pgRouting Manual

Release 2.0.0 (d4d49b7 master)

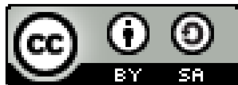
pgRouting Contributors

20 de September de 2013

Índice general

pgRouting extiende a las bases de datos geoespaciales PostGIS[1] /PostgreSQL[2] para proveer ruteo geoespacial y funcionalidad de análisis de redes.

Este es el manual para pgRouting 2.0.0 (d4d49b7 master).



El Manual de pgRouting está bajo la licencia Licencia Creative Commons Attribution-Share Alike 3.0[3]. Eres libre para usar este material de la manera que desees, pero pedimos que le otorges el crédito al proyecto pgRouting y cuando sea posible le pongas una liga hacia http://pgrouting.org. Para otras licencias usadas en pgRouting ver la página *License*.

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

CAPÍTULO 1

Generalidades

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 Instalación

Paquetes binarios se proporcionan para la versión actual en las siguientes plataformas:

1.2.1 Windows

Construcciones experimentales Winnie Bot:

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

1.2.2 Ubuntu/Debian

Paquetes de Ubuntu están disponibles en los repositorios de Launchpad:

- *estable* https://launchpad.net/~georepublic/+archive/pgrouting
- *inestable* 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
```

Usar el PPA de UbuntuGIS-unstable[14] para instalar PostGIS 2.0.

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

1.2.3 RHEL/CentOS/Fedora

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

1.2.4 OS X

- Homebrew

```
brew install pgrouting
```

1.2.5 Paquete fuente

Git 2.0.0-rc1 release	v2.0.0-rc1.tar.gz[15]	v2.0.0-rc1.zip[16]
Versión Git 2.0.0-beta	v2.0.0-beta.tar.gz[17]	v2.0.0-beta.zip[18]
Versión Git 2.0.0-alpha	v2.0.0-alpha.tar.gz[19]	v2.0.0-alpha.zip[20]
Rama principal de Git	master.tar.gz[21]	master.zip[22]
Rama de desarrollo en Git	develop.tar.gz[23]	develop.zip[24]

1.2.6 Usando Git

Protocolo de Git (solo lectura):

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

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

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

Ver *Guía de Compilación* para notas sobre la compilación desde la fuente.

1.3 Guía de Compilación

Para poder compilar pgRouting asegúrese de que se cumplan las siguientes dependencias:

- Compiladores de C y C++
- Postgresql versiones >= 8.4 (>= 9.1 recomendado)
- PostGIS version >= 1.5 (>= 2.0 recommended)
- La biblioteca de gráficos de Boost (BGL). Versión > = [por determinarse]

[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

- CMake >= 2.8.8
- (opcional, para la distancia de manejo) CGAL > = [por determinarse]
- (opcional, para la documentación) Sphinx > = 1.1
- (optional, for Documentation as PDF) Latex >= [TBD]

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

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

Currently these are:

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

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

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

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

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

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

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

1.3.1 Para MinGW en Windows

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

1.3.2 Para Linux

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

1.3.3 Con la documentación

Construir con documentación (requiere Sphinx[25]):

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

Reconstrucción de la documentación modificada solamente:

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

1.4 Soporte

Ayudas comunitarias de pgRouting son a través del website[26], documentation[27], tutoriales, listas de correo y otros. Si usted está buscando *apoyo comercial*, a continuación encontrará una lista de las empresas que prestan servicios de consultoría y de desarrollo para pgRouting.

1.4.1 Informes de problemas

Errores son registrados y manejados en un issue tracker[28]. Por favor siga los siguientes pasos:

1. Buscar en las entradas para ver si ya se ha informado de su problema. Si es así, añadir cualquier contexto extra que usted haya encontrado, o al menos indicar que también están teniendo el problema. Esto nos ayudará a priorizar los problemas comunes.
2. Si el problema no está reportado , crear una nueva entrada[29] para el problema.
3. En su informe incluya instrucciones explícitas para reproducir el problema. Las mejores entradas incluyen consultas SQL exactas que se necesitan para reproducir el problema.
4. Si usted puede probar las versiones anteriores de PostGIS para su problema, por favor hágalo. En su reporte, tenga en cuenta la versión más antigua en la que aparezca el problema.
5. Para las versiones donde se puede reproducir el problema, anote el sistema operativo y la versión de pgRouting, PostGIS y 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 Listas de correo y GIS StackExchange

Hay dos listas de correo para pgRouting alojados en las listas de correo del Servidor de OSGeo:

- Lista de correos para usuarios: http://lists.osgeo.org/mailman/listinfo/pgrouting-users
- Lista de correo para desarrolladores: http://lists.osgeo.org/mailman/listinfo/pgrouting-dev

Para preguntas generales y tópicos sobre el uso pgRouting, escriba a la lista de correo para usuario.

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

También se puede preguntar en StackExchange GIS[30] y etiquetar la pregunta con `pgRouting`. Buscar todas las preguntas con la etiqueta `pgRouting` bajo http://gis.stackexchange.com/questions/tagged/pgrouting o suscribirse al Alimentador de preguntas de pgRouting[31].

1.4.3 Soporte comercial

Para usuarios que requieren apoyo profesional, desarrollo y servicios de consultoría, considere ponerse en contacto con cualquiera de las siguientes organizaciones, que han contribuido significativamente al desarrollo de pgRouting:

Empresa	Oficinas en	Sitio web
Georepublic	Alemania, Japón	http://georepublic.info
iMaptools	Estados Unidos	http://imaptools.com
Orkney Inc.	Japón	http://www.orkney.co.jp
Camptocamp	Suiza, Francia	http://www.camptocamp.com

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

CAPÍTULO 2

Tutoriales

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 *Análisis de gráficas* 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 Topología para rutas

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

Copyright Stephen Woodbridge. El código fuente está liberado bajo la licencia MIT-X.

Resumen

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.

Puede utilizar las *funciones de análisis de gráficas* para apoyarse en la búsqueda de problemas topológicos en los datos. Si se necesitan nodos en los datos, también se cuenta con la función *pgr_nodeNetwork()* que le puede ser útil. Esta función divide todos los segmentos de los caminos y les asigna nodos entre ellos. Existen algunos casos donde no es lo apropiado.

Por ejemplo, cuando se tiene un cruce de puente o de un túnel, no quieres estos contengan un nodo, pero pgr_nodeNetwork no sabe que ese es el caso y va a crear un nodo, entonces el enrutador interpretará el puente o el paso a desnivel como si fuera una intersección plana en 2D. Para lidiar con este problema, se debe utilizar niveles z ya se para estos tipos de intersecciones o para otros casos en donde crear el nodo de la intersección no sea lo correcto.

Para los casos donde la topología debe construirse, las siguientes funciones pueden ser de utilidad. Una forma de preparar los datos para pgRouting es agregar los siguientes campos a la tabla y luego poblarlas según corresponda. Este ejemplo tiene como supuestos: que la tabla original ya tiene columnas como `one_way`, `fcc` y posiblemente otras y que contienen valores de datos específicos. Esto es sólo para darle una idea de lo que se puede hacer con los datos.

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

[1] woodbri@swoodbridge.com

```
    ADD COLUMN isolated integer;

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

La función *pgr_createTopology()* creará la tabla `vertices_tmp` y rellenará las columnas `source` y `target`. El ejemplo siguiente termina de llenar las columnas restantes. En este ejemplo, la columna `fcc` contiene códigos de las características de la calle y las declaraciones del `CASE` las convierte a una velocidad media.

```
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;
```

Ahora su base de datos debe estar lista para usarse en cualquiera (mayoría?) de los algoritmos de pgRouting.

Véase también

- *pgr_createTopology*
- *pgr_nodeNetwork*
- *pgr_pointToId*

2.1.3 Análisis de gráficas

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

Copyright Stephen Woodbridge. El código fuente está liberado bajo la licencia MIT-X.

Resumen

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.

Analizar un gráfico

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

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

In the vertices table "mytab_vertices_pgr":

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

[2] woodbri@swoodbridge.com

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

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

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

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

Si quieren visualizar en una imagen gráfica, entonces usted puede utilizar algo como MapServer para representar los bordes y los vértices y el estilizar en base a `cnt` o si están aislados, etc. También se puede hacer esto con una herramienta como graphviz o geoserver u otras herramientas similares.

Analizar las calles unidireccionales

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.

Esta consulta agrega dos columnas a la tabla vertices_tmp `ein int` y `eout int` (ein = borde de entrada, eout=borde de salida) y las rellena con los conteos correspondientes. Después de su ejecución, en un grafo se pueden identificar los nodos con problemas potenciales utilizando la siguiente consulta.

Las reglas se definen como un conjunto de cadenas de texto que si coinciden con el valor de `col` se considera como válido para el origen o destino dentro o fuera de la condición.

Ejemplo

Supongamos que tenemos una tabla "st" de bordes y una columna "one_way" que podría tener valores como:

- 'FT' - dirección unidireccional de la fuente para el nodo de destino.
- 'TF' - dirección unidireccional desde el nodo destino hasta el nodo fuente.
- 'B' - calle de doble sentido.
- ''- campo vacío, suponer doble sentido.
- <NULL> - Campo NULL, usar bandera de two_way_if_null, es decir, doble sentido cuando nulo.

Entonces se puede formar la siguiente consulta para analizar las calles unidireccionales para errores.

```
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.

Véase también

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

2.1.4 Consultas Personalizadas

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 Consejos de Rendimiento

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 *Soporte*) 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 *Soporte*) 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 *Datos Muestra* 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 Datos Muestra

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.

Crear tabla

```
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
);
```

Insertar los datos de la red

```
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.

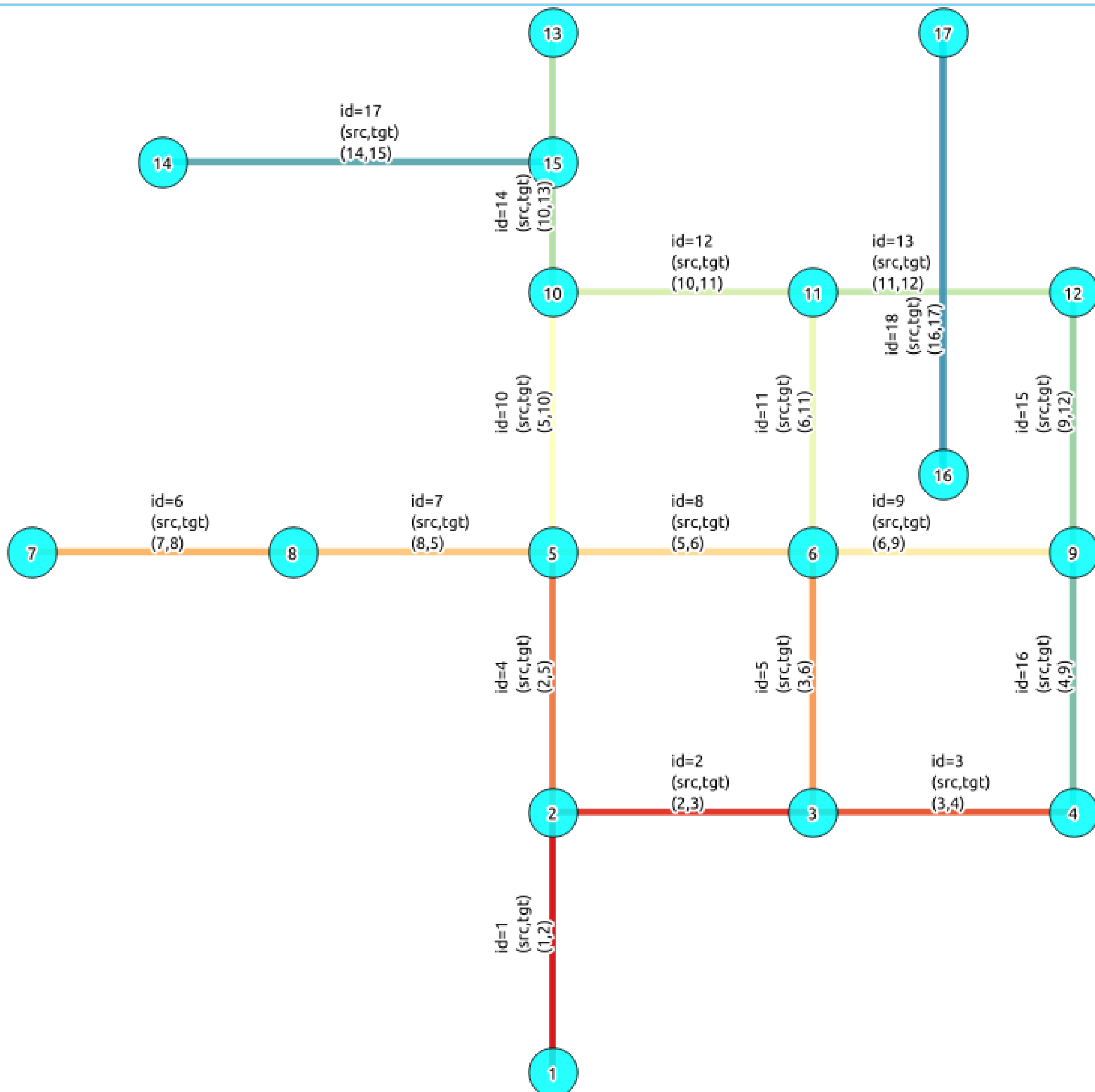
```
SELECT pgr_createTopology('edge_table',0.001);
```

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*



Para una introducción más completa de como construir una aplicación de ruteo, leer el Taller de pgRouting[3].

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

CAPÍTULO 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[]

Nombre

`pgr_costResult[]` — un conjunto de registros para describir un recorrido con el atributo de costo.

Descripción

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

seq identificador secuencial, indicando el orden del recorrido

id1 nombre genérico, a ser especificado por la función, típicamente el identificador del nodo

id2 nombre genérico, a ser especificado por la función, típicamente el identificador del borde

cost atributo de costo

3.1.2 pgr_costResult3[] - resultados múltiples de recorridos con costo

Nombre

`pgr_costResult3[]` — un conjunto de registros para describir varios resultados de recorridos con el atributo de costo.

Descripción

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

seq identificador secuencial, indicando el orden del recorrido

id1 nombre genérico, a ser especificado por la función, típicamente el identificador del recorrido

id2 nombre genérico, a ser especificado por la función, típicamente el identificador del nodo

id3 nombre genérico, a ser especificado por la función, típicamente el identificador del borde

cost atributo de costo

Historia

- Nuevo en la versión 2.0.0
- Sustituye a `path_result`

Véase también

- *Introduction*

3.1.3 pgr_geomResult[]

Nombre

`pgr_geomResult[]` — un conjunto de registros para describir un recorrido que incluye el atributo de la geometría.

Descripción

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

seq identificador secuencial, indicando el orden del recorrido

id1 nombre genérico, a ser especificado por la función

id2 nombre genérico, a ser especificado por la función

geom atributo de geometría

Historia

- Nuevo en la versión 2.0.0
- Sustituye a `geoms`

Véase también

- *Introduction*

CAPÍTULO 4

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

Nombre

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

Sinopsis

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')
```

Descripción

Parameters

La función de creación de topología requiere los siguientes parámetros:

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.

> **Advertencia:** 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.

Historia

- Renombrado en la versión 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.

Advertencia:
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');
```

Advertencia:
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:='tg
```

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

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:='mygeo
```

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:='mygeo
                           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 g
```

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

Ejemplos

```
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 *Datos Muestra* network.

Véase también

- *Topología para rutas* 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.

> **Advertencia:** 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.

> **Advertencia:**
> An error would occur when the arguments are not given in the appropriate order: In this example, the column source column `source` of the table `mytable` is passed to the function as the geometry column, and the geometry column `the_geom` is passed to the function as the source column.
> `SELECT pgr_createVerticesTable('edge_table','source','the_geom','target');`

When using the named notation

The order of the parameters do not matter:

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

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

Parameters defined with a default value can be 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');
```

> **Advertencia:**
> An error would occur when the arguments are not given in the appropriate order: In this example, the column `src` of the table `mytable` is passed to the function as the geometry column, and the geometry column `mygeom` is passed to the function as the source column.
> `SELECT pgr_createVerticesTable('mytable','src','mygeom','tgt');`

When using the named notation

The order of the parameters do not matter:

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

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

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

Selecting rows using rows_where parameter

Selecting rows based on the gid.

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

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

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

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

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

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

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

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

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 *Datos Muestra* network.

See Also

- *Topología para rutas* 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

Nombre

`pgr_anlizeGraph` — Analyzes the network topology.

Sinopsis

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')
```

Descripción

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.

Historia

- Nuevo en la versión 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.

Advertencia:
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');
```

Advertencia:
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='||qu
```

```
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='||quo
```

Ejemplos

```
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 *Datos Muestra* network.

Vease también

- *Topología para rutas* 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

Nombre

`pgr_analyzeOneway` — Analizarcalles unidireccionales e identifica segmentos invertidos

Sinopsis

Esta función analiza las calles unidireccionales en un gráfico e identifica cualquier segmentos invertido.

```
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);
```

Descripción

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

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

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

Prerequisites

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

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

Parameters

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

s_in_rules `text[]` reglas de **entrada** del nodo inicial

s_out_rules `text[]` reglas de **salida** del nodo inicial

t_in_rules `text[]` reglas de **entrada** del nodo final

t_out_rules `text[]` reglas de **salida** del nodo final

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`.

Nota: 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.

Historia

- Nuevo en la versión 2.0.0

Ejemplos

```
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)
```

La consulta usa la red de ejemplo *Datos Muestra*

Véase también

- *Topología para rutas* for an overview of a topology for routing algorithms.
- *Análisis de gráficas* 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

Nombre

`pgr_nodeNetwork` - Crea los nodos de una tabla de bordes de la red.

Author Nicolas Ribot

Copyright Nicolas Ribot, el código fuente está liberado bajo la licencia MIT-X.

Sinopsis

La función carga los bordes de una tabla que no tiene los nodos en las intersecciones y reescribe los bordes con los nodos en una nueva tabla.

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

Descripción

Un problema común asociado con la incorporación de datos SIG en pgRouting es el hecho de que los datos a menudo no están correctamente referenciados con nodos. Esto provoca topologías no válidas, que resultan en rutas que son incorrectas.

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

Historia

- Nuevo en la versión 2.0.0

Example

Let's create the topology for the data in *Datos Muestra*

```
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

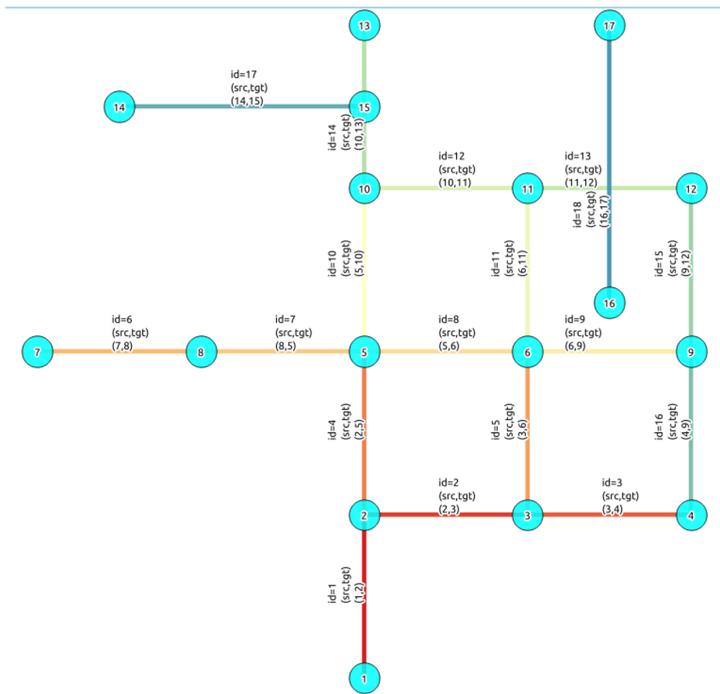
```
SELECT pgr_createTopology('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_createTopology('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait .....
NOTICE:  Creating Topology, Please wait...
NOTICE:  -------------> TOPOLOGY CREATED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

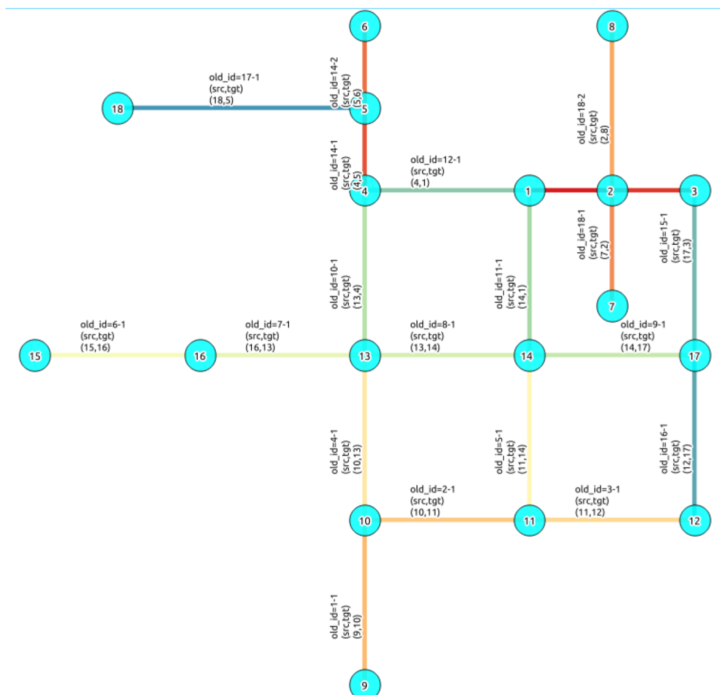
```
SELECT pgr_analyzegraph('edge_table_noded', 0.001);
NOTICE:  PROCESSING:
NOTICE:  pgr_analyzeGraph('edge_table_noded',0.001,'the_geom','id','source','target','true')
NOTICE:  Performing checks, pelase wait...
NOTICE:  Analyzing for dead ends. Please wait...
NOTICE:  Analyzing for gaps. Please wait...
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 = segment
                where edges.id in (select old_id from segmented where i>1) );
```

We recreate the topology:

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

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

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

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

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

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

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

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

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

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

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

```
NOTICE:                      Ring geometries: 0
pgr_createtopology
--------------------
OK
(1 row)
```

Véase también

Topología para rutas 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 - Ruta más corta de todos los pares, algoritmo de Johnson

Nombre

`pgr_apspJohnson` - Devuelve todos los costos para cada par de nodos en el gráfico.

Sinopsis

El algoritmo de Johnson, es una manera de encontrar los caminos más cortos entre todos los pares de vértices en una gráfica ponderada, dirigida y esparcida. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, cost) para cada par de nodos en el gráfico.

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

Descripción

sql una consulta SQL que debe porporcionar los bordes de la gráfica que se analizará:

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

source `int4` Identificador del vértice fuente

target `int4` Identificador del vértice objetivo

cost `float8` un valor positivo para el costo del recorrido del borde

Regresa un conjunto del tipo de datos *pgr_costResult[]*:

seq secuencia de registros

id1 Identificador del nodo de procedencia

id2 Identificador del nodo de llegada

cost costo para atravesar desde `id1` hasta `id2`

Historia

- Nuevo en la versión 2.0.0

Ejemplos

```
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 consulta usa la red de ejemplo *Datos Muestra*

Véase también

- *pgr_costResult[]*
- *pgr_apspWarshall - Camino más corto de todos los pares, Algoritmo de Floyd-Warshall*
- http://en.wikipedia.org/wiki/Johnson%27s_algorithm

4.2.2 pgr_apspWarshall - Camino más corto de todos los pares, Algoritmo de Floyd-Warshall

Nombre

`pgr_apspWarshall` - Devuelve todos los costos de cada par de nodos en la gráfica.

Sinopsis

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);
```

Descripción

sql una consulta SQL quedebe proporcionar los bordes de la gráfica que va a ser analizada:

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

id `int4` identificador del borde

source `int4` Identificador del vértice inicial de este borde

target `int4` Identificador del vértice final de este borde

cost `float8` un valor positivo para el costo de atravesar este borde

directed `true` Si la gráfica es direccionada

reverse_cost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utiliza para el calcular el costo de la travesía del borde en la dirección opuesta.

Devuelve un conjunto del tipo de datos *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del nodo de partida

id2 Identificador del nodo de llegada

cost costo para viajar desde el nodo `id1` hasta el nodo `id2`

Historia

- Nuevo en la versión 2.0.0

Ejemplos

```
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 consulta usa la red del ejemplo *Datos Muestra*

Véase también

- *pgr_costResult[]*
- *pgr_apspJohnson - Ruta más corta de todos los pares, algoritmo de Johnson*
- http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

4.2.3 pgr_astar - Camino más corto A*

Nombre

`pgr_astar` — Regresa el camino más corto usando el algoritmo A*.

Sinopsis

El algoritmo A* (pronunciado "A Star") se basa en el algoritmo de Dijkstra con una heurística que evalua sólo un subconjunto de la gráfica general, permiténdole resolver la mayoría de los problemas del camino más corto. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, cost) que conforman un camino.

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

Descripción

sql Consulta SQL, que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` identificador del borde

source `int4` Identificador del vértice de procedencia del borde

target `int4` Identificador del vértice de llegada del borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

x1 Coordenada `x` del punto inicial del borde

y1 Coordenada `y` del punto inicial del borde

x2 Coordenada `x` del punto final del borde

y2 Coordenada `y` del punto final del borde

reverse_cost (opcional) el costo para el recorrido inverso del borde. Esto sólo se utiliza cuando los parámetros `directed` y `has_rcost` son `True` (ver el comentario anterior sobre los costos negativos).

source `int4` identificador del punto de partida del recorrido

target `int4` identificador del punto de llegada del recorrido

directed `true` Si la gráfica es direccionada

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utilizan para el calcular el costo de la travesía del borde en la dirección opuesta.

Arroja un conjunto del tipo de datos *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del nodo visitado

id2 Identificador del borde usado (`-1` para el último)

cost Costo del recorrido desde el nodo `id1` usando el borde `id2` hasta su otro extremo

Historia

- Renombrado en la versión 2.0.0

Ejemplos

- Sin `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)
```

- Con `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)
```

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

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

4.2.4 pgr_bdAstar - Camino más corto bidireccional A*

Nombre

`pgr_bdAstar` - Regresa el camino más corto usando el algoritmo A* bidireccional.

Sinopsis

Este es un algoritmo de búsqueda bidireccional A*. Busca desde la fuente hasta el destino y al mismo tiempo desde el destino al origen y termina cuando estos las búsquedas se reúnen en el centro. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, costo) que conforman un camino.

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

Descripción

sql Consulta SQL que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` identificador del borde

source `int4` Identificador del vértice inicial del borde

target `int4` Identificador del vértice final del borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

x1 Coordenada `x` del punto del inicio del borde

y1 Coordenada y del punto del inicio del borde

x2 Coordenada x del punto del final del borde

y2 Coordenada y del punto del final del borde

reverse_cost (opcional) el costo para el recorrido inverso del borde. Se utiliza sólo cuando los parámetros `directed` y `has_rcost` son `True` (ver el comentario anterior acerca de los costos negativos).

source `int4` identificador del punto de partida

target `int4` Identificador del punto de llegada

directed `true` Si la gráfica es direccionada

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utilizan para el calcular el costo de la travesía del borde en la dirección opuesta.

Arroja un conjunto del tipo de datos *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del nodo visitado

id2 Identificador del borde (`-1` para el ultimo registro)

cost costo para atravesar desde el nodo `id1` usando el borde‘‘id2‘‘ hasta su otro extremo

> **Advertencia:** Usted debe reconectarse a la base de datos después de `CREATE EXTENSION pgRouting`. De lo contrario la función arrojará el error `Error computing path: std::bad_alloc`.

Historia

- Nuevo en la versión 2.0.0

Ejemplos

- Sin `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)
```

- Con `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)
```

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

- *pgr_costResult[]*
- *pgr_bdDijkstra - Camino más corto bidireccional de Dijkstra*
- http://en.wikipedia.org/wiki/Bidirectional_search
- http://en.wikipedia.org/wiki/A*_search_algorithm

4.2.5 pgr_bdDijkstra - Camino más corto bidireccional de Dijkstra

Nombre

`pgr_bdDijkstra` - Devuelve el recorrido más corto bidireccional usando el algoritmo de Dijkstra

Sinopsis

Este es un algoritmo de búsqueda bidireccional de Dijkstra. Realiza una búsqueda desde la fuente hacia el destino y, al mismo tiempo, desde el destino hacia el origen, terminando donde estas búsquedas se reúnen en el centro. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, cost) que conforman un camino.

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

Descripción

sql Consulta SQL, que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` identificador del borde

source `int4` Identificador del vértice inicial de este borde

target `int4` Identificador del vértice final de este borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

reverse_cost (opcional) El costo para el recorrido inverso del borde. Esto sólo se utiliza cuando los parámetros `directed` y `has_rcost` son `True` (ver el comentario anterior sobre los costos negativos).

source `int4` identificador del punto de partida

target `int4` Identificador del punto de llegada

directed `true` Si la gráfica es direccionada

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utiliza para el calcular el costo de la travesía del borde en la dirección opuesta.

Devuelve un conjunto del tipo de datos *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del nodo visitado

id2 identificador del borde (`-1` para el último)

cost costo del recorrido desde el nodo `id1` usando el borde `id2` hasta el otro extremo del borde

Historia

- Nuevo en la versión 2.0.0

Ejemplos

- Sin `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)
```

- Con `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)
```

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

- *pgr_costResult[]*
- *pgr_bdAstar - Camino más corto bidireccional A**
- http://en.wikipedia.org/wiki/Bidirectional_search
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

4.2.6 pgr_dijkstra - Camino más corto de Dijkstra

Nombre

`pgr_dijkstra` — Devuelve el Camino más corto usando el algoritmo de Dijkstra

Sinopsis

El algoritmo de Dijkstra, fue concebido por el científico computacional holandés, Edsger Dijkstra en 1956. Se trata de un algoritmo de búsqueda gráfica que resuelve el problema del camino más corto de una sola fuente con costos no negativos, generando un árbol de ruta más corta. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, cost) que conforman un recorrido.

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

Descripción

sql Consulta SQL, que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` identificador del borde

source `int4` Identificador del vértice inicial del borde

target `int4` Identificador del vértice final del borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

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` identificador del punto de partida

target `int4` Identificador del punto de llegada

directed `true` Si la gráfica es direccional

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utilizan para el calcular el costo de la travesía del borde en la dirección opuesta.

Devuelve un conjunto del tipo de datos *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del nodo visitado

id2 Identificador del borde (`-1` para el último borde)

cost costo para atravesar desde el nodo `id1` usando el borde `id2` hasta el otro extremo del borde

Historia

- Renombrado en la versión 2.0.0

Ejemplos

- Sin `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)
```

- Con `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)
```

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

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

4.2.7 pgr_kDijkstra - Camino más corto camino con múltiples destinos de Dijkstra

Nombre

- `pgr_kdijkstraCost` - Devuelve los costos de K caminos más cortos usando el algoritmo de Dijkstra.
- `pgr_kdijkstraPath` - Devuelve los K caminos más cortos usando el algoritmo de Dijkstra.

Sinopsis

Estas funciones permiten calcular las rutas a todos los destinos desde un nodo de partida único hasta múltiples nodos de destino. Devuelve un conjunto de *pgr_costResult* o de *pgr_costResult3*. `pgr_kdijkstraCost` devuelve un registro para cada nodo destino y el costo total de la ruta hasta ese nodo. `pgr_kdijkstraPath` devuelve un registro por cada borde en la ruta desde la fuente hasta el destino junto con el costo para atravesar ese borde.

```
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);
```

Descripción

sql Consulta SQL que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` Identificador del borde

source `int4` Identificador del vértice inicial del borde

target `int4` Identificador del vértice final del borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

reverse_cost (opcional) El costo para el recorrido inverso del borde. Esto sólo se utiliza cuando los parámetros `directed` y `has_rcost` son `True` (ver el comentario anterior sobre los costos negativos).

source `int4` Identificador del punto de partida

targets `int4[]` Una matriz de identificadores de los puntos de llegada

directed `true` Si la gráfica es direccionada

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utilizan para el calcular el costo de la travesía del borde en la dirección opuesta.

`pgr_kdijkstraCost` devuelve el conjunto de *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del vértice de partida (esto siempre será punto de partida de la fuente en la consulta).

id2 Identificador del vértice de llegada

cost Costo para recorrer el camino desde `id1` hasta `id2`. Costo será -1.0 si no hay camino al identificador de vértice de destino.

`pgr_kdijkstraPath` devuelve el conjunto de *pgr_costResult3[] - resultados múltiples de recorridos con costo*:

seq Secuencia de registros

id1 Identificador del destino de la ruta (identifica el camino hacia el destino).

id2 path edge source node id

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

cost Costo para atravesar este borde o -1.0 si no hay ningún camino a este objetivo

Historia

- Nuevo en la versión 2.0.0

Ejemplos

- Devolviendo un resultado de costos `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)
```

- Devolviendo resultado de un trayecto `path`

```
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)
```

No hay ninguna garantía de que el resultado anterior esté ordenado en la dirección del flujo de la ruta, es decir, puede estar invertido. Usted necesitará comprobar si `st_startPoint()` de la ruta es la misma que la ubicación del nodo de inicio y si no, entonces utilizar `st_reverse()` para invertir la dirección de la ruta. Este comportamiento es de funciones pertenecientes a PostGIS `st_linemerge()` y `st_union()` y no pertenecen a pgRouting.

Véase también

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

4.2.8 pgr_ksp - K caminos más cortos

Nombre

`pgr_ksp` — Devuelve K caminos más cortos.

Sinopsis

El algoritmo del camino más corto K, está basado en el algoritmo de Yen. "K" es el número de caminos más cortos deseados. Regresa un conjunto de registros *pgr_costResult3* (seq, id1, id2, id3, cost) que conforman K caminos.

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

Descripción

sql Consulta SQL, que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` identificador del borde

source `int4` Identificador del vértice inicial de este borde

target `int4` Identificador del vértice final del borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

reverse_cost (opcional) el costo para el recorrido inverso del borde. Se utiliza sólo cuando el parámetro `has_rcost` es `True` (ver el comentario anterior acerca de los costos negativos).

source `int4` Identificador del punto de partida

target `int4` Identificador del punto de llegada

paths `int4` Cantidad de rutas alternativas

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utilizan para el calcular el costo de la travesía del borde en la dirección opuesta.

Arroja un conjunto del tipo de datos *pgr_costResult[]*:

seq sequence for ording the results

id1 Identificador de la ruta

id2 Identificador del nodo visitado

id3 Identificador del borde (`0` para el ultimo registro)

cost costo para atravesar desde el nodo `id2` usando el borde `id3` hasta su otro extremo

El código base de KSP fue adquirido de http://code.google.com/p/k-shortest-paths/source.

Historia

- Nuevo en la versión 2.0.0

Ejemplos

- Sin `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)
```

- Con `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)
```

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

- *pgr_costResult3[] - resultados múltiples de recorridos con costo*
- http://en.wikipedia.org/wiki/K_shortest_path_routing

4.2.9 pgr_tsp - Vendedor Viajante

Nombre

- `pgr_tsp` - Devuelve la mejor ruta desde un nodo inicial vía un lista de nodos.
- `pgr_tsp` - Devuelve el mejor orden de la ruta cuando se le introduce una matriz de distancias
- `pgr_makeDistanceMatrix` - Devuelve una matriz de distancias Euclidiana desde los puntos que le provee la consulta sql.

Sinopsis

En el problema del vendedor viajante (TSP) o problema del vendedor se hace la siguiente pregunta: dada una lista de las ciudades y las distancias entre cada par de ciudades, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y vuelve a la ciudad de origen. Este algoritmo hace simulaciones para devolver una solución aproximada de alta calidad. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, cost) que conforman un camino.

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

Devuelve un conjunto de (seq integer, integer id1, id2 integer, cost float8) que representa el mejor orden para visitar los nodos en la matriz. `id1` es el índice en la matriz de distancia. `id2` es el identificador del punto de la consulta sql.

Si no se suministra el identificador de un punto de llegada `end_id` o si es -1 o si es el mismo que al identificador del punto de partida `start_id`, TSP supone que se regresa al punto de partida. Si se suministra un punto de llegada `end_id` entonces se supone que la ruta inicia y termina en los identificadores designados.

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

Descripción

Con distancias Euclidianas

El evaluador TSP se basa en los puntos ordenados utilizando la línea recta (euclidiana) de distancias entre nodos. La aplicación utiliza un algoritmo de aproximación que es muy rápido. No es una solución exacta, pero se garantiza que una solución se devuelve después de cierta cantidad de iteraciones.

sql Consulta SQL que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` Identificador del vértice

x `float8` coordenada x

y `float8` coordenada y

start_id `int4` Identificador del punto de partida

end_id `int4` identificador del punto de llegada, esto es *opcional*, si se incluye la ruta será optimizada desde la partida hasta la llegada, de lo contrario se supone que la partida y la llegada son el mismo punto.

La función devuelve el conjunto de *pgr_costResult[]*:

seq Secuencia de registros

id1 índice interno de la matriz de distancias

id2 `int4` identificador del nodo

cost costo para atravesar desde el nodo actual hasta el siguiente nodo.

Crear una matriz de distancias

Para usuarios necesitan una matriz de distancias tenemos una función simple que toma consultas SQL en `sql` como se describe anteriormente y devuelve un registro con `dmatrix` e `ids`.

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

La función devuelve un registro de `dmatrix`, `` ids``:

dmatrix `float8[][]` una matriz de distancia euclidiana simétrica basado en la consulta `sql`.

ids `integer[]` una matriz de identificadores basados en el orden de la matriz de distancias.

Con la matriz de distancia

Para los usuarios, que no requieren utilizar distancias euclidianas, tenemos también la posibilidad de pasar una matriz de distancia que resuelve y devuelve una lista ordenada de nodos con el mejor orden visita de cada nodo. Es el usuario debe llenar completamente la matriz de distancia.

matrix `float[][]` matriz de distancia de puntos

start `int4` índice del punto de inicio

end `int4` (opcional) índice del nodo final

El nodo final `end` es un parámetro opcional, se puede omitir cuando se requiere un bucle donde el principio `start` es un depósito y la ruta debe regresar de vuelta al depósito. Al incluir el parámetro final `end`, se optimiza el camino de principio `start` al final `end` y se reduce al mínimo la distancia de la ruta al ir incluyendo los puntos restantes.

La matriz de distancias es un multidimensional PostgreSQL array type[1] que debe ser de tamaño `N x N`.

El resultado será de N registros de `[ seq, id ]`:

seq Secuencia de registros

id índice dentro de la matriz

Notas al pie

Historia

- Renombrado en la versión 2.0.0
- Dependencia de la GAUL eliminada en la versión 2.0.0

Ejemplos

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

```
    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)
```

- Utilizando la matriz de distancia (un circuito a partir 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)
```

- Utilizando la matriz de distancia (a partir de 1, terminando en 2)

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

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

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

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

  seq | id1 | id2 | cost
 -----+-----+-----+------
    0 |   5 |   6 | 0.00
    1 |  10 |  11 | 0.00
    2 |   2 |   3 | 1.41
    3 |   3 |   4 | 0.00
```

```
   4 |   11 |   12 | 0.00
   5 |    8 |    9 | 0.71
   6 |   15 |   16 | 0.00
   7 |   16 |   17 | 2.12
   8 |    1 |    2 | 0.00
   9 |   14 |   15 | 1.41
  10 |    7 |    8 | 1.41
  11 |    6 |    7 | 0.71
  12 |   13 |   14 | 2.12
  13 |    0 |    1 | 0.00
  14 |    9 |   10 | 0.00
  15 |   12 |   13 | 0.00
  16 |    4 |    5 | 1.41
(17 rows)

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

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

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

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

4.2.10 pgr_trsp - Camino más corto con giros restringidos (TRSP)

Nombre

`pgr_trsp` — Devuelve el camino más corto con soporte para restricciones de giros

Sinopsis

El Camino más corto con giros restringido (TRSP), es un algoritmo de camino más corto que puede tomar en cuenta complicadas restricciones de giro, como las encontrados en las redes de carreteras navegables reales. El rendimiento es casi tan rápido como la búsqueda A*, pero tiene muchas características adicionales como funcionalidad en base a los bordes en vez de basarse en los nodos de la red. Devuelve un conjunto de registros *pgr_costResult* (seq, id1, id2, costo) que conforman un camino.

```
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]);
```

Descripción

El algoritmo del Camino más corto con giros restringidos (TRSP) es similar al de *Algoritmo Shooting Star* en cuanto a que puede uno especificar restricciones de giros.

La configuración del TRSP es parecido al del *camino más corto de Dijkstra* con el añadido de una tabla de restricciones de giros que es opcional. Esto hace que añadir restricciones de giro a una red de carreteras sea más fácil en comparación con del algoritmo de estrella fugaz en la que había que agregar los bordes varias veces cuando estaba involucrado en una restricción.

sql Consulta SQL que debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` Identificador del borde

source `int4` Identificador del vértice inicial del borde

target `int4` Identificador del vértice final del borde

cost `float8` valor del costo del recorrido sobre el borde. Un costo negativo evitará que el borde sea insertado en el gráfico.

reverse_cost (opcional) El costo para el recorrido inverso del borde. Esto sólo se utiliza cuando los parámetros `directed` y `has_rcost` son `True` (ver el comentario anterior sobre los costos negativos).

source `int4` **identificador** del nodo de partida

target `int4` **identificador** del nodo de llegada

directed `true` Si la gráfica es direccionada

has_rcost Si es `True`, el campo `reverse_cost` del conjunto de registros generados se utilizan para el calcular el costo de la travesía del borde en la dirección opuesta.

restrict_sql (opcional) una consulta SQL, que debe proporcionar un conjunto de registros con los siguientes campos:

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

to_cost `float8` restricción del costo de giro

target_id `int4` identificador del borde donde se aplica la restricción

via_path *text*‘ lista de bordes separados por comas que llegan al borde `target_id` que conforman esta restricción

Otra variante de TRSP que permite especificar el **Identificador del borde** de partida y de llegada junto con una fracción para interpolar la posición:

source_edge `int4` **identificador del borde** de partida

source_pos `float8` fracción de 1 que define la posición del sobre el borde de partida.

target_edge `int4` **Identificador del borde** de llegada

target_pos `float8` fracción de 1 que define la posición del sobre el borde de llegada.

Devuelve un conjunto del tipo de datos *pgr_costResult[]*:

seq Secuencia de registros

id1 Identificador del nodo visitado

id2 identificador del borde (`-1` para el último borde)

cost Costo para el recorrido desde el nodo `id1` usando el borde `id2` hasta su otro extremo

Historia

- Nuevo en la versión 2.0.0

Ejemplos

- Sin restricción de giros

```
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)
```

- Con restricción de giros

Las restricciones de giro requieren de información adicional, que puede ser almacenado en una tabla por separado:

```
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);
```

Entonces una consulta con restricciones de giro es creada de la siguiente forma:

```
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)
```

Las consultas usan la red de ejemplo *Datos Muestra*

Véase también

- *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.

Nota: 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

Advertencia: 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 *Datos Muestra* network.

See Also

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

4.3.2 pgr_alphaShape

Nombre

`pgr_alphashape` — Función central para el cálculo de la forma alfa.

Nota: Esta función no debe utilizarse directamente. Usar *pgr_drivingDistance* en su lugar.

Sinopsis

Devuelve una tabla con registros (x, y) que describen los vértices de una figura de alfa.

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

Descripción

sql `text` una consulta SQL la cual debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` Identificador del vértice

x `float8` Coordenada x

y `float8` Coordenada y

Devuelve un registro de vértices para cada fila:

x Coordenada x

y Coordenada y

Historia

- Renombrado en la versión 2.0.0

Ejemplos

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))))::num
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  F
) as a) as foo;

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

Las consulta usa la red de ejemplo *Datos Muestra*

Véase también

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

4.3.3 pgr_pointsAsPolygon

Nombre

`pgr_pointsAsPolygon` — Dibuja un figura Alfa alrededor de un conjunto de puntos dado.

Sinopsis

Devuelve la forma alfa con geometría de polígono.

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

Descripción

sql `text` una consulta SQL la cual debe proporcionar un conjunto de registros con los siguientes campos:

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

id `int4` Identificador del vértice

x `float8` coordenada x

y float8 coordenada y

Devuelve una geometría polígonal.

Historia

- Renombrado en la versión 2.0.0

Ejemplos

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_ge
                                      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(th
     FROM edge_table_vertices_pgr'))::numeric, 2) as st_area;

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

Las consulta usa la red de ejemplo *Datos Muestra*

Véase también

- *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.

Nota: 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 *Datos Muestra* network.

See Also

- *Guía del desarrollador* 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.

Nota: 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 *Datos Muestra* network.

See Also

- *Guía del desarrollador* 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.

Nota: 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 *Datos Muestra* network.

See Also

- *Guía del desarrollador* 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.

Nota: 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 *Datos Muestra* network.

See Also

- *Guía del desarrollador* 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.

Nota: 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.

Advertencia: 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

- *Guía del desarrollador* 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.

Nota: 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

- *Guía del desarrollador* 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.

Nota: 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

- *Guía del desarrollador* 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.

Nota: 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

- *Guía del desarrollador* 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.

Nota: 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

- *Guía del desarrollador* for the tree layout of the project.
- *pgr_startPoint* to get the start point of a (multi)linestring.

4.5 Funciones heredadas

En la versión 2.0 de pgRouting se hizo una reestructuración total en la nomenclatura de las funciones y se han sustituido muchas funciones que estaban disponibles en las versiones 1.x. Mientras esto puede ser un inconveniente para nuestros usuarios existentes, pensamos que esto era necesario para la viabilidad a largo plazo del proyecto, mejorar la respuesta a más de nuestros usuarios, poder añadir nuevas funcionalidades y probar la funcionalidad existente.

Se hizo mínimo esfuerzo para salvar a la mayoría de estas funciones y distribuir con la versión un archivo `pgrouting_legacy.sql` que no es parte de la extensión pgRouting y no están siendo apoyadas. Si puede utilizar estas funciones, que bueno. No hemos probado ninguna de estas funciones, entonces si encuentra problemas y publicar un parche para ayudar a otros usuarios, es correcto pero lo más probable es que ese archivo sea eliminado en versiones futuras, por lo que le recomendamos encarecidamente que convierta su código existente para utilizar las nuevas funciones documentadas y apoyadas por el equipo de soporte.

La siguiente es una lista de: tipos, modelos y funciones incluidas en el archivo `pgrouting_legacy.sql`. La lista es para proporcionarle comodidad, pero estas funciones son obsoletas, no apoyadas y probablemente tendrá que hacer cambios para hacerlas trabajar.

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 Funciones Descontinuadas

La nueva funcionalidad de las versiones principales pueden ser cambiadas y, por ende, las funciones pueden ser descontinuadas por varias razones. La funcionalidad que ha sido descontinuada aparecerá aquí.

4.6.1 Algoritmo Shooting Star

Version Descontinuada en 2.0.0

Reasons Errores sin resolver, sin mantenimiento, sustituido con *pgr_trsp - Camino más corto con giros restringidos (TRSP)*

Comment Por favor *contáctenos* si usted está interesado en patrocinar o mantener este algoritmo. La firma de la función sigue disponible en *Funciones heredadas* pero es simplemente un contenedor que genera un error. No hemos incluido ningún código anterior en esta versión.

CAPÍTULO 5

Desarrolladores

5.1 Guía del desarrollador

Nota: Toda la documentación debe estar en formato reStructuredText. Ver http://docutils.sf.net/rst.html para documentaciones introductorias.

5.1.1 Diseño del árbol de la fuente

cmake/ CMake scripts usados como parte de nuestro sistema de construcción.

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/ Esto es una implementación de la búsqueda A* basada en el uso de las bibliotecas de gráficas Boost para su implementación. Esto es una implementación de ruta más corta de Dijkstra con una heurística euclidiana.

core/common/ Por el momento esto no tiene un núcleo en "src", pero tiene varios códigos de contenedores de SQL y código de topología en el directorio "sql". *Algoritmos de Contenedores específicos serán movidos al árbol de algoritmos y se deben conseguir las pruebas adecuadas para validar los Contenedores.*

core/dijkstra/ Esto es una implementación de la solución de ruta más corta de Dikjstra usando las bibliotecas de gráficas Boost para la aplicación.

core/driving_distance/ Este paquete opcional crea polígonos de distancia de manejo basados en la solución de la ruta más corta de Dijkstra, luego crea polígonos basados en las distancias de igual costo desde el punto de inicial. Este paquete opcional requiere que las bibliotecas CGAL estén instaladas.

core/shooting_star/ *OBSOLETO y NO FUNCIONA y está SIENDO ELIMINADO* este es un algoritmo basado algoritmo del camino más corto que soporta restricciones de giro. Se basa en las bibliotecas de gráficas Boost. *NO* usar este algoritmo ya que está roto, en su lugar usar *trsp* que tiene la misma funcionalidad, es más rápido y da resultados correctos.

core/trsp/ Este es un algoritmo de ruta más corto giro restringido. Tiene algunas características como: poder especificar los puntos iniciales y finales en forma de porcentaje a lo largo de un borde. Las restricciones se almacenan en una tabla separada de los bordes del gráfico, haciendo mas sencilla la administración de los datos.

core/tsp/ Este paquete opcional proporciona la capacidad de calcular soluciones de problema del vendedor viajante y calcular la ruta resultante. Este paquete opcional requiere las que las bibliotecas GAUL se encuentren instaladas.

tools/ Varios scripts y herramientas.

lib/ Este es el directorio de salida donde se almacenan las bibliotecas compiladas y objetivos de la instalación antes de la instalación.

5.1.2 Diseño de documentación

Como se señaló anteriormente, la documentación debe hacerse usando archivos con formato reStructuredText.

La documentación se distribuye en el árbol de las fuentes. Este directorio "doc" de nivel superior está destinado a las portada de los temas de documentación de nivel alto como:

- Compilación y pruebas
- Instalación
- Tutoriales
- Materiales para guía de los usuarios
- Materiales de manuales de referencia
- etc.

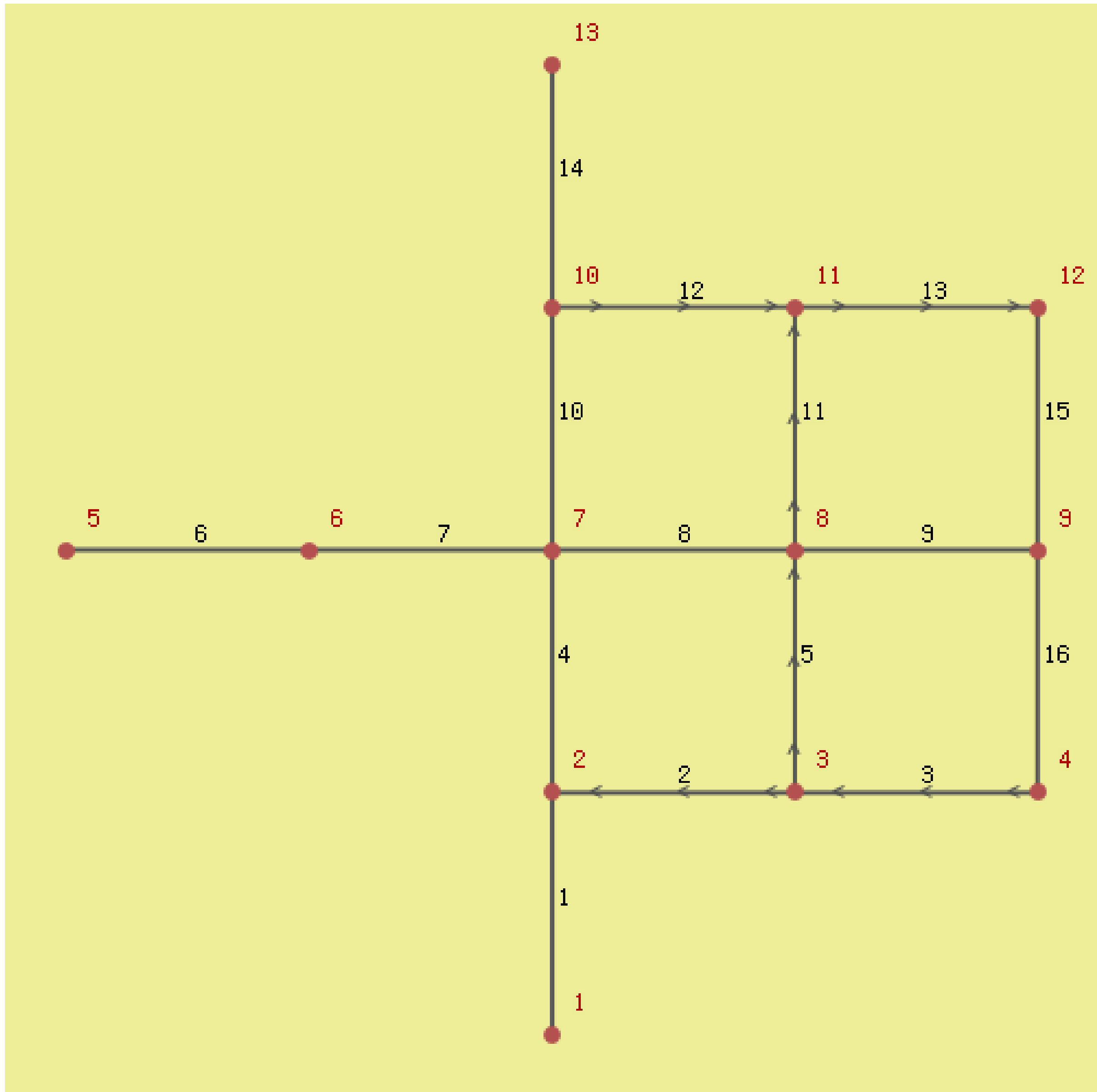
Puesto que la documentación específica del algoritmo se contiene en el árbol de código fuente con los archivos específicos de algoritmo, el proceso de construcción de la documentación y publicación tendrá que ensamblar los detalles con el material según se necesite.

También, para evitar que el directorio de "doc" se desordene, cada libro mayor, como los mencionados anteriormente, deberán estar incluidos en un directorio separado bajo el directorio "doc". Cualquier imágen u otros materiales relacionados con el libro, deberán mantenerse en ese directorio.

5.1.3 Pruebas de infraestructura

Se proporciona una infraestructura de prueba muy básica. Aquí están los fundamentos del cómo funciona. Necesitamos más casos de prueba. A más largo plazo que probablemente se tendrá un encargado de configurar marcos de pruebas travis-ci o jenkins .

Aquí está el gráfico para las pruebas TRSP.



Las pruebas se ejecutan mediante el script en el nivel superiortools/test-runner.pl y ejecuta todas la pruebas configuradas arrojando la estructura de resultados de la prueba. Esto puede ser mejorado en un futuro.

> También supone que se tienen instaladas las bibliotecas ya que hace uso del postgresql instalado. Probablemente esto necesita mejorarse para poner probar el árbol de la construcción. Pensar en eso.

Básicamente, cada directorio .../test/ debe incluir un *test.conf* Archivo que es un fragmento de script de perl que define que archivos de datos cargar y las pruebas a ejecutar. Se ha desarrollado mecanismos para permitir que la prueba y los datos sean específico para las versiones pg y PostGIS, pero no está habilitado todavía. Así que por ejemplo, *core/trsp/test/test-any-00.data* es un volcado (dump) de texto sin formato de sql que carga los datos necesarios para una serie de pruebas. Este es también el gráfo en la imagen de arriba. Usted puede especificar varios archivos para cargar, pero como grupo hay la necesidad de tener nombres únicos.

core/TRSP/test/test-any-00.test es un comando sql a ser ejecutado. Se ejecutará como:

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

Entonces si hay una diferencia se reporta el fracaso de la prueba.

5.2 Release Notes

5.2.1 Notas de la versión 2.0 de pgRouting

Con el lanzamiento de la versión 2.0 de pgRouting, las bibliotecas dejan de tener compatibilidad con las versiones *pgRouting 1.x* . Se hizo esto para poder reestructurar pgRouting, estandarizando los nombres de las funciones y preparando el proyecto para el futuros desarrollos. Como resultado de este esfuerzo, hemos sido capaces de simplificar pgRouting, añadiendo funcionalidades importantes, integrando la documentación y probando el árbol del código fuente, haciéndolo más sencillo para que varios desarrolladores puedan contribuir.

Para los cambios importantes, consulte las siguientes notas de la versión. Para ver la lista completa de cambios revise la lista de Git commits[1] en Github.

Cambios en la versioón 2.0.0

- Análisis gráfico - herramientas para detectar y arreglar algunos problemas de conexión en un gráfico de red
- Una colección de funciones útiles
- Dos nuevos algoritmos para el camino más corto de todos pares (pgr_apspJohnson, pgr_apspWarshall)
- Algoritmos para el Dijkstra bidireccional y la búsqueda A*(pgr_bdAstar, pgr_bdDijkstra)
- Búsqueda de uno a varios nodos (pgr_kDijkstra)
- K alternate paths shortest path (pgr_ksp)
- Nuevo solucionador TSP que simplifica el código donde el proceso de compilación (pgr_tsp), ya no depen-dene de la biblioteca "Gaul Library"
- Ruta más corta con giros restringidos (pgr_trsp) que reemplaza a la estrella fugaz "Shooting Star"
- Se deja de soportar a la Estrella fugaz "Shooting Star"
- Se construye una infraestructura de pruebas que se ejecuta antes de que se incorporen grandes cambios al código principal.
- Se probó y se arreglaron todos los errores excepcionales registrados en la versión 1.x existente en la base del código 2.0-dev.
- Proceso de compilación mejorados para Windows
- Automated testing on Linux and Windows platforms trigger by every commit
- Diseño modular de bibliotecas
- Compatibilidad con PostgreSQL 9.1 o posterior
- Compatibilidad con PostGIS 2.0 o posterior
- Se instala como EXTENSION de PostgreSQL
- Los Tipos de datos son unificados
- Soporte para esquema en los parámetros de las funciones
- Soporte para el prefijo `st_` de las funciones de PostGIS
- Prefijo `pgr_` agregado a las funciones y tipos
- Mejor documentación: http://docs.pgrouting.org

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

5.2.2 Notas de versión de pgRouting 1.x

Las siguientes notas han sido copiadas desde el archivo anterior de `RELEASE_NOTES` y se mantienen como referencia. A partir de la versión *version 2.0.0* , las notas seguirán un esquema diferente.

Cambios para la versión 1.05

- Corrección de errores

Cambios para la versión 1.03

- Creación de topología mucho más rápida
- Corrección de errores

Cambios para la versión 1.02

- Corrección de errores de Shooting*
- Se resolvieron problemas de compilación

Cambios para la versión 1.01

- Corrección de errores de Shooting*

Cambios para la versión 1.0

- Núcleo y funciones adicionales están separadas
- Proceso de compilación con CMake
- Corrección de errores

Cambios para la versión 1.0.0b

- Archivos SQL adicionales con nombres más simples para las funciones de la envolturas
- Corrección de errores

Cambios para la versión 1.0.0a

- Algoritmo de ruta más corta Shooting* para redes de camino reales
- Se arreglaron varios errores de SQL

Cambios para la versión 0.9.9

- Soporte PostgreSQL 8.2
- Funciones de ruta más cortos regresan nulo si no pudieron encontrar ningún camino

Cambios para versión 0.9.8

- Esquema de numeración ha sido añadido a las funciones de ruta más cortos
- Se agregaron funciones de ruta más cortos dirigidas
- routing_postgis.SQL fue modificado para utilizar dijkstra en la búsqueda TSP

Índices y tablas

- *genindex*
- *search*