pgRouting Manual

Version 2.0.0 (d4d49b7 master)

pgRouting Contributors

20 September 2013

Table des matières

pgRouting est une extension de la base de données géospatiale PostGIS [1]/PostgreSQL [2] afin de proposer des fonctionnalités de routage géospatial et d'autres analyses de réseaux.

Ceci est le manuel de pgRouting 2.0.0 (d4d49b7 master).

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

Général

1.1 Introduction

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

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

1.1.1 License

The following licenses can be found in pgRouting :

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

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

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

1.1.2 Contributors

Individuals (in alphabetical order)

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

Corporate Sponsors (in alphabetical order)

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

Camptocamp, CSIS (University of Tokyo), Georepublic, Google Summer of Code, iMaptools, Orkney, Paragon Corporation

1.1.3 More Information

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

1.2 Installation

Les paquets binaires sont fournis pour la version courante sur les plateformes suivantes :

1.2.1 Windows

Winnie Bot Experimental Builds :
– PostgreSQL 9.2 32-bit, 64-bit [13]

1.2.2 Ubuntu/Debian

Les paquets Ubuntu sont disponibles dans les dépôts Launchpad :
– *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
```

Utiliser UbuntuGIS-unstable PPA [14] pour installer PostGIS 2.0.

1.2.3 RHEL/CentOS/Fedora

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

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

1.2.4 OS X

– Homebrew

```
brew install pgrouting
```

1.2.5 Paquets Sources

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

1.2.6 Utiliser Git

Protocole Git (en lecture seule) :

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

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

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

Voir *Guide pour le Build* les notes sur comment compiler depuis les sources.

1.3 Guide pour le Build

Pour être capable de compiler pgRouting, vérifiez que les dépendances suivantes sont présentes :
– Compilateurs C et C++
– Version Postgresql >= 8.4 (>= 9.1 recommandée)
– Version PostGIS >= 1.5 (>= 2.0 recommandée)
– La Boost Graph Library (BGL). Version >= [à déterminer]
– CMake >= 2.8.8
– (optionnel, pour Driving Distance) CGAL >= [à déterminer]
– (optionnel, pour la Documentation) Sphinx >= 1.1
– (optional, for Documentation as PDF) Latex >= [TBD]
The cmake system has variables the can be configured via the command line options by setting them with -D<varable>=<value>. You can get a listing of these via :

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

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

Currently these are :

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

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

WITH_DD :BOOL=ON – Turn on/off building driving distance code. WITH_DOC :BOOL=OFF – Turn on/off building the documentation BUILD_HTML :BOOL=ON – If WITH_DOC=ON, turn on/off building HTML BUILD_LATEX :BOOL=OFF – If WITH_DOC=ON, turn on/off building PDF BUILD_MAN :BOOL=ON – If WITH_DOC=ON, turn on/off building MAN pages

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

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

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

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

1.3.1 Pour MinGW sur Windows

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

1.3.2 Pour Linux

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

1.3.3 Avec la Documentation

Build avec la documentation (recquiert Sphinx [25]) :

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

Rebuild seulement la documentation modifiée :

25. http ://sphinx-doc.org/

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

1.4 Support

Le support de la communauté pgRouting est disponible depuis le site web [26], documentation [27], tutoriels, listes emails et autres. Si vous cherchez un *support commercial*, vous trouverez ci-dessous une liste des sociétés fournissant des services de développement et de conseil.

1.4.1 Reporter un problème

Les bugs sont reportés et gérés dans un système de gestion d'incidents [28]. Merci de suivre ces étapes :

1. Cherchez les tickets pour voir si votre problème a déjà été reporté. Si oui, ajoutez tout contexte supplémentaire que vous pourriez avoir trouvé, ou vous pouvez simplement ajouter que vous avez également le même problème. Cela nous aidera à établir la priorité des incidents en commun.
2. Si votre problème n'est pas reporté, créer un nouvel incident [29] pour ça.
3. Dans votre rapport, incluez explicitement les instructions pour reproduire votre problème. Les meilleurs tickets incluent le SQL exact nécessaire pour reproduire un problème.
4. Si vous pouvez tester les versions anciennes de PostGIS pour votre problème, merci de le faire. Dans votre ticket, mentionnez la plus récente version où le problème apparaît.
5. Pour les versions où vous pouvez reproduire le problème, notez le système d'exploitation et la version de pgRouting, PostGIS et PostgreSQL.
6. It is recommended to use the following wrapper on the problem to pin point the step that is causing the problem.

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

1.4.2 Liste d'emails et StackExchange SIG

Il y a deux listes emails pour pgRouting hébergé sur le serveur de mailing list OSGeo.
– Listes d'emails utilisateurs : http ://lists.osgeo.org/mailman/listinfo/pgrouting-users
– Liste des emails développeurs : http ://lists.osgeo.org/mailman/listinfo/pgrouting-dev
Pour les questions générales et les sujets sur comment utiliser pgRouting, veuillez écrire à la liste des emails utilisateurs.

Vous pouvez aussi demander à GIS StackExchange [30] et étiquetter la question avec `pgrouting`. Trouvez toutes les questions étiquetées avec `pgrouting` ici http ://gis.stackexchange.com/questions/tagged/pgrouting ou abonnez-vous à pgRouting questions feed [31].

1.4.3 Support Commercial

Pour les utilisateurs qui demandent un support professionnel et des services de conseil, veuillez contacter l'une des organisations suivantes, qui ont contribué de façon significative au développement de pgRouting :

26. http ://www.pgrouting.org
27. http ://docs.pgrouting.org
28. https ://github.com/pgrouting/pgrouting/issues
29. https ://github.com/pgRouting/pgrouting/issues/new
30. http ://gis.stackexchange.com/
31. http ://gis.stackexchange.com/feeds/tag ?tagnames=pgrouting&sort=newest

Société	Bureaux à	Site web
Georepublic	Allemagne, Japon	http ://georepublic.info
iMaptools	États-Unis	http ://imaptools.com
Orkney Inc.	Japon	http ://www.orkney.co.jp
Camptocamp	Suisse, France	http ://www.camptocamp.com

CHAPITRE 2

Tutoriel

2.1 Tutorial

2.1.1 Getting Started

This is a simple guide to walk you through the steps of getting started with pgRouting. In this guide we will cover :
– How to create a database to use for our project
– How to load some data
– How to build a topology
– How to check your graph for errors
– How to compute a route
– How to use other tools to view your graph and route
– How to create a web app

How to create a database to use for our project

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

For Postgresql 9.1 and later versions

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

For older versions of postgresql

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

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

createdb -T template_pgrouting mydatabase
```

How to load some data

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

shp2pgsql
– this is the postgresql shapefile loader

ogr2ogr
– this is a vector data conversion utility

osm2pgsql
– this is a tool for loading OSM data into postgresql

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

How to build a topology

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

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

See *pgr_createTopology* for more information.

How to check your graph for errors

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

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

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

See *Analytiques de graphe* for more information.

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

See *pgr_nodeNetwork* for more information.

How to compute a route

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

```
select pgr_<algorithm>(<SQL for edges>, start, end, <additonal options>)
```

As you can see this is fairly straight forward and you can look and the specific algorithms for the details on how to use them. What you get as a result from these queries will be a set of record of type *pgr_costResult[]* or *pgr_geomResult[]*. These results have information like edge id and/or the node id along with the cost or geometry for the step in the path from *start* to *end*. Using the ids you can join these result back to your edge table to get more information about each step in the path.
– See also *pgr_costResult[]* and *pgr_geomResult[]*.

How to use other tools to view your graph and route

TBD

How to create a web app

TBD

2.1.2 Topologie de routage

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

Présentation

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

Vous pouvez utiliser les *graph analysis functions* pour vous aider à voir où vous pourrie avoir des problèmes de topologie dans vos données. Si vous avez besoin de nouer vos données, nous avons aussi une fonction *pgr_nodeNetwork()* qui pourrait fonctionner pour vous. La fonction sépare TOUS les segments et les noue. Il y a certaines cas où cela pourrait ne pas être la bonne chose à faire.

Par exemple, quand vous avez une intersection entre une route supérieure et inférieure, vous ne voulez pas qu'elle soit nouée, mais pgr_nodeNetwork ne sait pas que c'est le cas et va les nouer avec eux ce qui n'est pas bien parce qu'ensuite le conducteur sera capable d'éteindre la route supérieure sur la route inférieure comme cela était une intersection plate 2D. Pour faire face à ce problème certains jeux de données à ces types utilisent les z-levels à ces types d'intersections et autres données pourraient ne pas nouer cette intersection ce qui serait ok.

Pour ces cas où la topologie a besoin d'être ajoutée les fonctions suivantes peuvent être utiles. Une façon de préparer les données pour pgRouting est d'ajouter les colonnes suivantes à votre table et ensuite les remplir comme approprié. Cet exemple fait un tas d'hypothèses comme que vos tables de données originales ont certaines colonnes comme `one_way`, `fcc`, et possiblement autres et qu'ils contiennent des valeurs de données spécifiques. Ceci est seulement pour vous donner une idée de ce que vous pouvez faire avec vos données.

```
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

La fonction *pgr_createTopology()* va créer la table `vertices_tmp` et remplir la `source` et les colonnes `target`. L'exemple suivant remplit les colonnes suivantes. Dans cet exemple, la colonne `fcc` contient le code de la classe fonctionnalité et les instructions `CASE` le convertissent en une vitesse moyenne.

```
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;
```

Maintenant votre base de données devrait être prête pour utiliser n'importe (la plupart ?) des algorithmes de

pgRouting.

Voir aussi

– *pgr_createTopology*
– *pgr_nodeNetwork*
– *pgr_pointToId*

2.1.3 Analytiques de graphe

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

Présentation

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

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

Analyser un graphe

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

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

In the vertices table "mytab_vertices_pgr" :
– Deadends are indentified by `cnt=1`
– Potencial gap problems are identified with `chk=1`.

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

2. woodbri@swoodbridge.com

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

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

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

Si vous voulez visualiser ceux-ci sur une image graphique, alors vous pouvez utiliser quelque chose comme mapserver pour rendre les arêtes et les sommets et le style basé sur `cnt` ou ils sont isolés, etc. Vous pouvez aussi faire cela avec un outil comme graphviz, ou geoserver ou autres outils similaires.

Analyser les routes à sens unique

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

Cette requête va ajouter deux colonnes à la table vertices_tmp `ein int` et `eout int` et la remplir avec les comptes appropriés. Après avoir exécuté ceci sur un graphe vous pouvez identifier les noeuds avec des problèmes potentiels avec la requête suivante.

Les règles sont définies comme un tableau de chaînes de caractères qui s'ils correspondent à la valeur `col` serait être comptée comme vraie pour la source ou cible sous ou en dehors de la condition.

Exemple

Supposons que nous avons un tableau "st" des arêtes et une colonne à "sens unique" qui pourrait avoir des valeurs comme :
– 'FT' - sens unique de la source au noeud cible.
– 'TF' - sens unique de la cible au noeud source.
– 'B' - route à deux voies.
– '' - champ vide, supposé à deux voies.
– <NULL> - champ NULL, utiliser le flag two_way_if_null.
Ensuite nous pourrions former la requête suivante pour analyser les routes à sens unique pour les erreurs.

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

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

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

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

The above tools do not detect all network issues, but they will identify some common problems. There are other problems that are hard to detect because they are more global in nature like multiple disconnected networks. Think

of an island with a road network that is not connected to the mainland network because the bridge or ferry routes are missing.

Voir aussi

– *pgr_analyzeGraph*
– *pgr_analyzeOneway*
– *pgr_nodeNetwork*

2.1.4 Requête personnalisée

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

id int4

source int4

target int4

cost float8

reverse_cost float8

x float8

y float8

x1 float8

y1 float8

x2 float8

y2 float8

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

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

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

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

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

id int4 Default `id`

source int4 Default `source`

target int4 Default `target`

the_geom text Default `the_geom`

oneway text Default `oneway`

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

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

edge_table text

tolerance float8

s_in_rules text[]

s_out_rules text[]

t_in_rules text[]

t_out_rules text[]

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

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

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

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

2.1.5 Conseils pour la performance

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

Analize the new topology based on the actual topology :

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

Or create a new topology if the change is permanent :

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

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

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

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

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

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

2.1.6 User's wrapper contributions

How to contribute.

Use an issue tracker (see *Support*) with a title containing : *Proposing a wrapper : Mywrappername*. The body will contain :
– author : Required
– mail : if you are subscribed to the developers list this is not necessary
– date : Date posted
– comments and code : using reStructuredText format
Any contact with the author will be done using the developers mailing list. The pgRouting team will evaluate the wrapper and will be included it in this section when approved.

No contributions at this time

2.1.7 Use's Recipes contributions

How to contribute.

Use an issue tracker (see *Support*) with a title containing : *Proposing a Recipe : Myrecipename*. The body will contain :
– author : Required
– mail : if you are subscribed to the developers list this is not necessary
– date : Date posted
– comments and code : using reStructuredText format
Any contact with the author will be done using the developers mailing list. The pgRouting team will evaluate the recipe and will be included it in this section when approved.

Comparing topology of a unnoded network with a noded network

Author pgRouting team.

This recipe uses the *Données d'échantillon* network.

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

No more contributions

2.2 Données d'échantillon

The documentation provides very simple example queries based on a small sample network. To be able to execute the sample queries, run the following SQL commands to create a table with a small network data set.

Créer une table

```
CREATE TABLE edge_table (
    id serial,
    dir character varying,
    source integer,
    target integer,
    cost double precision,
    reverse_cost double precision,
    x1 double precision,
    y1 double precision,
    x2 double precision,
    y2 double precision,
    the_geom geometry
);
```

Insérer les données du réseau

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

```
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  0,2,   1,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  1,2,   2,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,2,   3,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  3,2,   4,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,2,   2,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  3,2,   3,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  2,3,   3,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1,-1,  3,3,   4,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  2,3,   2,4);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  4,2,   4,3);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  4,1,   4,2);
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  0.5,3.5,  1.999999999999,3
INSERT INTO edge_table (cost,reverse_cost,x1,y1,x2,y2) VALUES ( 1, 1,  3.5,2.3,  3.5,4);


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

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

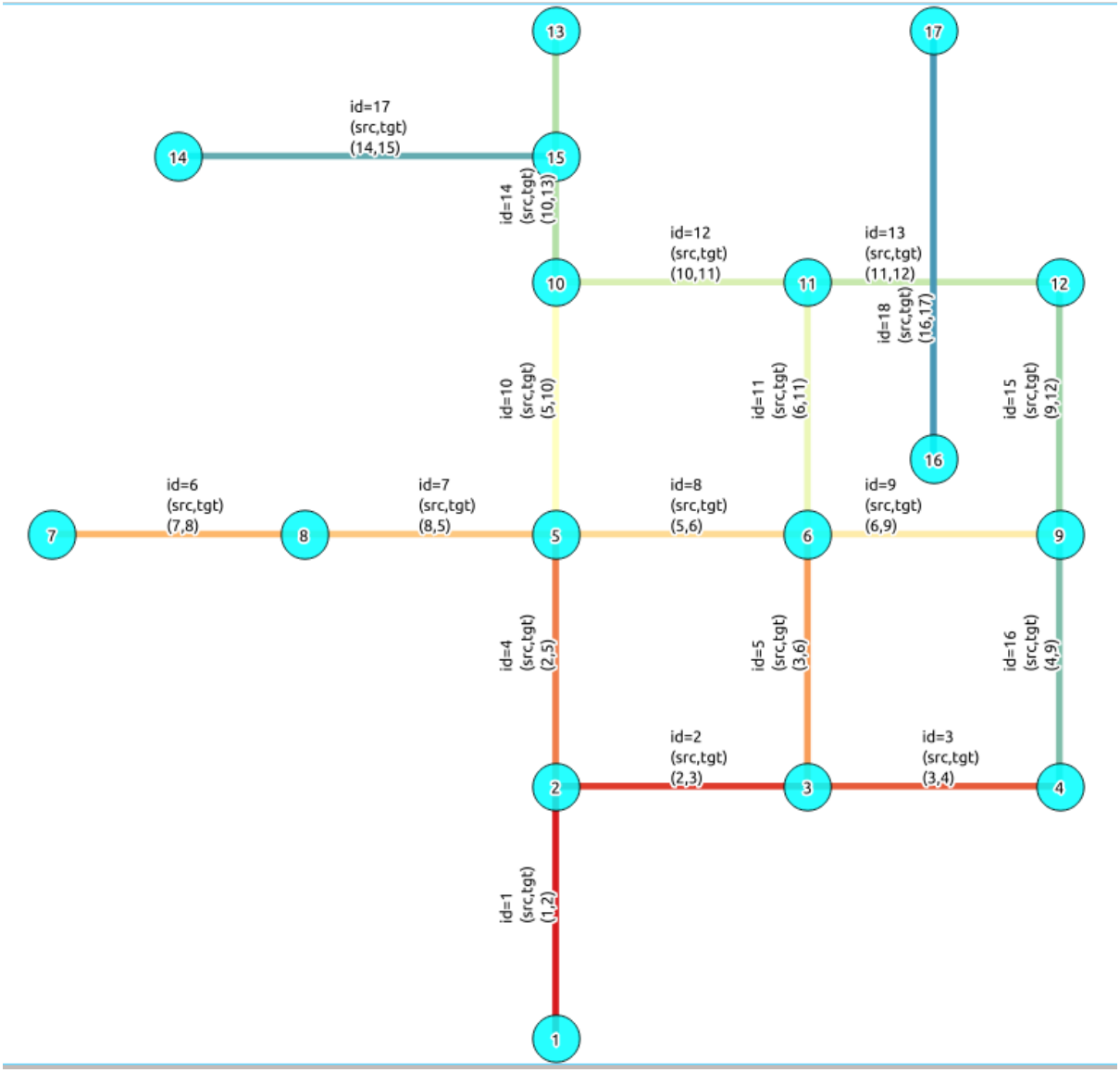
```
SELECT pgr_createTopology('edge_table',0.001);
```

This table is used in some of our examples

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

INSERT INTO vertex_table VALUES
        (1,2,0), (2,2,1), (3,3,1), (4,4,1), (5,0,2), (6,1,2), (7,2,2),
        (8,3,2), (9,4,2), (10,2,3), (11,3,3), (12,4,3), (13,2,4);
```

The network created in *edge_table*



Pour une introduction plus complète sur comment construire une application de routage, lire l'Atelier pgRouting [3].

3. http ://workshop.pgrouting.org

CHAPITRE 3

Data Types

3.1 pgRouting Data Types

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

3.1.1 pgr_costResult[]

Nom

`pgr_costResult[]` — Un ensemble objets pour décrire un résultat de chemin avec un attribut coût.

Description

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

seq séquence ID indiquant l'ordre du chemin

id1 nom générique, à être spécifié par la fonction, typiquement l'id du noeud

id2 nom générique, à être spécifié par la fonction, typiquement l'id de l'arête

cost attribut coût

3.1.2 pgr_costResult3[] - Résultats du chemin multiple avec coût

Nom

`pgr_costResult3[]` — Un ensemble d'objets pour décrire un résultat de chemin avec un attribut coût.

Description

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

seq séquence ID indiquant l'ordre du chemin

id1 nom générique, à être spécifié par la fonction, typiquement l'id du chemin

id2 nom générique, à être spécifié par la fonction, typiquement l'id du noeud

id3 nom générique, à être spécifié par la fonction, typiquement l'id de l'arête

cost attribut coût

Histoire

– Nouveau depuis la version 2.0.0
– Remplace `path_result`

Voir aussi

– *Introduction*

3.1.3 pgr_geomResult[]

Nom

`pgr_geomResult[]` — Un ensemble d'enregistrements pour décrire un résultat de chemin avec un attribut géométrie.

Description

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

seq séquence ID indiquant l'ordre du chemin

id1 nom générique, à être spécifié par la fonction

id2 nom générique, à être spécifié par la fonction

geom attribut de géométrie

Histoire

– Nouveau depuis la version 2.0.0
– Remplace `geoms`

Voir aussi

– *Introduction*

Functions reference

4.1 Topology Functions

The pgRouting's topology of a network, represented with an edge table with source and target attributes and a vertices table associated with it. Depending on the algorithm, you can create a topology or just reconstruct the vertices table, You can analyze the topology, We also provide a function to node an unoded network.

4.1.1 pgr_createTopology

Nom

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

Synopsis

The function returns :
– `OK` after the network topology has been built and the vertices table created.
– `FAIL` when the network topology was not built due to an error.

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

Description

Parameters

La fonction de création de topologie accepte les paramètres suivants :

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

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

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

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

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

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

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

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

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

The Vertices Table

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

The structure of the vertices table is :

id `bigint` Identifier of the vertex.

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

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

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

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

the_geom `geometry` Point geometry of the vertex.

Histoire

- Renommé depuis la version 2.0.0

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

The simplest way to use pgr_createtopology is :

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

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

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

We get the same result AS the simplest way to use the function.

Warning :
An error would occur when the arguments are not given in the appropriate order : In this example, the column `id` of the table `ege_table` is passed to the function AS the geometry column, and the geometry column `the_geom` is passed to the function AS the id column.
`SELECT pgr_createTopology('edge_table',0.001,'id','the_geom','source','target');`
ERROR : Can not determine the srid of the geometry "id" in table public.edge_table

When using the named notation

The order of the parameters do not matter :

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

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the id.

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

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

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

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

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

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

For the following table

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

Using positional notation :

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

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

Warning :
An error would occur when the arguments are not given in the appropiriate order : In this example, the column `gid` of the table `mytable` is passed to the function AS the geometry column, and the geometry column `mygeom` is passed to the function AS the id column.
`SELECT pgr_createTopology('mytable',0.001,'gid','mygeom','src','tgt');`
ERROR : Can not determine the srid of the geometry "gid" in table public.mytable

When using the named notation

The order of the parameters do not matter :

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the id.

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

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

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

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

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

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

```
DROP TABLE IF EXISTS otherTable;
CREATE TABLE otherTable AS  (SELECT 100 AS gid, st_point(2.5,2.5) AS other_geom) ;
SELECT  pgr_createTopology('mytable',0.001,'mygeom','gid','src','tgt',
                    rows_where:='mygeom && (SELECT st_buffer(other_geom,1) FROM otherTable WHERE gi
```

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

Exemples

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

 pgr_createtopology
```

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

SELECT pgr_createTopology('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.0001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  --------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr

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

The example uses the *Données d'échantillon* network.

Voir aussi

– *Topologie de routage* for an overview of a topology for routing algorithms.
– *pgr_createVerticesTable* to reconstruct the vertices table based on the source and target information.
– *pgr_analyzeGraph* to analyze the edges and vertices of the edge table.

4.1.2 pgr_createVerticesTable

Name

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

Synopsis

The function returns :
– `OK` after the vertices table has been reconstructed.
– `FAIL` when the vertices table was not reconstructed due to an error.

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

Description

Parameters

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

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

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

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

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

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

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

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

The Vertices Table

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

The structure of the vertices table is :

id `bigint` Identifier of the vertex.

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

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

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

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

the_geom `geometry` Point geometry of the vertex.

History

- Renamed in version 2.0.0

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

The simplest way to use pgr_createVerticesTable is :

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

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

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

We get the same result as the simplest way to use the function.

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

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

When using the named notation

The order of the parameters do not matter :

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

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the id.

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

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

```
SELECT  pgr_createVerticesTable('edge_table',rows_where:='the_geom && (select st_buffer(the_geom,
```

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

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

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

For the following table

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

Using positional notation :

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

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

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

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

When using the named notation

The order of the parameters do not matter :

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the gid.

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

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

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

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

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

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

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

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

Examples

```
    SELECT pgr_createVerticesTable('edge_table');
    NOTICE:  PROCESSING:
NOTICE:  pgr_createVerticesTable('edge_table','the_geom','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Populating public.edge_table_vertices_pgr, please wait...
NOTICE:    ----->   VERTICES TABLE CREATED WITH  17 VERTICES
NOTICE:                                        FOR   18  EDGES
NOTICE:    Edges with NULL geometry,source or target: 0
NOTICE:                               Edges processed: 18
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ------------------------------------------------

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

The example uses the *Données d'échantillon* network.

See Also

– *Topologie de routage* for an overview of a topology for routing algorithms.
– *pgr_createTopology* to create a topology based on the geometry.
– *pgr_analyzeGraph* to analyze the edges and vertices of the edge table.
– *pgr_analyzeOneway* to analyze directionality of the edges.

4.1.3 pgr_analyzeGraph

Nom

`pgr_anlizeGraph` — Analyzes the network topology.

Synopsis

The function returns :
– `OK` after the analysis has finished.
– `FAIL` when the analysis was not completed due to an error.

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

Description

Prerequisites

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

Parameters

The analyze graph function accepts the following parameters :

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

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

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

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

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

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

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

The function returns :
– `OK` after the analysis has finished.
  – Uses the vertices table : <edge_table>_vertices_pgr.
  – Fills completly the `cnt` and `chk` columns of the vertices table.
  – Returns the analysis of the section of the network defined by `rows_where`
– `FAIL` when the analysis was not completed due to an error.
  – The vertices table is not found.
  – A required column of the Network table is not found or is not of the appropriate type.
  – The condition is not well formed.
  – The names of source , target or id are the same.
  – The SRID of the geometry could not be determined.

The Vertices Table

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

The structure of the vertices table is :

id `bigint` Identifier of the vertex.

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

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

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

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

the_geom `geometry` Point geometry of the vertex.

Histoire

– Nouveau depuis la version 2.0.0

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

The simplest way to use pgr_analyzeGraph is :

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

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

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

We get the same result as the simplest way to use the function.

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

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

ERROR : Can not determine the srid of the geometry "id" in table public.edge_table

When using the named notation

The order of the parameters do not matter :

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

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the id. Displays the analysis a the section of the network.

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 10');
```

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

```
SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='the_geom && (SELECT st_buffer(the_geom,0
```

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

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

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

For the following table

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

Using positional notation :

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

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

Warning :
An error would occur when the arguments are not given in the appropriate order : In this example, the column `gid` of the table `mytable` is passed to the function as the geometry column, and the geometry column `mygeom` is passed to the function as the id column.
`SELECT pgr_analyzeGraph('mytable',0.001,'gid','mygeom','src','tgt');`
ERROR : Can not determine the srid of the geometry "gid" in table public.mytable

When using the named notation

The order of the parameters do not matter :

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the id.

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

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

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

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

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

Selecting the rows WHERE the geometry is near the place='myhouse' of the table `othertable`. (note the use of quote_literal)

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

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

Exemples

```
SELECT  pgr_create_topology('edge_table',0.001);
SELECT pgr_analyzeGraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                            Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                      Ring geometries: 0

 pgr_analizeGraph
--------------------
 OK
(1 row)

SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id < 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id < 10')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 4
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                      Ring geometries: 0
```

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

SELECT  pgr_analyzeGraph('edge_table',0.001,rows_where:='id >= 10');
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','id >= 10')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                            Dead ends: 8
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                      Ring geometries: 0

 pgr_analizeGraph
--------------------
 OK
(1 row)

-- Simulate removal of edges
SELECT pgr_createTopology('edge_table', 0.001,rows_where:='id <17');
SELECT pgr_analyzeGraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 3
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                      Ring geometries: 0

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

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

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

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

The examples use the *Données d'échantillon* network.

Voir aussi

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

4.1.4 pgr_analyzeOneway

Nom

`pgr_analyzeOneway` — Analyser les routes à sens unique et identifier les segments marginaux.

Synopsis

Cette fonction analyse les routes à sens unique dans un graphe et identifie tout segment marginaux.

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

Description

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

So why do we care if the are not feasible ? Well if the direction of an edge was reversed by mistake we could generate exactly these conditions. Think about a divided highway and on the north bound lane one segment got

entered wrong or maybe a sequence of multiple segments got entered wrong or maybe this happened on a roundabout. The result would be potentially a *source* and/or a *sink* node.

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

Prerequisites

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

Parameters

edge_table `text` Network table name. (may contain the schema name as well)
s_in_rules Noeud source `text[]` **dans** rules
s_out_rules Noeud source `text[]` **hors de** rules
t_in_rules Noeud cible `text[]` **dans** rules
t_out_rules Noeud cible `text[]` **hors de** rules
oneway `text` oneway column name name of the network table. Default value is `oneway`.
source `text` Source column name of the network table. Default value is `source`.
target `text` Target column name of the network table. Default value is `target`.
two_way_if_null `boolean` flag to treat oneway NULL values as bi-directional. Default value is `true`.

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

The function returns :
– `OK` after the analysis has finished.
  – Uses the vertices table : <edge_table>_vertices_pgr.
  – Fills completely the `ein` and `eout` columns of the vertices table.
– `FAIL` when the analysis was not completed due to an error.
  – The vertices table is not found.
  – A required column of the Network table is not found or is not of the appropriate type.
  – The names of source , target or oneway are the same.

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

The Vertices Table

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

The structure of the vertices table is :

id `bigint` Identifier of the vertex.
cnt `integer` Number of vertices in the edge_table that reference this vertex. See *pgr_analyzeGgraph*.
chk `integer` Indicator that the vertex might have a problem. See *pgr_analyzeGraph*.
ein `integer` Number of vertices in the edge_table that reference this vertex as incoming.
eout `integer` Number of vertices in the edge_table that reference this vertex as outgoing.
the_geom `geometry` Point geometry of the vertex.

Histoire

– Nouveau depuis la version 2.0.0

Exemples

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

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

– *Topologie de routage* for an overview of a topology for routing algorithms.
– *Analytiques de graphe* for an overview of the analysis of a graph.
– *pgr_analyzeGraph* to analyze the edges and vertices of the edge table.
– *pgr_createVerticesTable* to reconstruct the vertices table based on the source and target information.

4.1.5 pgr_nodeNetwork

Nom

`pgr_nodeNetwork` - Noue une table réseau d'arêtes.

Author Nicolas Ribot

Synopsis

La fonction lit les arêtes d'une table réseau de nœuds non "noded" et écrit les arêtes "noded" dans une nouvelle table.

```
text pgr_nodenetwork(text edge_table, float8, tolerance,
                     text  id='id', text the_geom='the_geom',text table_ending='noded')
```

Description

Un problème commun associé avec des données SIG dans pgRouting est le fait que les données sont souvent non "noded" correctement. Cela créée des topologies invalides, qui résultent dans des routes incorrectes.

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

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

Parameters

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

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

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

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

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

The output table will have for `edge_table_noded`

id `bigint` Unique identifier for the table

old_id `bigint` Identifier of the edge in original table

sub_id `integer` Segment number of the original edge

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

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

the geom `geometry` Geometry column of the noded network

Histoire

– Nouveau depuis la version 2.0.0

Example

Let's create the topology for the data in *Données d'échantillon*

```
SELECT pgr_createTopology('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  --------------> TOPOLOGY CREATED FOR  18 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
pgr_createtopology
--------------------
 OK
(1 row)
```

Now we can analyze the network.

```
SELECT pgr_analyzegraph('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                            Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                      Ring geometries: 0
pgr_analyzegraph
------------------
 OK
(1 row)
```

The analysis tell us that the network has a gap and and an intersection. We try to fix the problem using :

```
SELECT pgr_nodeNetwork('edge_table', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_nodeNetwork('edge_table',0.001,'the_geom','id','noded')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Processing, pelase wait .....
NOTICE:    Splitted Edges: 3
NOTICE:   Untouched Edges: 15
NOTICE:       Total original Edges: 18
NOTICE:   Edges generated: 6
NOTICE:   Untouched Edges: 15
NOTICE:         Total New segments: 21
NOTICE:   New Table: public.edge_table_noded
NOTICE:  ---------------------------------
pgr_nodenetwork
-----------------
 OK
(1 row)
```

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

```
SELECT old_id,sub_id FROM edge_table_noded ORDER BY old_id,sub_id;
 old_id | sub_id
--------+--------
 1      |      1
 2      |      1
 3      |      1
 4      |      1
 5      |      1
 6      |      1
 7      |      1
 8      |      1
 9      |      1
 10     |      1
 11     |      1
 12     |      1
 13     |      1
 13     |      2
 14     |      1
 14     |      2
 15     |      1
 16     |      1
```

```
 17     |      1
 18     |      1
 18     |      2
(21 rows)
```

We can create the topology of the new network

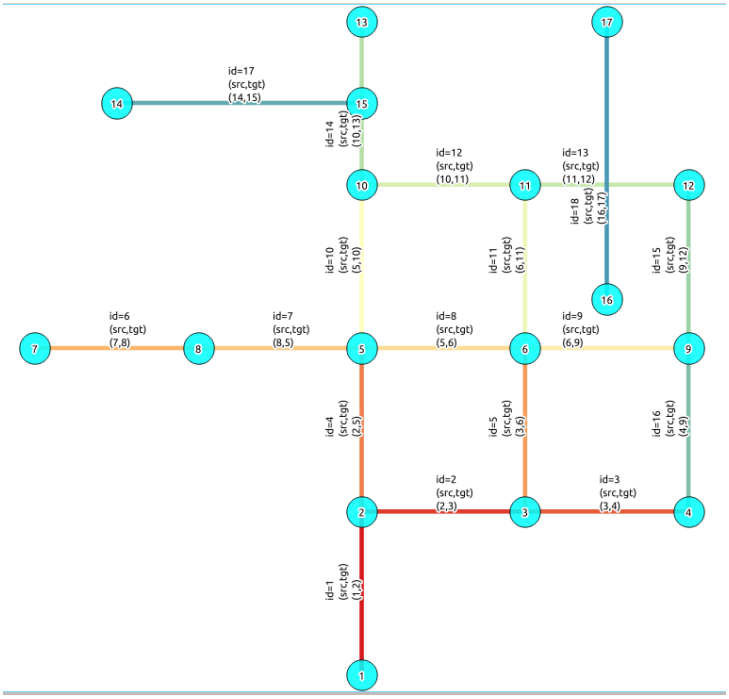
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED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

Now let's analyze the new topology

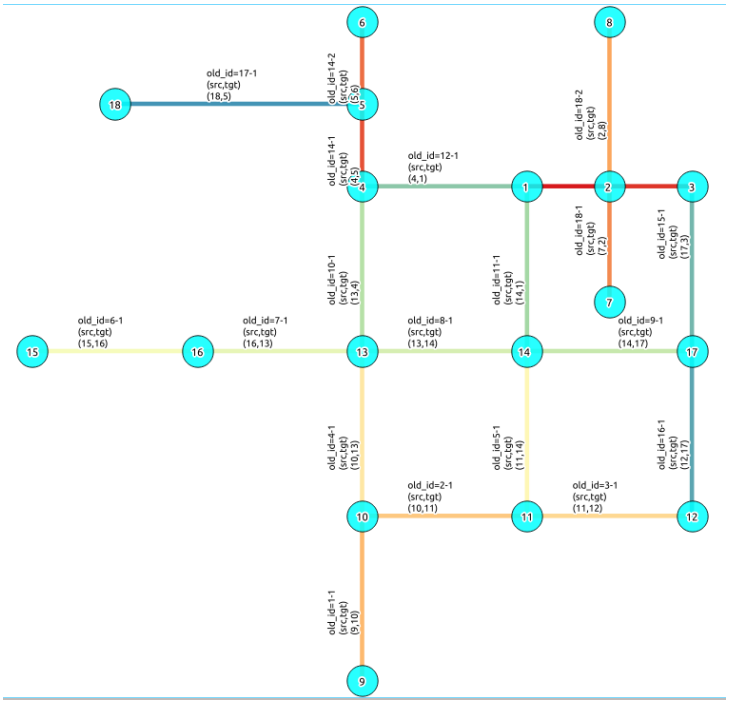
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...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

Images

Before Image



After Image



Comparing the results

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

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

Now, we are going to include the segments 13-1, 13-2 14-1, 14-2 ,18-1 and 18-2 into our edge-table, copying the data for dir,cost,and reverse cost with tho following steps :
– Add a column old_id into edge_table, this column is going to keep track the id of the original edge
– Insert only the segmented edges, that is, the ones whose max(sub_id) >1

```
alter table edge_table drop column if exists old_id;
alter table edge_table add column old_id integer;
insert into edge_table (old_id,dir,cost,reverse_cost,the_geom)
        (with
        segmented as (select old_id,count(*) as i from edge_table_noded group by old_id)
        select  segments.old_id,dir,cost,reverse_cost,segments.the_geom
                from edge_table as edges join edge_table_noded as segments on (edges.id = segments
                where edges.id in (select old_id from segmented where i>1) );
```

We recreate the topology :

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

NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED FOR  24 edges
NOTICE:  Rows with NULL geometry or NULL id: 0
NOTICE:  Vertices table for table public.edge_table is: public.edge_table_vertices_pgr
NOTICE:  ----------------------------------------------
pgr_createtopology
--------------------
OK
(1 row)
```

To get the same analysis results as the topology of edge_table_noded, we do the following query :

```
SELECT pgr_analyzegraph('edge_table', 0.001,rows_where:='id not in (select old_id from edge_table
```

```
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target',
                          'id not in (select old_id from edge_table where old_id is not null)')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 0
NOTICE:                            Dead ends: 6
NOTICE:  Potential gaps found near dead ends: 0
NOTICE:               Intersections detected: 0
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
OK
(1 row)
```

To get the same analysis results as the original edge_table, we do the following query :

```
SELECT pgr_analyzegraph('edge_table', 0.001,rows_where:='old_id is null')

NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table',0.001,'the_geom','id','source','target','old_id is null')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
NOTICE:  Analyzing for isolated edges. Please wait...
NOTICE:  Analyzing for ring geometries. Please wait...
NOTICE:  Analyzing for intersections. Please wait...
NOTICE:              ANALYSIS RESULTS FOR SELECTED EDGES:
NOTICE:                    Isolated segments: 2
NOTICE:                            Dead ends: 7
NOTICE:  Potential gaps found near dead ends: 1
NOTICE:               Intersections detected: 1
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
OK
(1 row)
```

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

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

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

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

Voir aussi

Topologie de routage for an overview of a topology for routing algorithms. *pgr_analyzeOneway* to analyze directionality of the edges. *pgr_createTopology* to create a topology based on the geometry. *pgr_analyzeGraph* to analyze the edges and vertices of the edge table.

4.2 Routing Functions

4.2.1 pgr_apspJohnson - Plus court chemin toutes paires, algorithme de Johnson

Nom

`pgr_apspJohnson` - Retourne tous les coûts pour chaque paire de noeuds dans le graphe.

Synopsis

L'algorithme de Johnson est un moyen de trouver les plus courts chemins entre toutes les paires de sommets dans un graphe épars, pondéré arête, dirigé. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost) pour chaque paire de nœuds dans le graphe.

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

Description

sql une requête SQL qui maintient les arêtes pour le graphe qui sera analysé :

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

source `int4` identifiant du sommet source pour cette arête

target `int4` identifiant du sommet cible pour cette arête

cost `float8` une valeur positive pour le coût pour traverser cette arête

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 ID noeud source

id2 ID nœud cible

cost coût pour traverser de `id1` en utilisant `id2`

Histoire

– Nouveau depuis la version 2.0.0

Exemples

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

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

La requête utilise le réseau *Données d'échantillon*.

Voir aussi

– *pgr_costResult[]*
– *pgr_apspWarshall - Plus court chemin toutes paires, Algorithme Floyd-Warshall*
– http ://en.wikipedia.org/wiki/Johnson%27s_algorithm

4.2.2 pgr_apspWarshall - Plus court chemin toutes paires, Algorithme Floyd-Warshall

Nom

`pgr_apspWarshall` - Retourne tous les coûts pour chaque paire de noeuds dans le graphe.

Synopsis

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

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

Description

sql une requête SQL qui maintient les arêtes pour le graphe qui sera analysé :

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

id `int4` identifiant de l'arête
source `int4` identifiant du sommet source pour cette arête
target `int4` identifiant du sommet cible pour cette arête
cost `float8` une valeur positive pour le coût pour traverser cette arête

directed `true` si le graphe est dirigé
reverse_cost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne
id1 ID noeud source
id2 ID nœud cible
cost coût pour traverser de `id1` en utilisant `id2`

Histoire

– Nouveau depuis la version 2.0.0

Exemples

```
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
[...]
```

La requête utilise le réseau *Données d'échantillon*.

Voir aussi

– *pgr_costResult[]*
– *pgr_apspJohnson - Plus court chemin toutes paires, algorithme de Johnson*
– http ://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

4.2.3 pgr_astar - Plus court chemin A*

Nom

`pgr_astar` — Retourne le plus court chemin en utilisant l'algorithme A*.

Synopsis

L'algorithme A* (prononcé "A Etoile") est basé sur l'algorithme de Dijkstra avec une heuristique qui autorise de résoudre la plupart des problèmes de plus court chemin par l'évaluation de seulement d'un sous-ensemble du graphe général. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost), qui fabriquent un chemin.

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

Description

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

x1 `x` coordonnée du point de départ de l'arête

y1 `y` coordonnée du point de départ de l'arête

x2 `x` coordonnée du point final de l'arête

y2 `y` coordonnée du point final de l'arête

reverse_cost (optionnel) le coût pour la traversée inverse de l'arête. Ceci est seulement utilisé quand les paramètres `directed` et `has_rcost` sont `true` (voir la remarque ci-dessus sur les coûts négatifs).

source `int4` id du point de départ

target `int4` id du point final

directed `true` si le graphe est dirigé

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 ID noeud

id2 ID arête (`-1` pour la dernière ligne)

cost coût pour traverser à partir de `id1` en utilisant `id2`

Histoire

– Renommé depuis la version 2.0.0

Exemples

– Sans `reverse_cost`

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

```
 seq | node | edge | cost
-----+------+------+------
   0 |    4 |   16 |    1
   1 |    9 |    9 |    1
   2 |    6 |    8 |    1
   3 |    5 |    4 |    1
   4 |    2 |    1 |    1
   5 |    1 |   -1 |    0

(4 rows)
```

– Avec `reverse_cost`

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

```
 seq | node | edge | cost
-----+------+------+------
   0 |    4 |    3 |    1
```

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

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

4.2.4 pgr_bdAstar - plus court chemin bidirectionnel A*

Nom

`pgr_bdAstar` - Retourne le plus court chemin en utilisant l'algorithme bidirectionnel A*.

Synopsis

Ceci est un algorithme de recherche bidirectionnel A*. Il recherche d'une source vers une destination et en même temps depuis la destination vers la source et se finit quand ces deux recherches se rencontrent au milieur. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost), qui fabriquent un chemin.

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

Description

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

x1 `x` coordonnée du point de départ de l'arête

y1 `y` coordonnée du point de départ de l'arête

x2 `x` coordonnée du point final de l'arête

y2 `y` coordonnée du point final de l'arête

reverse_cost (optionnel) le coût pour la traversée inverse de l'arête. Ceci est seulement utilisé quand les paramètres `directed` et `has_rcost` sont `true` (voir la remarque ci-dessus sur les coûts négatifs).

source `int4` id du point de départ

target `int4` id du point final

directed `true` si le graphe est dirigé

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 ID noeud

id2 ID arête (`-1` pour la dernière ligne)

cost coût pour traverser à partir de `id1` en utilisant `id2`

Warning : Vous devez vous reconnecter à la base de données après `CREATE EXTENSION pgrouting`. Sinon la fonction va retourner `Error computing path: std::bad_alloc`.

Histoire

– Nouveau depuis la version 2.0.0

Exemples

– Sans `reverse_cost`

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

```
 seq | node | edge | cost
-----+------+------+------
   0 |    4 |    3 |    0
   1 |    3 |    5 |    1
   2 |    6 |   11 |    1
   3 |   11 |   12 |    0
   4 |   10 |   -1 |    0
(5 rows)
```

– Avec `reverse_cost`

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

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

– *pgr_costResult[]*
– *pgr_bdDijkstra - Plus court chemin bidirectionnel Dijkstra*
– http ://en.wikipedia.org/wiki/Bidirectional_search
– http ://en.wikipedia.org/wiki/A*_search_algorithm

4.2.5 pgr_bdDijkstra - Plus court chemin bidirectionnel Dijkstra

Nom

`pgr_bdDijkstra` - Retourne le plus court chemin en utilisant l'algorithme bidirectionnel Dijkstra.

Synopsis

C'est un algorithme de recherche bidirectionnel Dijkstra. Cela recherche depuis la source vers la destination et en même temps depuis la destination vers la source et se termine quand ces deux recherches se rencontrent au milieu. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost), qui fabriquent un chemin.

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

Description

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

reverse_cost (optionnel) le coût pour la traversée inverse de l'arête. Ceci est seulement utilisé quand les paramètres `directed` et `has_rcost` sont `true` (voir la remarque ci-dessus sur les coûts négatifs).

source `int4` id du point de départ

target `int4` id du point final

directed `true` si le graphe est dirigé

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 ID noeud

id2 ID arête (`-1` pour la dernière ligne)

cost coût pour traverser depuis `id1` en utilisant `id2`

Histoire

– Nouveau depuis la version 2.0.0

Exemples

– Sans `reverse_cost`

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

 seq | node | edge | cost
-----+------+------+------
   0 |    4 |    3 |    0
   1 |    3 |    5 |    1
   2 |    6 |   11 |    1
   3 |   11 |   12 |    0
   4 |   10 |   -1 |    0
(5 rows)
```

– Avec `reverse_cost`

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

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

– *pgr_costResult[]*
– *pgr_bdAstar - plus court chemin bidirectionnel A**
– http ://en.wikipedia.org/wiki/Bidirectional_search
– http ://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

4.2.6 pgr_dijkstra - Plus court chemin Dijkstra

Nom

`pgr_dijkstra` — Retourne le plus court chemin en utilisant l'algorithme Dijkstra.

Synopsis

L'algorithme Dijkstra, conçu par l'informaticien Néerlandais Edsger Dijkstra en 1956. C'est un algorithme de recherche de graphe qui résout le problème de plus court chemin à source unique pour un graphe à coûts de chemin non négatifs, produisant un arbre de plus court chemin. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost) rows, qui fabriquent un chemin.

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

Description

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

reverse_cost `float8` (optional) the cost for the reverse traversal of the edge. This is only used when the `directed` and `has_rcost` parameters are `true` (see the above remark about negative costs).

source `int4` id du point de départ

target `int4` id du point final

directed `true` si le graphe est dirigé

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 ID noeud

id2 ID arête (`-1` pour la dernière ligne)

cost coût pour traverser à partir de `id1` en utilisant `id2`

Histoire

– Renommé depuis la version 2.0.0

Exemples

– Sans `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)
```

– Avec `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)
```

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

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

4.2.7 pgr_kDijkstra - Plus court chemin Dijkstra à multiples destinations

Nom

– `pgr_kdijkstraCost` - Retourne les coûts pour les K plus courts chemins utilisant l'algoithme Dijkstra.
– `pgr_kdijkstraPath` - Retourne les chemins pour les K plus courts chemins en utilisant l'algorithme Dijkstra.

Synopsis

Ces fonctions vous autorisent à avoir un unique nœud de départ et des nœuds de destination multiple et va calculer les routes pour toutes les destinations depuis le noeud source. Retourne un ensemble de *pgr_costResult3* ou *pgr_costResult3*. `pgr_kdijkstraCost` Retourne un enregistrement pour chaque nœud et le coût est le coût total de la route vers ce noeud. `pgr_kdijkstraPath` retourne un enregistrement pour chaque nœud dans ce chemin depuis la source vers la destination et le coût pour traverser cette arête.

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

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

Description

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

reverse_cost (optionnel) le coût pour la traversée inverse de l'arête. Ceci est seulement utilisé quand les paramètres `directed` et `has_rcost` sont `true` (voir la remarque ci-dessus sur les coûts négatifs).

source `int4` id du point de départ

targets `int4[]` un tableau d'ids des points finaux

directed `true` si le graphe est dirigé

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

`pgr_kdijkstraCost` retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 id source du chemin de sommets (cela va toujours être la source du point final dans la requête).

id2 id cible du sommet du chemin

cost coût pour traverser le chemin de `id1` à `id2`. Le coût sera de -1.0 si il n'y a pas de chemin pour cet id de sommet cible.

`pgr_kdijkstraPath` retourne un ensemble de *pgr_costResult3[] - Résultats du chemin multiple avec coût* :

seq séquence de ligne

id1 id chemin cible (identifie le chemin cible).

id2 path edge source node id

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

cost coût pour traverser l'arête ou -1.0 si il n'y a pas de chemin pour cette cible

Histoire

– Nouveau depuis la version 2.0.0

Exemples

– Retourne un résultat `cost`

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

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

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

 seq | path | node | edge | cost
-----+------+------+------+------
   0 |    4 |   10 |   12 |    1
   1 |    4 |   11 |   13 |    1
   2 |    4 |   12 |   15 |    1
   3 |    4 |    9 |   16 |    1
   4 |    4 |    4 |   -1 |    0
   5 |   12 |   10 |   12 |    1
   6 |   12 |   11 |   13 |    1
   7 |   12 |   12 |   -1 |    0
(8 rows)
```

– Retourne un résultat

```
SELECT id1 as path, st_astext(st_linemerge(st_union(b.the_geom))) as the_geom
  FROM pgr_kdijkstraPath(
                  'SELECT id, source, target, cost FROM edge_table',
                  10, array[4,12], false, false
            ) a,
            edge_table b
WHERE a.id3=b.id
GROUP by id1
ORDER by id1;

path |            the_geom
------+--------------------------------
    4 | LINESTRING(2 3,3 3,4 3,4 2,4 1)
   12 | LINESTRING(2 3,3 3,4 3)
(2 rows)
```

Il n'y a pas d'assurance que le résultat au-dessus sera ordonné dans la direction du flot de la route, càd : il peut être inversé. Vous aurez besoin de vérifier si `st_startPoint()` de la route est la même que la localisation du point de départ et si ça ne l'est pas, alors appelez `st_reverse()` pour inverser la direction de la route. Ce comportement est l'une des fonctions PostGIS `st_linemerge()` et `st_union()` et non pgRouting.

Voir aussi

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

4.2.8 pgr_ksp - K-plus court chemin

Nom

`pgr_ksp` — Retourne les "K" plus courts chemins.

Synopsis

L'algorithme de routage de K plus court chemin basé sur l'algorithme de Yen. "K" est le nombre de plus courts chemins souhaités. Retourne un ensemble de lignes *pgr_costResult3* (seq, id1, id2, id3, cost), qui fabriquent un chemin.

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

Description

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

reverse_cost (optionnel) le coût pour la traversée inverse de l'arête. Ceci est utilisé seulement quand le paramètre `has_rcost` est `true` (voir la remarque au-dessus sur les coûts négatifs).

source `int4` id du point de départ

target `int4` id du point final

paths `int4` bombre de routes alternatives

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

Retourne un ensemble de *pgr_costResult[]* :

seq sequence for ording the results

id1 ID route

id2 ID noeud

id3 ID arête (`0` pour la dernière ligne)

cost coût pour traverser de `id2` en utilisant `id3`

Code de base KSP ici : http ://code.google.com/p/k-shortest-paths/source.

Histoire

– Nouveau depuis la version 2.0.0

Exemples

– Sans `reverse_cost`

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

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

– Avec `reverse_cost`

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

 seq | route | node | edge | cost
-----+-------+------+------+------
   0 |     0 |    7 |    6 |    1
   1 |     0 |    8 |    7 |    1
   2 |     0 |    5 |    8 |    1
```

```
   3 |     0 |     6 |   11 |     1
   4 |     0 |    11 |   13 |     1
   5 |     0 |    12 |    0 |     0
   6 |     1 |     7 |    6 |     1
   7 |     1 |     8 |    7 |     1
   8 |     1 |     5 |    8 |     1
   9 |     1 |     6 |    9 |     1
  10 |     1 |     9 |   15 |     1
  11 |     1 |    12 |    0 |     0
(12 rows)
```

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

– *pgr_costResult3[] - Résultats du chemin multiple avec coût*
– http ://en.wikipedia.org/wiki/K_shortest_path_routing

4.2.9 pgr_tsp - Voyageur du commerce

Nom

– `pgr_tsp` - Retourne la meilleure route à partir d'un point de départ via une liste de nœuds.
– `pgr_tsp` - Retourne le meilleur ordre de route quand passée une matrice de distance.
– `pgr_makeDistanceMatrix` - Retourne une matrice de distance Euclidienne à partir de points fournis par le résultat sql.

Synopsis

Le problème du voyageur du commerce (TSP) demande la question suivante : étant donnée une liste de villes et les distances entre chaque paire de villes, quelle est la route la plus courte possible qui visite chaque ville exactement une fois et retourne à la ville originale ? Cet algorithme utilise un recuit simulé pour retourner une solution approximative de haute qualité. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost), qui constituent un chemin.

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

Retourne un ensemble de (seq integer, id1 integer, id2 integer, cost float8) qui est le meilleur ordre pour visiter les noeuds dans la matrice. `id1` est l'index dans la matrice de distance. `id2` est l'id du point à partir du sql.

Si aucun `end_id` est donné ou est -1 ou égal au start_id alors le résultat TSP est supposé être une boucle circulaire retournant au départ. Si `end_id` est fourni alors la route est supposée commencer et finir aux ids désignés.

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

Description

Avec distances euclidiennes

Le solveur TSP est basé sur l'ordonnancement des points en utilisant la distance (euclidienne) de ligne droite entre les noeuds. L'implémentation est utilisé un algorithme d'approximation qui est très rapide. Ce n'est pas une solution exacte, mais il est garanti qu'une solution est retournée après un certain nombre d'itérations.

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id int4 identifiant du sommet

x coordonnée x float8

y coordonnée y float8

start_id int4 id du point de départ

end_id int4 id du point final, c'est *OPTIONNEL*, si inclure la route est optimisée d'un point de départ à la fin, sinon c'est supposé que le départ et la fin sont le même point.

La fonction retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 index interne de la matrice de distance

id2 id du noeud

cost coût pour traverser du nœud courant au prochain nœud

Créer une matrice de distance

Pour les utilisateurs qui ont besoin d'une matrice de distance, nous avons une fonction simple qui prend le SQL dans sql comme décrit au-dessus et retourne avec dmatrix et ids.

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

La fonction retourne un enregistrement de dmatrix, ids :

dmatrix float8[][] une distance symétrique euclidienne basée sur sql.

ids integer[] un tableau d'ids comme ils sont ordonnés dans la matrice de distances.

Avec matrice de distance

Pour les utilisateurs, qui ne veulent pas utiliser les distances euclidiennes, nous fournissons aussi la capacité de passer une matrice de distances et retourne une liste ordonnée de nœuds pour le meilleur ordre pour visiter chacun. C'est à l'utilisateur de remplir complètement la matrice de distances.

matrix float[][] matrice de distances de points

start int4 index du point de départ

end int4 (optionnel) index du point final

Le noeud end est un paramètre optionnel, vous pouvez juste le laisser ainsi si vous voulez une boucle où start''est le dépôt et la route retourne au dépôt. Si vous incluez le paramètre ''end, nous optimisons le chemin de start à end et minimisons la distance de la route en incluant les points restants.

La matrice de distances est un tableau multidimensionnel PostgreSQL array type [1] qui doit être de taille N x N.

Le résultat sera de N enregistrements de [seq, id] :

seq séquence de ligne

id index dans la matrice

Histoire

– Renommé depuis la version 2.0.0
– dépendance GAUL supprimée depuis la version 2.0.0

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

Exemples

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

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

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

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

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

– Utiliser la matrice de distances (boucle A en partant de 1)

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

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

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

– Utiliser la matrice de distance (en partant de 1, jusqu'à 2)

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

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

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

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

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

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

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

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

4.2.10 pgr_trsp - Plus court chemin avec restriction de virage (TRSP)

Nom

`pgr_trsp` — Retourne le plus court chemin avec support pour les restrictions de virage.

Synopsis

Le plus court chemin avec restriction de virage (TRSP) est un algorithme de plus court chemin qui peut optionnellement prendre en compte les restrictions comme celles trouvées dans les réseaux routiers navigables réels. La performance est pratiquement aussi rapide que la recherche A* mais a beaucoup de fonctionnalités additionnelles puisque fonctionnant avec des arêtes plus qu'avec des nœuds du réseau. Retourne un ensemble de lignes *pgr_costResult* (seq, id1, id2, cost), qui constituent un chemin.

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

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

Description

L'algorithme de plus court chemin avec restriction de virage (TRSP) est similaire au *Algorithme Shooting Star* dans lequel vous pouvez spécifier les restrictions de virage.

La configuration TRSP est essentiellement la même que *Dijkstra shortest path* avec l'addition d'une table optionnelle de restrictions de virage. Cela rend plus facile l'ajout des restrictions de virage à un réseau routier, que d'essayer de les ajouter au Shooting Star où vous aviez à ajouter les mêmes arêtes plusieurs fois si vous étiez impliqués dans une restriction.

sql une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant de l'arête

source `int4` identifiant du sommet source

target `int4` identifiant du sommet cible

cost `float8` valeur, du coût de l'arête traversée. Un coût négatif va prévenir l'arête d'être insérée dans le graphe.

reverse_cost (optionnel) le coût pour la traversée inverse de l'arête. Ceci est seulement utilisé quand les paramètres `directed` et `has_rcost` sont `true` (voir la remarque ci-dessus sur les coûts négatifs).

source `int4` **NODE id** du point de départ

target `int4` **NODE id** du point final

directed `true` si le graphe est dirigé

has_rcost si `true`, la colonne `reverse_cost` du SQL générant l'ensemble des lignes va être utilisé pour le coût de la traversée de l'arête dans la direction opposée.

restrict_sql (optionnel) une requête SQL, qui devrait retourner un ensemble de lignes avec les colonnes suivantes :

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

to_cost `float8` coût de restriction de virage

target_id `int4` id cible

via_path `text` liste séparée par virgule d'arêtes dans l'ordre inverse de `rule`

Un autre variante de TRSP autorise de spécifier **EDGE id** de source et cible ensemble avec une fraction pour interpoler la position :

source_edge `int4` **EDGE id** d'une arête de départ

source_pos `float8` fraction de 1 définit la position de l'arête de départ

target_edge `int4` **EDGE id** de l'arête finale

target_pos `float8` fraction de 1 définit la position de l'arête finale

Retourne un ensemble de *pgr_costResult[]* :

seq séquence de ligne

id1 ID noeud

id2 ID arête (`-1` pour la dernière ligne)

cost coût pour traverser à partir de `id1` en utilisant `id2`

Histoire

– Nouveau depuis la version 2.0.0

Exemples

– Sans restrictions de virage

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

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

– Avec restrictions de virage

Les restrictions de virage requièrent des informations additionnelles, qui peuvent être stockées dans une table séparée :

```
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);
```

Ensuite une requête avec des restrictions de virage est créée comme :

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

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

– *pgr_costResult[]*

– *genindex*
– *search*

4.3 With Driving Distance Enabled

Driving distance related Functions

4.3.1 pgr_drivingDistance

Name

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

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

Synopsis

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

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

Description

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

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

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

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

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

source `int4` id of the start point

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

directed `true` if the graph is directed

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

Returns set of *pgr_costResult[]* :

seq row sequence

id1 node ID

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

cost cost to get to this node ID

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

History

– Renamed in version 2.0.0

Examples

– Without `reverse_cost`

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

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

– With `reverse_cost`

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

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

The queries use the *Données d'échantillon* network.

See Also

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

4.3.2 pgr_alphaShape

Nom

`pgr_alphashape` — Fonction Core pour le calcul de la forme alpha

Note : Cette fonction ne devrait pas être utilisée directement. Utilisez *pgr_drivingDistance* à la place.

Synopsis

Retourne un tableau avec des lignes (x, y) qui décrivent les sommets d'une forme alpha.

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

Description

sql `text` une requête SQL, qui doit retourner un ensemble de lignes avec les colonnes suivantes :

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

id `int4` identifiant du sommet

x coordonnée x `float8`

y coordonnée y `float8`

Retourne un enregistrement sommet pour chaque ligne :

x coordonnée x

y coordonnée y

Histoire

– Renommé depuis la version 2.0.0

Exemples

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

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

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

```
SELECT round(st_area(ST_MakePolygon(ST_AddPoint(foo.openline, ST_StartPoint(foo.openline))))::nume
from (select st_makeline(points order by id)  as openline from
```

```
(SELECT st_makepoint(x,y) as points ,row_number() over() AS id
FROM pgr_alphAShape('SELECT id, x, y FROM vertex_table')
) as a) as foo;

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


SELECT * FROM pgr_alphAShape('SELECT id::integer, st_x(the_geom)::float as x, st_y(the_geom)::floa
  x   |  y
------+-----
 0.5  | 3.5
   0  |   2
   2  |   0
   4  |   1
   4  |   2
   4  |   3
 3.5  |   4
   2  |   4
(8 rows)

SELECT round(st_area(ST_MakePolygon(ST_AddPoint(foo.openline, ST_StartPoint(foo.openline))))::nume
from (select st_makeline(points order by id)  as openline from
(SELECT st_makepoint(x,y) as points ,row_number() over() AS id
FROM pgr_alphAShape('SELECT id::integer, st_x(the_geom)::float as x, st_y(the_geom)::float as y
) as a) as foo;

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

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

4.3.3 pgr_pointsAsPolygon

Nom

`pgr_pointsAsPolygon` — Dessine une forme alpha à partir d'un ensemble de points.

Synopsis

Retourner la forme alpha comme polygone geometry.

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

Description

sql `text` une requête SQL, qui doit retourner un ensemble de lignes avec les colonnes suivantes :

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

id int4 identifiant du sommet

x coordonnée x float8

y coordonnée y float8

Retourne une géométrie polygone.

Histoire

– Renommé depuis la version 2.0.0

Exemples

In the following query there is not way to control which point in the polygon is the first in the list, so you may get similar but different results than the following which are also correct. Each of the pgr_pointsAsPolygon queries below is followed by one the compute the area of the polygon. This area should remain constant regardles of the order of the points making up the polygon.

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

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

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

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


SELECT ST_ASText(pgr_pointsASPolygon('SELECT id::integer, st_x(the_geom)::float as x, st_y(the_geo
                                      FROM edge_table_vertices_pgr'));
                    st_astext
-----------------------------------------------------------
 POLYGON((0.5 3.5,0 2,2 0,4 1,4 2,4 3,3.5 4,2 4,0.5 3.5))
(1 row)

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

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

Les requêtes utilisent le réseau *Données d'échantillon*.

Voir aussi

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

Indices and tables

– *genindex*
– *search*

4.4 Developers's Functions

4.4.1 pgr_getColumnName

Name

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

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

Synopsis

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

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

Description

Parameters

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

col `text` column name to be retrived.

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

History

– New in version 2.0.0

Examples

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

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

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

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

The queries use the *Données d'échantillon* network.

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_isColumnInTable* to check only for the existance of the column.
– *pgr_getTableName* to retrieve the name of the table as is stored in the postgres administration tables.

4.4.2 pgr_getTableName

Name

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

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

Synopsis

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

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

Description

Parameters

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

Returns

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

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

History

– New in version 2.0.0

Examples

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

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

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

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

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

The examples use the *Données d'échantillon* network.

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_isColumnInTable* to check only for the existance of the column.
– *pgr_getTableName* to retrieve the name of the table as is stored in the postgres administration tables.

4.4.3 pgr_isColumnIndexed

Name

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

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

Synopsis

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

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

Description

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

col `text` Column name to be checked for.

Returns :

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

History

– New in version 2.0.0

Examples

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

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

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

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

The example use the *Données d'échantillon* network.

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_isColumnInTable* to check only for the existance of the column in the table.
– *pgr_getColumnName* to get the name of the column as is stored in the postgres administration tables.
– *pgr_getTableName* to get the name of the table as is stored in the postgres administration tables.

4.4.4 pgr_isColumnInTable

Name

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

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

Synopsis

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

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

Description

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

col `text` Column name to be checked for.

Returns :

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

History

– New in version 2.0.0

Examples

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

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

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

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

The example use the *Données d'échantillon* network.

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_isColumnIndexed* to check if the column is indexed.
– *pgr_getColumnName* to get the name of the column as is stored in the postgres administration tables.
– *pgr_getTableName* to get the name of the table as is stored in the postgres administration tables.

4.4.5 pgr_pointToId

Name

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

Note : This function is intended for the developer's aid. Use *pgr_createTopology* or *pgr_createVerticesTable* instead.

Synopsis

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

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

Description

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

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

vertname `text` Vertices table name WITH schema included.

srid `integer` SRID of the geometry point.

This function returns the id of the row that corresponds to the `point` geometry
– When the `point` geometry already exists in the vertices table `vertname`, it returns the corresponing `id`.
– When the `point` geometry is not found in the vertices table `vertname`, the funcion inserts the `point` and returns the corresponing `id` of the newly created vertex.

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

History

– Renamed in version 2.0.0

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_createVerticesTable* to create a topology based on the geometry.
– *pgr_createTopology* to create a topology based on the geometry.

4.4.6 pgr_quote_ident

Name

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

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

Synopsis

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

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

Description

Parameters

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

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

History

– New in version 2.0.0

Examples

Everything is lower case so nothing needs to be quoted.

```
SELECT pgr_quote_ident('the_geom');

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


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

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

The column is upper case so its double quoted.

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

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

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

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

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

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

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

Ignores extra `.` separators.

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

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

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_getTableName* to get the name of the table as is stored in the postgres administration tables.

4.4.7 pgr_version

Name

`pgr_version` — Query for pgRouting version information.

Synopsis

Returns a table with pgRouting version information.

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

Description

Returns a table with :

version `varchar` pgRouting version

tag `varchar` Git tag of pgRouting build

hash `varchar` Git hash of pgRouting build

branch `varchar` Git branch of pgRouting build

boost `varchar` Boost version

History

– New in version 2.0.0

Examples

– Query for full version string

```
SELECT pgr_version();
```

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

– Query for `version` and `boost` attribute

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

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

See Also

– *pgr_versionless* to compare two version numbers

4.4.8 pgr_versionless

Name

`pgr_versionless` — Compare two version numbers.

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

Synopsis

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

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

Description

v1 `text` first version number

v2 `text` second version number

History

– New in version 2.0.0

Examples

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

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

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_version* to get the current version of pgRouting.

4.4.9 pgr_startPoint

Name

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

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

Synopsis

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

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

Description

Parameters

geom `geometry` Geometry of a MULTILINESTRING or LINESTRING.

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

History

– New in version 2.0.0

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_endPoint* to get the end point of a (multi)linestring.

4.4.10 pgr_endPoint

Name

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

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

Synopsis

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

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

Description

Parameters

geom `geometry` Geometry of a MULTILINESTRING or LINESTRING.

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

History

– New in version 2.0.0

See Also

– *Guide du développeur* for the tree layout of the project.
– *pgr_startPoint* to get the start point of a (multi)linestring.

4.5 Fonctions antérieures

La sortie de pgRouting 2.0 a complètement restructuré le nommage des fonctions et rendu obsolète nombre de fonctions qui étaient disponibles dans les versions 1.x. Même si nous réalisons que cela peut être gênant pour nos utilisateurs actuels, nous considérons que cela est nécessaire pour la viabilité du projet sur le long terme pour être plus une solution à nos utilisateurs et d'être en mesure d'ajouter de nouvelles fonctionnalités et tester les fonctionnalités existantes.

Nous avons fait un effort minimum pour sauver la plupart de ces fonctions et les distribuer avec la version dans un fichier `pgrouting_legacy.sql` qui n'est pas une partie de l'extension pgrouting et n'est pas supportée. Si vous pouvez utiliser ces fonctions, c'est bien. Nous n'avons pas testé toutes ces fonctions donc si vous trouvez des problèmes et voulez poster une requête ou un patch pour aider les autres utilisateurs c'est très bien, mais il est probable que ce fichier soit supprimé dans une future version et nous recommandons que vous convertissiez votre code existant pour utiliser les nouvelles fonctions documentées et supportées.

La liste suivant est une liste des TYPEs, CASTs et FUNCTION inclus dans le fichier `pgrouting_legacy.sql`. La liste est fournie pour commodité mais ces fonctions sont déconseillées, non supportées, et probablement ont besoin de certaines modifications pour fonctionner.

4.5.1 TYPEs & CASTs

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

4.5.2 FUNCTIONs

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

4.6 Fonctions discontinues

Particulièrement avec les nouvelles versions majeures, des fonctionnalités peuvent changer et des fonctions peuvent être retirées pour diverses raisons. Toute fonctionnalité qui a été retirée est listée ici.

4.6.1 Algorithme Shooting Star

Version Supprimé depuis 2.0.0

Reasons Bugs non résolus, maintenance non assurée, remplacé par *pgr_trsp - Plus court chemin avec restriction de virage (TRSP)*

Comment Merci de *nous contacter* si vous êtes intéressés pour sponsoriser ou maintenir cet algorithme. La signature de fonction est toujours disponible dans *Fonctions antérieures* mais c'est juste une enveloppe qui lance une erreur. Nous avons inclus tout le code d'origine dans cette version.

CHAPITRE 5

Développeur

5.1 Guide du développeur

Note : Toute la documentation devrait être dans le format reStructuredText. Voir : <http ://docutils.sf.net/rst.html> pour les docs d'introduction.

5.1.1 Disposition de l'Arbre des sources

cmake/ Scripts cmake utilisés en partie pour notre système de build.

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

core/astar/ Ceci est une implémentation de la Recherche A* basée sur l'utilisation de la bibliothèque Boost Graph. Ceci est une implémentation du plus court chemin Dijkstra avec une Heuristique Euclidienne.

core/common/ Pour le moment ceci n'a pas de core in "src", mais a un certain nombre de codes de wrappers SQL et de codes de topologie dans le répertoire "sql". *Les wrappers spécifiques d'algorithmes devraient être déplacées à l'arbre d'algorithme et les tests appropriés devraient être ajoutés pour valider les wrappers.*

core/dijkstra/ Ceci est une implémentation de la solution du plus court chemin Dikjstra utilisant les bibliothèques Boost Graph pour l'implémentation.

core/driving_distance/ Ce paquet optionnel créée des polygones de driving distance basées sur la résolution de la solution du plus court chemin Dijkstra, ensuite en créant des polygones basés sur les distances de coût égal depuis le point de départ. Ce paquet optionnel requiert d'avoir installé les librairies CGAL.

core/shooting_star/ *DESAPPROUVE et NE FONCTIONNE PAS et SUR LE POINT D'ETRE SUPPRIME* Ceci est une arête basée sur l'algorithme du plus court chemin qui supporte les restrictions de virage. It is based on Boost Graph. Ne *PAS* utiliser cet algorithme comme il est cassé, utiliser *trsp* à la place, qui a la même fonctionnalité et est plus rapide et donne des résultats corrects.

core/trsp/ Ceci est un algorithme du plus court chemin avec restrictions de virage. Il a des caractéristiques sympathiques comme vous pouvez spécifiez les points de début et la fin comme un pourcentage d'une arête. Les restrictions sont sauvegardées dans une table séparée des arêtes du graphe et cela rend plus facile de gérer les données.

core/tsp/ Ce paquet optionnel fournit la capacité de calculer les solutions du problème du voyageur de commerce et calcule la route résultante. Ce paquet optionnel requiert l'installation des librairies GAUL.

tools/ Scripts divers et outils.

lib/ Ceci est le répertoire de sortie où les cibles des bibliothèques compilées et des installations sont présentées avant l'installation.

5.1.2 Mise en page de la documentation

Comme mentionné ci-dessus toute la documentation devrait être construite en utilisant les fichiers formattés re-StructuredText.

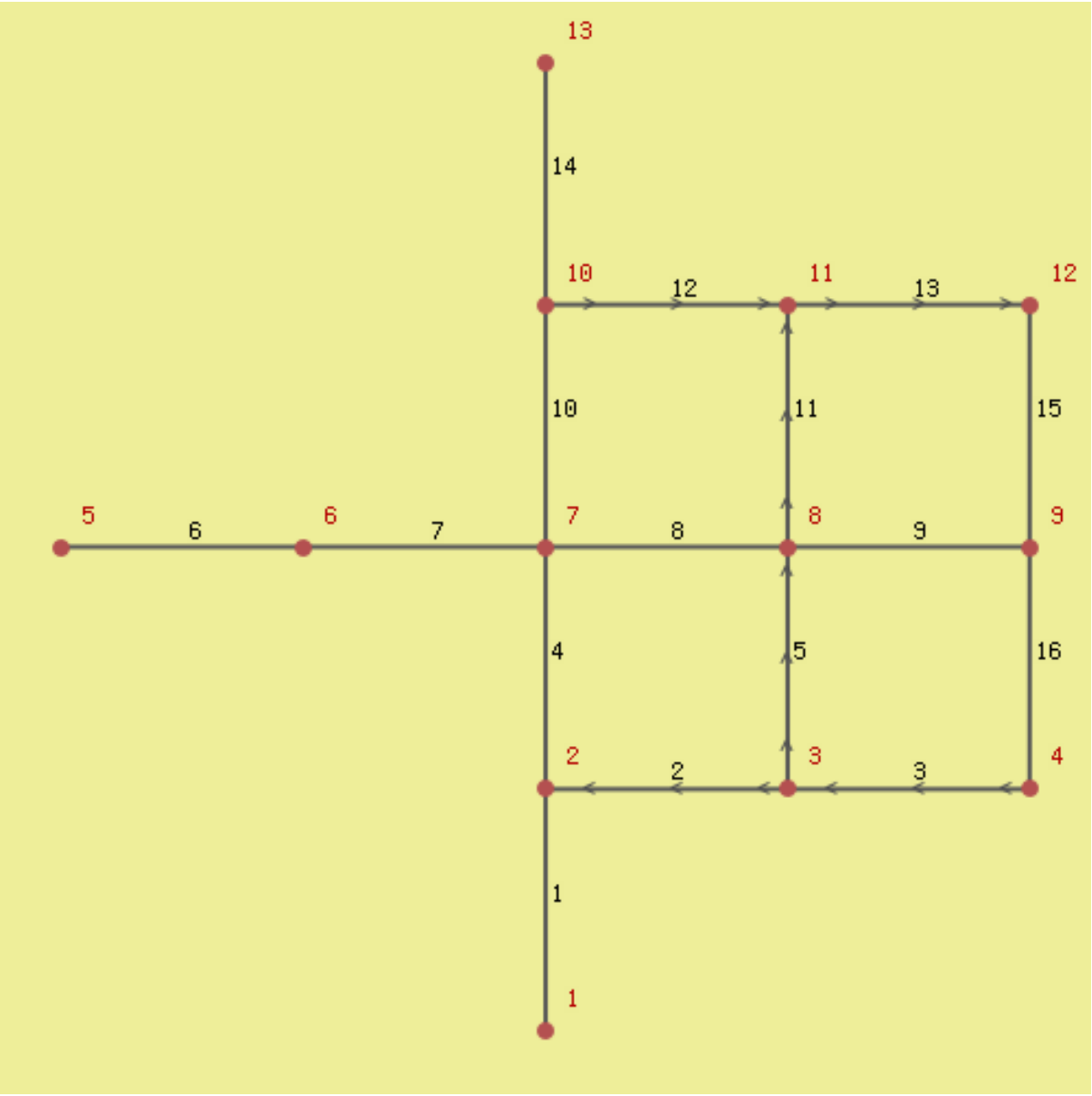
La documentation est distribuée dans l'arbre des sources. Ce répertoire "doc" de haut niveau est prévu pour la documentation de haut niveau couvrant les sujets comme :
– Compilation et tests
– Installation
– Tutoriels
– Documentation liminaire du Guide Utilisateur
– Documentation liminaire du Manuel de référence
– etc
Puisque la documentation spécifique de l'algorithme est contenue dans l'arbre de sources avec les fichiers spécifiques de l'algorithme, le processus de la construction de la documentation et la publication va avoir besoin d'assembler les détails avec la documentation de premier rang si besoin.

Aussi, pour empêcher le répertoire "doc" d'être encombré, chaque libre majeur comme ceux listés ci-dessus, doivent être contenu dans un répertoire séparé dans "doc". Toutes les images ou autres documents liés au livre doivent être aussi mis dans ce répertoire.

5.1.3 Infrastructure de test

L'infrastructure de test mise en place est très basique. Voici les bases de comment ça marche. Nous avons besoin de plus de cas de tests. A long terme nous devrons probablement avoir une personne pour installer les frameworks travis-ci ou jenkins.

Voici le graphe des tests TRSP.



Les tests sont lancés via le script au haut niveau tools/test-runner.pl qui exécute tous les tests configurés et pour le moment produit en résultats la structure de test. Cela peut être amélioré par la suite.

> Cela suppose aussi que vous avez installé les librairies car les tests requièrent postgresql installé. Il est probablement nécessaire de rendre cela mieux conçu pour que nous puissions essayer l'arbre de build. Je vais essayer de réfléchir à ça.

Simplement chaque répertoire .../test/ doit inclure un fichier *test.conf* qui est un fragment de script perl qui définit quels fichiers de données charger et quels tests lancer. J'ai intégré certains mécanismes pour autoriser tests et données d'être spécifiques aux version pg et postgis, mais je ne l'utilise pas encore. Ainsi par exemple, *core/trsp/test/test-any-00.data* est un texte plein généré qui va charger les données nécessaires pour un jeu de tests. C'est aussi le graphe dans l'image de dessus. Vous pouvez spécifier plusieurs fichiers à charger, mais comme un groupe ils ont besoin d'avoir des noms uniques.

core/trsp/test/test-any-00.test est une commande sql qui peut être exécutée. Elle s'exécute ainsi :

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

Ensuite si il y a une différence, alors un échec de test est reporté.

5.2 Release Notes

5.2.1 Notes de version pgRouting 2.0

Avec la version de pgRouting 2.0 la librairie a abandonné la compatibilité restrospective aux versions *pgRouting 1.x*. Nous avons fait ça pour que nous puissions restructurer pgRouting, standardiser le nommage de fonction, et préparer le projet pour un développement future. Comme un résultat de cet effort, nous avons été capable de simplifier pgRouting, ajouter de façon significative de nouvelles fonctionnalités, intégrer de la documentation et tester dans l'arbre source et rendre plus facile pour les multiples développeurs de contribuer.

Pour les changements importants voir les notes de versions suivantes. Pour voir la liste complète des changements, vérifiez la liste de *Git commits <https ://github.com/pgRouting/pgrouting/commits>* sur Github.

Changements pour la version 2.0.0

- Graph Analytics - outils pour la détection et la résolution certains problèmes de connexion dans un graphe.
- Une collection de fonctions utiles.
- Deux nouveaux algorithmes de plus court chemin toutes paires (pgr_apspJohnson, pgr_apspWarshall)
- Algorithmes de recherches bidirectionnels Dijkstra et A-star (pgr_bdAstar, pgr_bdDijkstra)
- Recherche à nœuds un à plusieurs (pgr_kDijkstra)
- Plus court chemin K chemins alternatifs (pgr_ksp)
- Nouveau solveur TSP qui simplifie le code et le processus de build (pgr_tsp), supprimé la dépendance "Gaul Library"
- Le plus court chemin à virage restreint (pgr_trsp) qui remplace Shooting Star
- Support supprimé pour Shooting Star
- Construit une infrastructure de test qui est exécutée avant que des changements de code majeurs soient enregistrés.
- Testé et résolu la plupart des bugs non résolus rapportés sur la 1.x et existant dans la base de code 2.0-dev.
- Processus de build amélioré pour Windows
- Automated testing on Linux and Windows platforms trigger by every commit
- Conception en librairie modulaire
- Compatibilité avec PostgreSQL 9.1 ou plus récent
- Compatibilité avec PostGIS 2.0 ou plus récent
- Installe comme une EXTENSION PostgreSQL
- Retourne les types remaniés et unifiés
- Support pour la table SCHEMA dans les paramètres de fonction
- Support pour préfixe de fonction `st_` PostGIS
- Ajouté préfixe `pgr_` aux fonctions et types
- Meilleure documentation : http ://docs.pgrouting.org

5.2.2 Notes de version pgRouting 1.0

Les notes de version suivantes ont été copiées depuis le fichier précédent `RELEASE_NOTES` et sont gardées comme une référence. Les notes de version à partir de *version 2.0.0* vont suivre un schéma différent.

Changements pour la version 1.05

- Résolutions de bug

Changements pour la version 1.03

- Création de topologie beaucoup plus rapide
- Résolutions de bug

Changements pour la version 1.02

– Résolutions de bug
– Problèmes de compilation résolus

Changements pour la version 1.01

– Résolutions de bug

Changements pour la version 1.0

– Fonctions principales et extra sont séparées
– Processus build Cmake
– Résolutions de bug

Changements pour la version 1.0.0b

– Fichier SQL additionnel avec des noms plus simples pour les fonctions de wrapper
– Résolutions de bug

Changements pour la version 1.0.0a

– Algorithme de plus court chemin Shooting* pour les réseaux routiers réels
– Plusieurs bugs SQL ont été résolus

Changements pour la version 0.9.9

– Support PostgreSQL 8.2
– Les fonctions de plus court chemin retournent un résultat vide s'ils ne pouvaient pas trouver un chemin

Changements pour la version 0.9.8

– Schéma de renumérotation a été ajouté aux fonctions de plus court chemin
– Les fonctions de plus court chemin ont été ajoutées
– routing_postgis.sql a été modifié pour utiliser Dijkstra dans la recherche TSP

Index et tableaux

– *genindex*
– *search*