version - Get version information*

3.2 pgr_analyzeGraph - Analyze a Graph

3.2.1 Name

`pgr_analyzeGraph` — Analyzes the "network" and "vertices_tmp" tables.

3.2.2 Synopsis

Analyzes the edge table and "vertices_tmp" tables and flags if nodes are deadends, ie. `vertices_tmp.cnt=1` and identifies nodes that might be disconnected because of gaps < `tolerance` or because of zlevel errors in the data.

```
character varying pgr_analyzeGraph(geom_table text, geo_cname text,
                                   tolerance double precision);
```

3.2.3 Description

geom_table `text` network table name (may contain the schema name as well)

geo_cname `text` geometry column name of the network table

tolerance `float8` snapping tolerance of disconnected edges (in projection unit)

History

- New in version 2.0.0

3.2.4 Examples

```
SELECT pgr_analyzeGraph('edge_table','the_geom',0.0001);

NOTICE:  Adding "cnt" and "chk" columns to vertices_tmp
NOTICE:  Adding unique index "vertices_tmp_id_idx".
NOTICE:  Adding index on "source" for "edge_table".
NOTICE:  Adding index on "target" for "edge_table".
NOTICE:  Adding index on "edge_table" for "the_geom".
NOTICE:  Populating vertices_tmp.cnt
NOTICE:  Analyzing graph for gaps and zlev errors.
NOTICE:  Found 1 potential problems at 'SELECT * FROM vertices_tmp WHERE chk=1'

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

The queries use the *Sample Data* network.

3.2.5 See Also

- *pgr_analyzeOneway - Analyze One Way Streets*
- *Graph Analytics*

3.3 pgr_analyzeOneway - Analyze One Way Streets

3.3.1 Name

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

3.3.2 Synopsis

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

```
text pgr_analyzeOneway(geom_table text, 1way_cname text, s_in_rules text[], s_out_rules text[],
                       t_in_rules text[], t_out_rules text[], 2way_if_null boolean);
```

3.3.3 Description

geom_table `text` network table name (may contain the schema name as well)

1way_cname `text` oneway column name name of the network table

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

2way_if_null `boolean` flag to treat oneway NULL values as bi-directional

This query will add two columns to the vertices_tmp table `ein int` and `eout int` and populate it with the appropriate counts.

The rules are defined as an array of text strings that if match the `1way_cname` value would be counted as `true` for the source or target **in** or **out** condition.

History

- New in version 2.0.0

3.3.4 Examples

```
SELECT pgr_analyzeOneway('edge_table', 'dir',
    ARRAY['', 'B', 'TF'],
    ARRAY['', 'B', 'FT'],
    ARRAY['', 'B', 'FT'],
    ARRAY['', 'B', 'TF'],
    true);

NOTICE:  Adding "ein" and "eout" columns to vertices_tmp
NOTICE:  Zeroing columns "ein" and "eout" on "vertices_tmp".
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 at 'SELECT * FROM vertices_tmp WHERE ein=0 or eout=0'

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

The queries use the *Sample Data* network.

3.3.5 See Also

- *pgr_analyzeGraph - Analyze a Graph*
- *Graph Analytics*

3.4 pgr_createTopology - Building a Network Topology

3.4.1 Name

`pgr_createTopology` — Builds a network topology with source and target information.

3.4.2 Synopsis

The function returns `OK` after the network topology has been built.

```
varchar pgr_createTopology(varchar geom_table, double precision tolerance,
                           varchar geo_cname, cvarchar gid_cname);
```

3.4.3 Description

This function assumes the `source` and `target` columns exist on table `geom_table` and are of type integer or bigint.

```
CREATE TABLE edge_table (
            gid_cname integer,
            source integer, -- or bigint
            target integer, -- or bigint
            geo_cname geometry
);
```

It fills the `source` and `target` id column for all edges. All line ends with a distance less than tolerance, are assigned the same id.

The topology creation function accepts the following parameters:

geom_table `varchar` network table name (may contain the schema name as well)

tolerance `float8` snapping tolerance of disconnected edges (in projection unit)

geo_cname `varchar` geometry column name of the network table

gid_cname `varchar` primary key column name of the network table

The function `OK` after the network topology has been built.

History

- Renamed in version 2.0.0

3.4.4 Examples

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

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

The queries use the *Sample Data* network.

3.4.5 See Also

- *Routing Topology*
- *pgr_nodeNetwork - Nodes an network edge table.*
- *pgr_pointToId - Inserts point into a vertices table*

3.5 pgr_isColumnInTable - Check if column exists

3.5.1 Name

`pgr_isColumnInTable` — Check if a column exists in a table.

3.5.2 Synopsis

Returns `true` or `false` if column "col" exists in table "tab".

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

3.5.3 Description

tab `text` table name with or without schema component

col `text` column name to be checked for

History

- New in version 2.0.0

3.5.4 Examples

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

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

The queries use the *Sample Data* network.

3.5.5 See Also

- *pgr_isColumnIndexed - Check if column is indexed*

3.6 pgr_isColumnIndexed - Check if column is indexed

3.6.1 Name

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

3.6.2 Synopsis

Returns `true` or `false` if column "col" in table "tab" is indexed.

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

3.6.3 Description

tab `text` table name with or without schema component

col `text` column name to be checked for

History

- New in version 2.0.0

3.6.4 Examples

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

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

The queries use the *Sample Data* network.

3.6.5 See Also

- *pgr_isColumnInTable - Check if column exists*

3.7 pgr_nodeNetwork - Nodes an network edge table.

3.7.1 Name

`pgr_nodeNetwork` - Nodes an network edge table.

Author Nicolas Ribot

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

3.7.2 Synopsis

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

```
text pgr_nodeNetwork(text table_in, text gid_cname, text geo_cname,
                     text table_out, double precision tolerance)
```

3.7.3 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 accomodate those situations.

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

table_in `text` name of the input table to be noded

gid_cname `text` name of the primary key column for the input table

geo_cname `text` name of the geometry column that should be noded

table_out `text` name of the output table that will be created

tolerance `float8` tolerance for coinicident points (in projection unit)

The `table_out` table has a structure like:

id `integer` is a reference to the record in `table_in`

sub_id `bigint` is a sequence number that identifies all the new segments generated from the original record.

geom `geometry` attribute

History

- New in version 2.0.0

3.7.4 Examples

```
SELECT * FROM pgr_nodeNetwork('edge_table', 'id', 'the_geom', 'edge_table_noded', 0.000001);

              pgr_nodenetwork
-------------------------------------------
 edge_table_noded generated with: 16 segments
(1 row)
```

3.7.5 See Also

- *pgr_createTopology - Building a Network Topology*

3.8 pgr_pointToId - Inserts point into a vertices table

3.8.1 Name

`pgr_pointToId` — Inserts a point into a temporary vertices table.

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

3.8.2 Synopsis

Inserts a point into a temporary vertices table, and returns an id of a new point or an existing point. Tolerance is the minimal distance between existing points and the new point to create a new point.

```
bigint pgr_pointToId(geometry point, double precision tolerance);
```

3.8.3 Description

point `geometry` of the existing point

tolerance `float8` snapping tolerance of disconnected edges (in projection unit)

Returns point id (`bigint`) of a new or existing point.

History

- Renamed in version 2.0.0

3.8.4 See Also

- *pgr_createTopology - Building a Network Topology*
- *Routing Topology*
- *pgr_nodeNetwork - Nodes an network edge table.*

3.9 pgr_quote_ident - Quote table name with Schema Component

3.9.1 Name

`pgr_quote_ident` — Quote table name with or without schema component.

3.9.2 Synopsis

Function to split a string on `.` characters and then quote the components as postgres identifiers and then join them back together with `.` characters. Multile `.` will get collapsed into a single `.`, so `schema...table` till get returned as `schema."table"` and `Schema.table` becomes `"Schema"."table"`.

```
text pgr_quote_ident(text tab);
```

3.9.3 Description

tab `text` table name with or without schema component

Returns table name with or without schema as `text`.

History

- New in version 2.0.0

3.9.4 Examples

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

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


SELECT pgr_quote_ident('edge_table');
 pgr_quote_ident
-----------------
 edge_table
(1 row)


SELECT pgr_quote_ident('Public...edge_table');
   pgr_quote_ident
---------------------
 "Public".edge_table
(1 row)
```

3.9.5 See Also

- [TBD]

3.10 pgr_version - Get version information

3.10.1 Name

`pgr_version` — Query for pgRouting version information.

3.10.2 Synopsis

Returns a table with pgRouting version information.

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

3.10.3 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

3.10.4 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)
```

3.10.5 See Also

- *pgr_versionless - Compare version numbers*

3.11 pgr_versionless - Compare version numbers

3.11.1 Name

`pgr_version` — Compare two version numbers and return if smaller.

3.11.2 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);
```

3.11.3 Description

v1 `text` first version number

v2 `text` second version number

History

- New in version 2.0.0

3.11.4 Examples

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

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

3.11.5 See Also

- *pgr_version - Get version information*

pgRouting defines a few *custom data types*:

3.12 Custom Types

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

3.12.1 pgr_costResult[] - Path Result with Cost

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

id2 generic name, to be specified by the function

cost cost attribute

History

- New in version 2.0.0
- Replaces `path_result`

See Also

- *Custom Types*

3.12.2 pgr_geomResult[] - Path Result with Geometry

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

- *Custom Types*

3.13 pgr_costResult[] - Path Result with Cost

3.13.1 Name

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

3.13.2 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

id2 generic name, to be specified by the function

cost cost attribute

History

- New in version 2.0.0
- Replaces `path_result`

3.13.3 See Also

- *Custom Types*

3.14 pgr_geomResult[] - Path Result with Geometry

3.14.1 Name

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

3.14.2 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`

3.14.3 See Also

- *Custom Types*

pgRouting functions in alphabetical order:

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

3.15.1 Name

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

3.15.2 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);
```

3.15.3 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[] - Path Result with Cost*:

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

3.15.4 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 |  7 |    2
   3 |    1 |  8 |    3
[...]
```

The query uses the *Sample Data* network.

3.15.5 See Also

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

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

3.16.1 Name

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

3.16.2 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);
```

3.16.3 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[] - Path Result with Cost*:

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

3.16.4 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.

3.16.5 See Also

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

3.17 pgr_astar - Shortest Path A*

3.17.1 Name

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

3.17.2 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);
```

3.17.3 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[] - Path Result with Cost*:

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

3.17.4 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',
                7, 12, false, false
        );

 seq | node | edge | cost
-----+------+------+------
```

```
   0 |    7 |    8 |    1
   1 |    8 |   11 |    1
   2 |   11 |   13 |    1
   3 |   12 |   -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',
                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.

3.17.5 See Also

- *pgr_costResult[] - Path Result with Cost*
- http://en.wikipedia.org/wiki/A*_search_algorithm

3.18 pgr_bdAstar - Bi-directional A* Shortest Path

3.18.1 Name

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

3.18.2 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);
```

3.18.3 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[] - Path Result with Cost*:

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

3.18.4 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',
        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_bdAstar(
        'SELECT id, source, target, cost, x1, y1, x2, y2, 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.

3.18.5 See Also

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

3.19 pgr_bdDijkstra - Bi-directional Dijkstra Shortest Path

3.19.1 Name

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

3.19.2 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);
```

3.19.3 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[] - Path Result with Cost*:

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

3.19.4 Examples

- Without `reverse_cost`

```
SELECT seq, id1 AS node, id2 AS edge, cost
    FROM pgr_bdDijkstra(
        '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_bdDijkstra(
        '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.

3.19.5 See Also

- *pgr_costResult[] - Path Result with Cost*
- *pgr_bdAstar - Bi-directional A* Shortest Path*
- http://en.wikipedia.org/wiki/Bidirectional_search

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

3.20 pgr_dijkstra - Shortest Path Dijkstra

3.20.1 Name

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

3.20.2 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);
```

3.20.3 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[] - Path Result with Cost*:

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

3.20.4 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.

3.20.5 See Also

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

3.21 pgr_kDijkstra - Mutliple destination Shortest Path Dijkstra

3.21.1 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.

3.21.2 Synopsis

This function allow you to have a single start node and multiple destination nodes and will compute the routes to all the destinations from the source node. Returns a set of *pgr_costResult* (seq, id1, id2, cost) rows, that summarize the total path cost. `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_costResult[] pgr_kdijkstraPath(text sql, integer source,
                 integer[] targets, boolean directed, boolean has_rcost);
```

3.21.3 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[] - Path Result with Cost*:

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_costResult[] - Path Result with Cost*:

seq row sequence

id1 path vertex target id (identifies the target path).

id2 path edge id

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

History

- New in version 2.0.0

3.21.4 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 edge, cost FROM pgr_kdijkstraPath(
    'SELECT id, source, target, cost FROM edge_table',
    10, array[4,12], false, false
);

 seq | path | edge | cost
-----+------+------+------
   0 |    4 |   12 |    1
   1 |    4 |   13 |    1
   2 |    4 |   15 |    1
   3 |    4 |   16 |    1
   4 |   12 |   12 |    1
   5 |   12 |   13 |    1
(6 rows)
```

3.21.5 See Also

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

3.22 pgr_ksp - K-Shortest Path

3.22.1 Name

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

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

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

3.22.3 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[] - Path Result with Cost*:

seq route ID

id1 node ID

id2 edge ID (`0` for the last row)

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

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

History

- New in version 2.0.0

3.22.4 Examples

- Without `reverse_cost`

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

 route | node | edge | cost
-------+------+------+------
     0 |    7 |    8 |    1
     0 |    8 |   11 |    1
     0 |   11 |   13 |    1
     0 |   12 |    0 |    0
     1 |    7 |    8 |    1
     1 |    8 |    9 |    1
     1 |    9 |   15 |    1
     1 |   12 |    0 |    0
(8 rows)
```

- With `reverse_cost`

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

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

```
      0 |    8 |   11 |    1
      0 |   11 |   13 |    1
      0 |   12 |    0 |    0
      1 |    7 |    8 |    1
      1 |    8 |    9 |    1
      1 |    9 |   15 |    1
      1 |   12 |    0 |    0
(8 rows)
```

The queries use the *Sample Data* network.

3.22.5 See Also

- *pgr_costResult[] - Path Result with Cost*
- http://en.wikipedia.org/wiki/K_shortest_path_routing

3.23 pgr_tsp - Traveling Sales Person

3.23.1 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.

3.23.2 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)
```

3.23.3 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[] - Path Result with Cost*:

seq row sequence

id1 internal index to the distance matric

id2 `id` of the node

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

Create a distance matrix

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

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

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

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

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

With distance matrix

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

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

start `int4` index of the start point

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

[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.

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

3.23.4 Examples

- Using SQL parameter (all points from the table, atarting from 6 and ending at 5)

```
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 |   1 |   2 | 1.00
   3 |   0 |   1 | 1.41
   4 |   2 |   3 | 1.00
   5 |   7 |   8 | 1.41
   6 |   3 |   4 | 1.00
   7 |   8 |   9 | 1.00
   8 |  11 |  12 | 1.00
   9 |  10 |  11 | 1.00
  10 |   9 |  10 | 1.00
  11 |  12 |  13 | 2.83
  12 |   4 |   5 | 1.00
(13 rows)
```

- Using distance matrix (A loop starting from 1)

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

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

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

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

 seq | id
-----+----
   0 |  1
```

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

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

The queries use the *Sample Data* network.

3.23.5 See Also

- *pgr_costResult[] - Path Result with Cost*
- http://en.wikipedia.org/wiki/Traveling_salesman_problem
- http://en.wikipedia.org/wiki/Simulated_annealing

3.24 pgr_trsp - Turn Restriction Shortest Path (TRSP)

3.24.1 Name

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

3.24.2 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]);
```

3.24.3 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[] - Path Result with Cost*:

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

3.24.4 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 |    8 |    1
   1 |    8 |   11 |    1
   2 |   11 |   13 |    1
   3 |   12 |   -1 |    0
(4 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 |     8 |     1
   1 |     8 |    11 |     1
   2 |    11 |    13 |     1
   3 |    12 |    -1 |     0
(4 rows)
```

The queries use the *Sample Data* network.

3.24.5 See Also

- *pgr_costResult[] - Path Result with Cost*

If pgRouting is compiled with "Driving Distance" enabled:

3.25 pgr_alphashape - Alpha shape computation

3.25.1 Name

`pgr_alphashape` — Core function for alpha shape computation.

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

3.25.2 Synopsis

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

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

3.25.3 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

3.25.4 Examples

```
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)
```

The queries use the *Sample Data* network.

3.25.5 See Also

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

3.26 pgr_drivingDistance - Driving Distance

3.26.1 Name

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

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

3.26.2 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.

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

3.26.3 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[] - Path Result with Cost*:

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

3.26.4 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.

3.26.5 See Also

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

3.27 pgr_pointsAsPolygon - Polygon around set of points

3.27.1 Name

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

3.27.2 Synopsis

Returns the alpha shape as polygon geometry.

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

3.27.3 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

3.27.4 Examples

```
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)
```

The queries use the *Sample Data* network.

3.27.5 See Also

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

Some functions from previous releases may have been removed.

3.28 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.

3.28.1 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;
```

3.28.2 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)
```

3.29 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.

3.29.1 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

FOUR

DEVELOPER

4.1 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)
- The Boost Graph Library (BGL). Version >= [TBD]
- CMake >= 2.8.8
- (optional, for Driving Distance) CGAL >= [TBD]
- (optional, for Documentation) Sphinx >= 1.1

4.1.1 For MinGW on Windows

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

4.1.2 For Linux

```
mkdir build
cd build
cmake -DWITH_DD=ON ..
make
sudo make install
```

4.1.3 With Documentation

Build with documentation (requires Sphinx[1]):

```
cmake -DWITH_DOC=ON -DWITH_DD=ON ..
```

Rebuild modified documentation only:

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

[1] http://sphinx-doc.org/

4.2 Developer's Guide

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

4.2.1 Source Tree Layout

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

core/ This is the algorithm source tree. Each alogorithm 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.

4.2.2 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

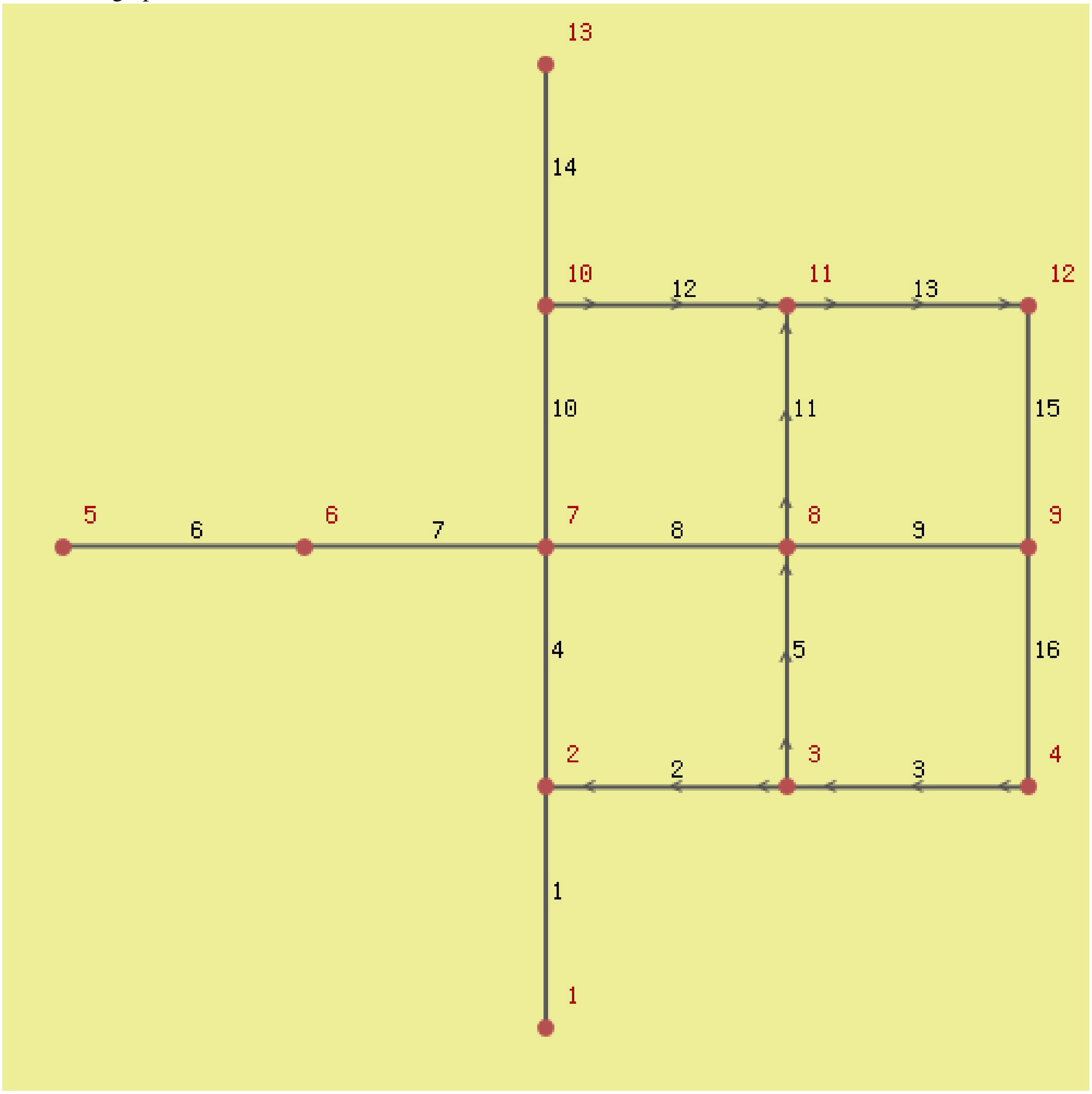
Since the algorithm specific documentation is contained in the source tree with the algorithm specific files, the process of building the documentation and publishing it will need to assemble the details with the front material as needed.

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.

4.2.3 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.

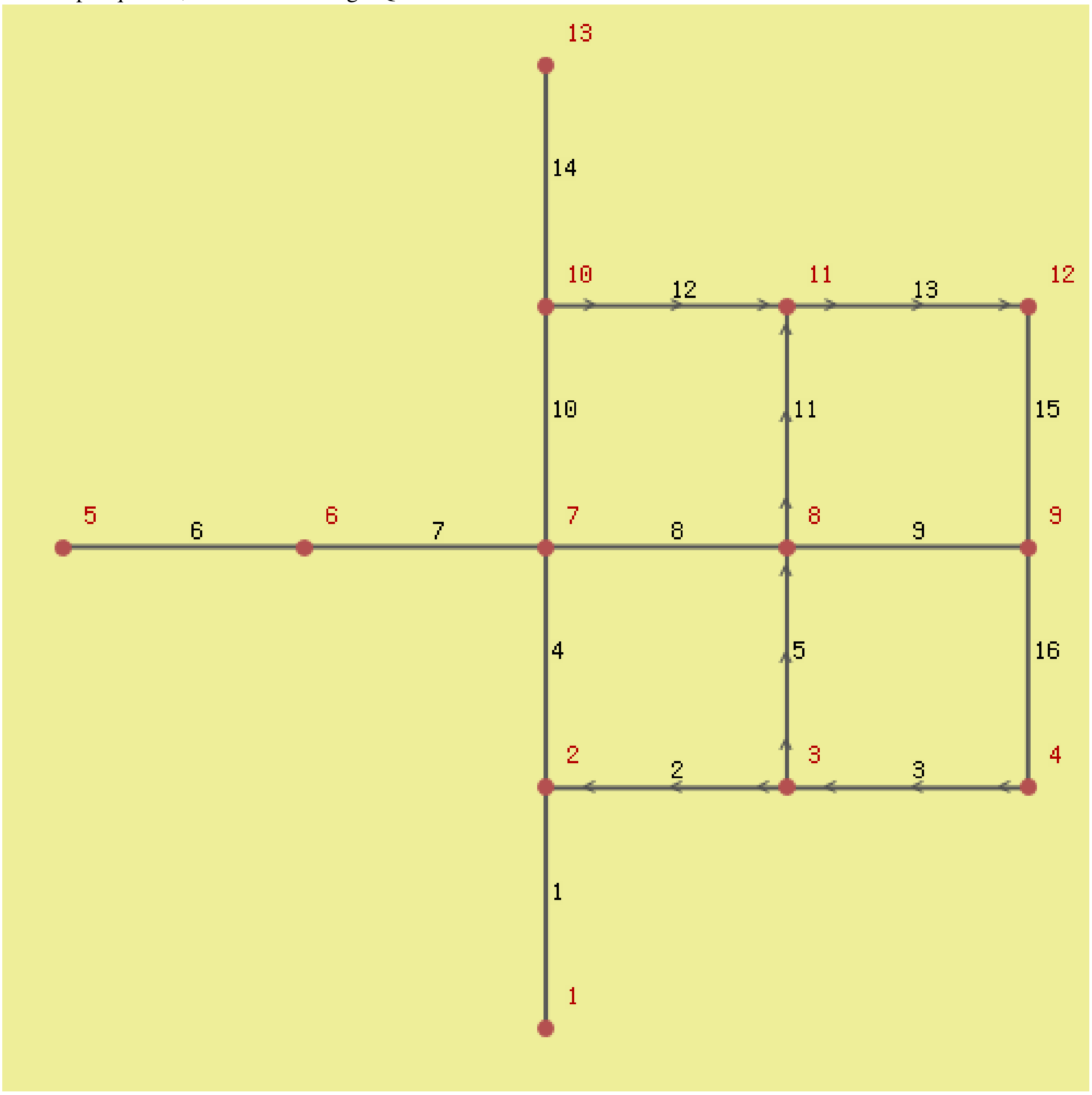
core/trsp/test/test-any-00.test is a sql command to be run. It will get run as:

```
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.

4.3 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 dataset.



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,
    to_cost double precision,
    rule text,
    the_geom geometry(Linestring)
);

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

Insert network data

```
    INSERT INTO edge_table VALUES (1, ’B’, 1, 2, 1, 1, 2, 0, 2, 1, NULL, NULL,
’0102000000020000000000000000000040000000000000000000000000000040000000000000F03F’);
    INSERT INTO edge_table VALUES (2, ’TF’, 2, 3, -1, 1, 2, 1, 3, 1, NULL, NULL,
’0102000000020000000000000000000040000000000000F03F0000000000000840000000000000F03F’);
    INSERT INTO edge_table VALUES (3, ’TF’, 3, 4, -1, 1, 3, 1, 4, 1, NULL, NULL,
’0102000000020000000000000000000840000000000000F03F0000000000001040000000000000F03F’);
    INSERT INTO edge_table VALUES (4, ’B’, 2, 7, 1, 1, 2, 1, 2, 2, NULL, NULL,
’0102000000020000000000000000000040000000000000F03F00000000000000400000000000000040’);
    INSERT INTO edge_table VALUES (5, ’FT’, 3, 8, 1, -1, 3, 1, 3, 2, NULL, NULL,
’0102000000020000000000000000000840000000000000F03F00000000000008400000000000000040’);
    INSERT INTO edge_table VALUES (6, ’B’, 5, 6, 1, 1, 0, 2, 1, 2, NULL, NULL,
’01020000000200000000000000000000000000000000000040000000000000F03F0000000000000040’);
    INSERT INTO edge_table VALUES (7, ’B’, 6, 7, 1, 1, 1, 2, 2, 2, NULL, NULL,
’010200000002000000000000000000F03F000000000000004000000000000000400000000000000040’);
    INSERT INTO edge_table VALUES (8, ’B’, 7, 8, 1, 1, 2, 2, 3, 2, NULL, NULL,
’0102000000020000000000000000000040000000000000004000000000000008400000000000000040’);
    INSERT INTO edge_table VALUES (9, ’B’, 8, 9, 1, 1, 3, 2, 4, 2, NULL, NULL,
’0102000000020000000000000000000840000000000000004000000000000010400000000000000040’);
    INSERT INTO edge_table VALUES (10, ’B’, 7, 10, 1, 1, 2, 2, 2, 3, NULL, NULL,
’0102000000020000000000000000000040000000000000004000000000000000400000000000000840’);
    INSERT INTO edge_table VALUES (11, ’FT’, 8, 11, 1, -1, 3, 2, 3, 3, NULL, NULL,
’0102000000020000000000000000000840000000000000004000000000000008400000000000000840’);
    INSERT INTO edge_table VALUES (12, ’FT’, 10, 11, 1, -1, 2, 3, 3, 3, NULL, NULL,
’0102000000020000000000000000000040000000000000084000000000000008400000000000000840’);
    INSERT INTO edge_table VALUES (13, ’FT’, 11, 12, 1, -1, 3, 3, 4, 3, NULL, NULL,
’0102000000020000000000000000000840000000000000084000000000000010400000000000000840’);
    INSERT INTO edge_table VALUES (14, ’B’, 10, 13, 1, 1, 2, 3, 2, 4, NULL, NULL,
’0102000000020000000000000000000040000000000000084000000000000000400000000000001040’);
    INSERT INTO edge_table VALUES (15, ’B’, 9, 12, 1, 1, 4, 2, 4, 3, NULL, NULL,
’0102000000020000000000000000001040000000000000004000000000000010400000000000000840’);
    INSERT INTO edge_table VALUES (16, ’B’, 4, 9, 1, 1, 4, 1, 4, 2, NULL, NULL,
’0102000000020000000000000000001040000000000000F03F00000000000010400000000000000040’);
```

```
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);
```

Release Notes

4.4 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[2] on Github.

4.4.1 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
- 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

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

4.5 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.

4.5.1 Changes for release 1.05

- Bugfixes

4.5.2 Changes for release 1.03

- Much faster topology creation
- Bugfixes

4.5.3 Changes for release 1.02

- Shooting* bugfixes
- Compilation problems solved

4.5.4 Changes for release 1.01

- Shooting* bugfixes

4.5.5 Changes for release 1.0

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

4.5.6 Changes for release 1.0.0b

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

4.5.7 Changes for release 1.0.0a

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

4.5.8 Changes for release 0.9.9

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

4.5.9 Changes for release 0.9.8

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

Indices and tables

- *genindex*
- *search*