Workshop - FOSS4G routing with pgRouting tools, OpenStreetMap road data and GeoExt Manual

Release 1

Daniel Kastl, Frédéric Junod

July 05, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

Abstract

pgRouting adds routing functionality to PostGIS. This introductory workshop will show you how. It gives a practical example of how to use pgRouting with OpenStreetMap road network data. It explains the steps to prepare the data, make routing queries, assign costs and use GeoExt to show your route in a web-mapping application.

Navigation for road networks requires complex routing algorithms that support turn restrictions and even time-dependent attributes. pgRouting is an extendible open-source library that provides a variety of tools for shortest path search as extension of PostgreSQL and PostGIS. The workshop will explain about shortest path search with pgRouting in real road networks and how the data structure is important to get faster results. Also you will learn about difficulties and limitations of pgRouting in GIS applications.

To give a practical example the workshop makes use of OpenStreetMap data of Barcelona. You will learn how to convert the data into the required format and how to calibrate the data with "cost" attributes. Furthermore we will explain the difference of the main routing algorithms "Dijkstra", "A-Star" and "Shooting-Star". By the end of the workshop you will have a good understanding of how to use pgRouting and how to get your network data prepared.

To learn how to get the output from rows and columns to be drawn on a map, we will build a basic map GUI with GeoExt. We listened to the students feedback of the last year's and want to guide you through the basic steps to build a simple browser application. Our goal is to make this as easy as possible, and to show that it's not difficult to integrate with other FOSS4G tools. For that reason we selected GeoExt, which is a JavaScript library providing the groundwork for creating web-mapping applications based on OpenLayers and Ext.

Note:

- Workshop level: intermediate
- Attendee's previous knowledge: SQL (PostgreSQL, PostGIS), Javascript, HTML
- Equipments: This workshops will make use of the GIS LiveDVD if possible. Otherwise it will require VirtualBox installed to load a virtual machine image.

Presenter

- *Daniel Kastl* is founder and CEO of Georepublic UG and works in Germany and Japan. He is moderating and promoting the pgRouting community and development since 4 years, and he's an active OSM contributor in Japan.
- *Frédéric Junod* works at the Swiss office of Camptocamp for about five years. He's an active developer of many open source GIS projects from the browser (GeoExt, OpenLayers) to the server world (MapFish, Shapely, TileCache) and he is member of the pgRouting PSC.

Daniel and Frédéric are the auhtors of the previous pgRouting workshops, that have been held at FOSS4G events in Canada and South Africa and at local conferences in Japan.

License

CHAPTER

TWO

ABOUT

This workshop makes use of several FOSS4G tools, a lot more than the workshop title mentions. Also a lot of FOSS4G software is related to other open source projects and it would go too far to list them all. These are the four FOSS4G projects this workshop will focus on:

2.1 pgRouting

pgRouting is an extension of PostGIS and adds routing functionality to PostGIS/PostgreSQL. pgRouting is a further development of pgDijkstra (by Camptocamp SA). It was extended by Orkney Inc., and is currently developed and maintained by Georepublic.

pgRouting provides functions for:

- Shortest Path Dikstra: routing algorithm without heuristics
- Shortest Path A-Star: routing for large datasets (with heuristics)
- Shortest Path Shooting-Star: routing with turn restrictions (with heuristics)
- Traveling Salesperson Problem (TSP)
- Driving Distance calculation (Isolines)

Advantages of the database routing approach are:

- Accessible by multiple clients through JDBC, ODBC, or directly using Pl/pgSQL. The clients can either be PCs or mobile devices.
- Uses PostGIS for its geographic data format, which in turn uses OGC's data format Well Konwn Text (WKT) and Well Known Binary (WKB). This allows usage of existing open * data converters.
- Open Source software like qGIS and uDig can modify the data/attributes,
- Data changes can be reflected instantaneously through the routing engine. There is no need for precalculation.
- The "cost" parameter can be dynamically calculated through SQL and its value can come from multiple fields or tables.

pgRouting is available under the GPLv2 license.

pgRouting website: http://www.pgrouting.org

2.2 OpenStreetMap

"OpenStreetMap is a project aimed squarely at creating and providing free geographic data such as street maps to anyone who wants them. The project was started because most maps you think of as free actually have legal or technical restrictions on their use, holding back people from using them in creative, productive or unexpected ways." (Source: http://wiki.openstreetmap.org/index.php/Press)

OpenStreetMap is a perfect data source to use for pgRouting, because it's freely available and has no technical restrictions in terms of processing the data. Data availability still varies from country to country, but the worldwide coverage is improving day by day.

OpenStreetMap uses a topological data structure:

- Nodes are points with a geographic position.
- Ways are lists of nodes, representing a polyline or polygon.
- Relations are groups of nodes, ways and other relations which can be assigned certain properties.
- Tags can be applied to nodes, ways or relations and consist of name=value pairs.

OpenStreetMap website: http://www.openstreetmap.org

2.3 osm2pgrouting

osm2pgrouting is a command line tool that makes it easy to import OpenStreetMap data into a pgRouting database. It builds the routing network topology automatically and creates tables for feature types and road classes. osm2pgrouting was primarily written by Daniel Wendt and is now hosted on the pgRouting project site.

osm2pgrouting is available under the GPLv2 license.

Project website: http://pgrouting.postlbs.org/wiki/tools/osm2pgrouting

2.4 GeoExt

GeoExt is a "JavaScript Toolkit for Rich Web Mapping Applications". GeoExt brings together the geospatial know how of OpenLayers with the user interface savvy of Ext JS to help you build powerful desktop style GIS apps on the web with JavaScript.

GeoExt is available under the BSD license and is supported by a growing community of individuals, businesses and organizations.

GeoExt website: http://www.geoext.org

CHAPTER

THREE

INSTALLATION AND REQUIREMENTS

```
<?xml version=’1.0’ encoding=’UTF-8’?>
<osm version=’0.6’ generator=’xapi: OSM Extended API 2.0’ ... >
  ...
  <node id=’255405560’ lat=’41.4917468’ lon=’2.0257695’ version=’1’
                changeset=’19117’ user=’efrainlarrea’ uid=’32823’ visible=’true’
                timestamp=’2008-04-02T17:40:07Z’>
  </node>
  <node id=’255405551’ lat=’41.4866740’ lon=’2.0302842’ version=’3’
                changeset=’248452’ user=’efrainlarrea’ uid=’32823’ visible=’true’
                timestamp=’2008-04-24T15:56:08Z’>
  </node>
  <node id=’255405552’ lat=’41.4868540’ lon=’2.0297863’ version=’1’
                changeset=’19117’ user=’efrainlarrea’ uid=’32823’ visible=’true’
                timestamp=’2008-04-02T17:40:07Z’>
  </node>
  ...
  <way id=’35419222’ visible=’true’ timestamp=’2009-06-03T21:49:11Z’
                version=’1’ changeset=’1416898’ user=’Yodeima’ uid=’115931’>
    <nd ref=’415466914’/>
    <nd ref=’415466915’/>
    <tag k=’highway’ v=’unclassified’/>
    <tag k=’lanes’ v=’1’/>
    <tag k=’name’ v=’Carrer del Progrés’/>
    <tag k=’oneway’ v=’no’/>
  </way>
  <way id=’35419227’ visible=’true’ timestamp=’2009-06-14T20:37:55Z’
                version=’2’ changeset=’1518775’ user=’Yodeima’ uid=’115931’>
    <nd ref=’415472085’/>
    <nd ref=’415472086’/>
    <nd ref=’415472087’/>
    <tag k=’highway’ v=’unclassified’/>
    <tag k=’lanes’ v=’1’/>
    <tag k=’name’ v=’carrer de la mecanica’/>
    <tag k=’oneway’ v=’no’/>
  </way>
  ...
  <relation id=’903432’ visible=’true’ timestamp=’2010-05-06T08:36:54Z’
                version=’1’ changeset=’4619553’ user=’ivansanchez’ uid=’5265’>
    <member type=’way’ ref=’56426179’ role=’outer’/>
    <member type=’way’ ref=’56426173’ role=’inner’/>
    <tag k=’layer’ v=’0’/>
    <tag k=’leisure’ v=’common’/>
    <tag k=’name’ v=’Plaça Can Suris’/>
    <tag k=’source’ v=’WMS shagrat.icc.cat’/>
    <tag k=’type’ v=’multipolygon’/>
  </relation>
  ...
</osm>
```

The OSM data can be downloaded from OpenStreetMap website using an API (currently v0.6, see http://wiki.openstreetmap.org/index.php/OSM_Protocol_Version_0.6), or with some OSM tools like JOSM editor for example.

Also CloudMade offers extracts of maps from different places around the world. For data for Barcelona for example go to http://download.cloudmade.com/europe/spain and download:

```
wget http://download.cloudmade.com/europe/spain/spain.osm.bz2
```

CHAPTER

FOUR

OSM2PGROUTING IMPORT TOOL

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <type name="highway" id="1">
    <class name="motorway" id="101" />
    <class name="motorway_link" id="102" />
    <class name="motorway_junction" id="103" />
    ...
    <class name="road" id="100" />
  </type>
  <type name="junction" id="4">
    <class name="roundabout" id="401" />
  </type>
</configuration>
```

Detailed description of all possible types and classes can be found here: http://wiki.openstreetmap.org/index.php/Map_features.

When using the osm2pgrouting converter (see later), we take only nodes and ways of types and classes specified in "mapconfig.xml" file to be converted to pgRouting table format:

For Cape Town the OpenStreetMap data is very comprehensive with many details. A compilation of the greater Cape Town area created with JOSM is available as capetown_20080829.osm.

This tool makes it easy to import OpenStreetMap data and use it with pgRouting. It creates topology automatically and creates tables for feature types and road classes. osm2pgrouting was primarily written by Daniel Wendt and is now hosted on the pgRouting project site: http://pgrouting.postlbs.org/wiki/tools/osm2pgrouting

How to install (Ubuntu 8.04)

Check out the latest version from SVN repository:

```
svn checkout http://pgrouting.postlbs.org/svn/pgrouting/tools/osm2pgrouting/trunk osm2pgrouting
Required packages/libraries:

1. PostgreSQL 2. PostGIS 3. pgRouting 4. Boost library 5. Expat library 6. libpq library

Note: if you already compiled pgRouting point 1. to 4. should already be installed.

Then compile

cd osm2pgrouting
make
How to use

1. First you need to create a database and add PostGIS and pgRouting functions:

createdb -U postgres osm
createlang -U postgres plpgsql osm
```

```
psql -U postgres -f /usr/share/postgresql-8.3-postgis/lwpostgis.sql osm
psql -U postgres -f /usr/share/postgresql-8.3-postgis/spatial_ref_sys.sql osm

psql -U postgres -f /usr/share/postlbs/routing_core.sql osm
psql -U postgres -f /usr/share/postlbs/routing_core_wrappers.sql osm
psql -U postgres -f /usr/share/postlbs/routing_topology.sql osm
2. You can define the features and attributes to be imported from the OpenStreetMap XML file in th

3. Open a terminal window and run osm2pgrouting with the following paramters

./osm2pgrouting -file /home/foss4g/capetown_20080829.osm \
                                -conf mapconfig.xml \
                                -dbname osm \
                                -user postgres \
                                -clean
Other available parameters are:

 * required:
        -file   <file>          -- name of your osm xml file
        -dbname <dbname>        -- name of your database
        -user   <user>          -- name of the user, which have write access to the database
        -conf   <file>          -- name of your configuration xml file
 * optional:
        -host   <host>          -- host of your postgresql database (default: 127.0.0.1)
        -port   <port>          -- port of your database (default: 5432)
        -passwd <passwd>        --  password for database access
        -clean                          -- drop peviously created tables
4. Connect to your database and see the tables that have been created

psql -U postgres osm
\d
                     List of relations
 Schema |        Name         |   Type   |  Owner
--------+---------------------+----------+----------
 public | classes             | table    | postgres
 public | geometry_columns    | table    | postgres
 public | nodes               | table    | postgres
 public | spatial_ref_sys     | table    | postgres
 public | types               | table    | postgres
 public | vertices_tmp        | table    | postgres
 public | vertices_tmp_id_seq | sequence | postgres
 public | ways                | table    | postgres
(8 rows)
Note: If tables are missing you might have forgotten to add PostGIS or pgRouting functions to your

Let’s do some more advanced routing with those extra information about road types and road classes
```

CHAPTER

FIVE

LOAD YOUR NETWORK DATA AND CREATE A NETWORK TOPOLOGY

Some network data already comes with a network topology that can be used with pgRouting immediately. But usually the data is in a different format than we need for pgRouting. Often network data is stored in the Shape file format (.shp) and we can use PostGIS' shape2postgresql converter to import the data into the database. OpenStreetMap stores its data as XML and it has its own importing tools for PostgreSQL database.

Later we will use the osm2pgrouting converter. But it does much more than the basic steps for simple routing, so we will start this workshop with the minimum required attributes.

5.1 Load the network data

After creating the workshop database and adding the PostGIS and pgRouting functions to this database (see previous chapter), we load the sample data to our database:

```
psql -U postgres routing
\i /home/foss4g/ways_without_topology.sql
```

Note: The SQL dump file was made from a database which already had PostGIS functions loaded, so it will report errors during import that these functions already exist. You can ignore these errors.

Let's see witch tables have been created:

```
\d

                  List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | geometry_columns | table | postgres
 public | spatial_ref_sys  | table | postgres
 public | ways             | table | postgres
(3 rows)

\d ways

               Table "public.ways"
  Column  |       Type       | Modifiers
----------+------------------+-----------
 gid      | integer          | not null
 length   | double precision |
 name     | character(200)   |
 the_geom | geometry         |
Indexes:
        "ways_pkey" PRIMARY KEY, btree (gid)
```

```
Check constraints:
        "enforce_dims_the_geom" CHECK (ndims(the_geom) = 2)
        "enforce_geotype_the_geom" CHECK (geometrytype(the_geom) =
                                        'MULTILINESTRING'::text OR the_geom IS NULL)
        "enforce_srid_the_geom" CHECK (srid(the_geom) = 4326)
Create network topology
```

Having your data imported into a PostgreSQL database usually requires one more step for pgRouting. You have to make sure that your data provides a correct network topology, which consists of links with source and target ID each.

If your network data doesn't have such network topology information already you need to run the "assign_vertex_id" function. This function assigns a source and a target ID to each link and it can "snap" nearby vertices within a certain tolerance.:

```
assign_vertex_id('<table>', float tolerance, '<geometry column', '<gid>')
```

First we have to add source and target column, then we run the assign_vertex_id function ... and wait.:

```
ALTER TABLE ways ADD COLUMN source integer;
ALTER TABLE ways ADD COLUMN target integer;
SELECT assign_vertex_id('ways', 0.00001, 'the_geom', 'gid');
```

Warning: The dimension of the tolerance parameter depends on your data projection. Usually it's either "degrees" or "meters". Because OSM data has a very good quality for Cape town we can choose a very small "snapping" tolerance: 0.00001 degrees

5.2 Add indices

Fortunately we didn't need to wait too long because the data is small. But your network data might be very large, so it's a good idea to add an index on source, target and geometry column.:

```
CREATE INDEX source_idx ON ways(source);
CREATE INDEX target_idx ON ways(target);
CREATE INDEX geom_idx ON ways USING GIST(the_geom GIST_GEOMETRY_OPS);
```

After these steps our routing database look like this:

```
\d

                    List of relations
 Schema |         Name         |   Type   |  Owner
--------+----------------------+----------+----------
 public | geometry_columns     | table    | postgres
 public | spatial_ref_sys      | table    | postgres
 public | vertices_tmp         | table    | postgres
 public | vertices_tmp_id_seq  | sequence | postgres
 public | ways                 | table    | postgres
(5 rows)


\d ways

              Table "public.ways"
  Column   |       Type       | Modifiers
-----------+------------------+-----------
 gid       | integer          | not null
 length    | double precision |
```

```
 name     | character(200)   |
 the_geom | geometry         |
 source   | integer          |
 target   | integer          |
Indexes:
        "ways_pkey" PRIMARY KEY, btree (gid)
Check constraints:
        "enforce_dims_the_geom" CHECK (ndims(the_geom) = 2)
        "enforce_geotype_the_geom" CHECK (geometrytype(the_geom) =
                                        'MULTILINESTRING'::text OR the_geom IS NULL)
        "enforce_srid_the_geom" CHECK (srid(the_geom) = 4326)
```

Now we are ready for routing with Dijkstra algorithm!

CHAPTER

SIX

SHORTEST PATH SEARCH

Todo

Add chapter introduction for Shortest Path Search

6.1 Dijkstra algorithm

Dijkstra algorithm was the first algorithm implemented in pgRouting. It doesn't require more attributes than source and target ID, and it can distinguish between directed and undirected graphs. You can specify if your network has "reverse cost" or not.

```
shortest_path( sql text,
                   source_id integer,
                   target_id integer,
                   directed boolean,
                   has_reverse_cost boolean )
```

Note:

- Source and target IDs are vertex IDs.
- Undirected graphs ("directed false") ignores "has_reverse_cost" setting
- Shortest Path Dijkstra core function

6.1.1 Core

Each algorithm has its core function (implementation), which is the base for its wrapper functions.

```
SELECT * FROM shortest_path('
                SELECT gid as id,
                         source::integer,
                         target::integer,
                         length::double precision as cost
                        FROM ways',
                10, 20, false, false);
```

```
 vertex_id | edge_id |         cost
-----------+---------+------------------------
        10 |     293 |  0.0059596293824534
         9 |    4632 |  0.0846731039249787
      3974 |    4633 |  0.0765635090514303
      2107 |    4634 |  0.0763951531894937
       ... |     ... |  ...
        20 |      -1 |                    0
(63 rows)
```

6.1.2 Wrapper

Wrapper WITHOUT bounding box

Wrapper functions extend the core functions with transformations, bounding box limitations, etc.. Wrappers can change the format and ordering of the result. They often set default function parameters and make the usage of pgRouting more simple.

```
SELECT gid, AsText(the_geom) AS the_geom
        FROM dijkstra_sp('ways', 10, 20);
```

```
  gid   |                              the_geom
--------+---------------------------------------------------------------
    293 | MULTILINESTRING((18.4074149 -33.9443308,18.4074019 -33.9443833))
   4632 | MULTILINESTRING((18.4074149 -33.9443308,18.4077388 -33.9436183))
   4633 | MULTILINESTRING((18.4077388 -33.9436183,18.4080293 -33.9429733))
    ... | ...
    762 | MULTILINESTRING((18.4241422 -33.9179275,18.4237423 -33.9182966))
    761 | MULTILINESTRING((18.4243523 -33.9177154,18.4241422 -33.9179275))
(62 rows)
```

Wrapper WITH bounding box

You can limit your search area by adding a bounding box. This will improve performance especially for large networks.

```
SELECT gid, AsText(the_geom) AS the_geom
        FROM dijkstra_sp_delta('ways', 10, 20, 0.1);
```

```
   gid  | the_geom
--------+---------------------------------------------------------------
   293  | MULTILINESTRING((18.4074149 -33.9443308,18.4074019 -33.9443833))
   4632 | MULTILINESTRING((18.4074149 -33.9443308,18.4077388 -33.9436183))
   4633 | MULTILINESTRING((18.4077388 -33.9436183,18.4080293 -33.9429733))
   ...  | ...
   762  | MULTILINESTRING((18.4241422 -33.9179275,18.4237423 -33.9182966))
   761 | MULTILINESTRING((18.4243523 -33.9177154,18.4241422 -33.9179275))
(62 rows)
```

Warning: The projection of OSM data is "degree", so we set a bounding box containing start and end vertex plus a 0.1 degree buffer for example.

6.2 A-Star algorithm

A-Star algorithm is another well-known routing algorithm. It adds geographical information to source and target of each network link. This enables the shortest path search to prefer links which are closer to the target of the search.

6.2.1 Prerequisites

For A-Star you need to prepare your network table and add latitute/longitude columns (x1, y1 and x2, y2) and calculate their values.

```
ALTER TABLE ways ADD COLUMN x1 double precision;
ALTER TABLE ways ADD COLUMN y1 double precision;
ALTER TABLE ways ADD COLUMN x2 double precision;
ALTER TABLE ways ADD COLUMN y2 double precision;

UPDATE ways SET x1 = x(startpoint(the_geom));
UPDATE ways SET y1 = y(startpoint(the_geom));

UPDATE ways SET x2 = x(endpoint(the_geom));
UPDATE ways SET y2 = y(endpoint(the_geom));

UPDATE ways SET x1 = x(PointN(the_geom, 1));
UPDATE ways SET y1 = y(PointN(the_geom, 1));

UPDATE ways SET x2 = x(PointN(the_geom, NumPoints(the_geom)));
UPDATE ways SET y2 = y(PointN(the_geom, NumPoints(the_geom)));
```

Note: “endpoint()” function fails for some versions of PostgreSQL (ie. 8.2.5, 8.1.9). A workaround for that problem is using the “PointN()” function instead:

6.2.2 Core

Shortest Path A-Star function is very similar to the Dijkstra function, though it prefers links that are close to the target of the search. The heuristics of this search are predefined, so you need to recompile pgRouting if you want to make changes to the heuristic function itself.

```
shortest_path_astar( sql text,
                     source_id integer,
                     target_id integer,
                     directed boolean,
                     has_reverse_cost boolean )
```

Note:

- Source and target IDs are vertex IDs.
- Undirected graphs (“directed false”) ignores “has_reverse_cost” setting
- Example of A-Star core function

```
SELECT * FROM shortest_path_astar('
                SELECT gid as id,
                         source::integer,
                         target::integer,
                         length::double precision as cost,
                         x1, y1, x2, y2
                        FROM ways',
                10, 20, false, false);
```

```
vertex_id | edge_id |         cost
-----------+---------+---------------------
        10 |     293 |  0.0059596293824534
         9 |    4632 |  0.0846731039249787
      3974 |    4633 |  0.0765635090514303
       ... |     ... |  ...
```

```
      20 |      -1 |                   0
(63 rows)
```

6.2.3 Wrapper

Wrapper function WITH bounding box

Wrapper functions extend the core functions with transformations, bounding box limitations, etc..

```
SELECT gid, AsText(the_geom) AS the_geom
        FROM astar_sp_delta('ways', 10, 20, 0.1);
```

```
  gid   |                              the_geom
--------+---------------------------------------------------------------
    293 | MULTILINESTRING((18.4074149 -33.9443308,18.4074019 -33.9443833))
   4632 | MULTILINESTRING((18.4074149 -33.9443308,18.4077388 -33.9436183))
   4633 | MULTILINESTRING((18.4077388 -33.9436183,18.4080293 -33.9429733))
    ... | ...
    762 | MULTILINESTRING((18.4241422 -33.9179275,18.4237423 -33.9182966))
    761 | MULTILINESTRING((18.4243523 -33.9177154,18.4241422 -33.9179275))
(62 rows)
```

Note: There is currently no wrapper function for A-Star without bounding box, since bounding boxes are very useful to increase performance. If you don't need a bounding box Dijkstra will be enough anyway.

Warning: The projection of OSM data is "degree", so we set a bounding box containing start and end vertex plus a 0.1 degree buffer for example.

6.3 Shooting-Star algorithm

Shooting-Star algorithm is the latest of pgRouting shortest path algorithms. Its speciality is that it routes from link to link, not from vertex to vertex as Dijkstra and A-Star algorithms do. This makes it possible to define relations between links for example, and it solves some other vertex-based algorithm issues like "parallel links", which have same source and target but different costs.

6.3.1 Prerequisites

For Shooting-Star you need to prepare your network table and add the "reverse_cost" and "to_cost" column. Like A-Star this algorithm also has a heuristic function, which prefers links closer to the target of the search.

```
ALTER TABLE ways ADD COLUMN reverse_cost double precision;
UPDATE ways SET reverse_cost = length;

ALTER TABLE ways ADD COLUMN to_cost double precision;

ALTER TABLE ways ADD COLUMN rule text;
```

Shooting-Star algorithm introduces two new attributes

- **rule**: a string with a comma separated list of edge IDs, which describes a rule for turning restriction (if you came along these edges, you can pass through the current one only with the cost stated in to_cost column)

- **to_cost**: a cost of a restricted passage (can be very high in a case of turn restriction or comparable with an edge cost in a case of traffic light)

```
shortest_path_shooting_star( sql text,
                       source_id integer,
                       target_id integer,
                       directed boolean,
                       has_reverse_cost boolean )
```

Note:

- Source and target IDs are link IDs.
- Undirected graphs ("directed false") ignores "has_reverse_cost" setting
- Example for Shooting-Star "rule"

> **Warning:** Shooting* algorithm calculates a path from edge to edge (not from vertex to vertex). Column vertex_id contains start vertex of an edge from column edge_id.

To describe turn restrictions:

```
 gid | source | target | cost | x1 | y1 | x2 | y2 | to_cost | rule
-----+--------+--------+------+----+----+----+----+---------+------
  12 |      3 |     10 |    2 |  4 |  3 |  4 |  5 |    1000 | 14
```

... means that the cost of going from edge 14 to edge 12 is 1000, and

```
 gid | source | target | cost | x1 | y1 | x2 | y2 | to_cost | rule
-----+--------+--------+------+----+----+----+----+---------+------
  12 |      3 |     10 |    2 |  4 |  3 |  4 |  5 |    1000 | 14, 4
```

... means that the cost of going from edge 14 to edge 12 through edge 4 is 1000.

If you need multiple restrictions for a given edge then you have to add multiple records for that edge each with a separate restriction.

```
 gid | source | target | cost | x1 | y1 | x2 | y2 | to_cost | rule
-----+--------+--------+------+----+----+----+----+---------+------
  11 |      3 |     10 |    2 |  4 |  3 |  4 |  5 |    1000 | 4
  11 |      3 |     10 |    2 |  4 |  3 |  4 |  5 |    1000 | 12
```

... means that the cost of going from either edge 4 or 12 to edge 11 is 1000. And then you always need to order your data by gid when you load it to a shortest path function..

6.3.2 Core

```
SELECT * FROM shortest_path_shooting_star('
                SELECT gid as id,
                         source::integer,
                         target::integer,
                         length::double precision as cost,
                         x1, y1, x2, y2,
                         rule, to_cost
                        FROM ways',
                293, 761, false, false);
```

```
 vertex_id | edge_id |         cost
-----------+---------+---------------------
      4232 |     293 |  0.0059596293824534
      3144 |     293 |  0.0059596293824534
      4232 |    4632 |  0.0846731039249787
       ... |     ... |  ...
        51 |     761 |  0.0305298478239596
(63 rows)
```

6.3.3 Wrapper

Wrapper functions extend the core functions with transformations, bounding box limitations, etc..

```
SELECT gid, AsText(the_geom) AS the_geom
        FROM shootingstar_sp('ways', 293, 761, 0.1, 'length', true, true);
```

```
  gid   |                              the_geom
--------+---------------------------------------------------------------
    293 | MULTILINESTRING((18.4074149 -33.9443308,18.4074019 -33.9443833))
    293 | MULTILINESTRING((18.4074149 -33.9443308,18.4074019 -33.9443833))
   4632 | MULTILINESTRING((18.4074149 -33.9443308,18.4077388 -33.9436183))
    ... | ...
    762 | MULTILINESTRING((18.4241422 -33.9179275,18.4237423 -33.9182966))
    761 | MULTILINESTRING((18.4243523 -33.9177154,18.4241422 -33.9179275))
(62 rows)
```

Note: There is currently no wrapper function for A-Star without bounding box, since bounding boxes are very useful to increase performance. If you don't need a bounding box Dijkstra will be enough anyway.

Warning: The projection of OSM data is "degree", so we set a bounding box containing start and end vertex plus a 0.1 degree buffer for example.

CHAPTER

SEVEN

ADVANCED USAGE OF PGROUTING SHORTEST PATH SEARCH

An ordinary shortest path query with result usualy looks like this:

```
SELECT * FROM shortest_path_shooting_star(
        'SELECT gid as id, source, target, length as cost, x1, y1, x2, y2, rule,
        to_cost, reverse_cost FROM ways', 1955, 5787, true, true);
```

Query result:

```
 vertex_id | edge_id |         cost
-----------+---------+---------------------
      8134 |    1955 | 0.00952475464810279
      5459 |    1956 |  0.0628075563112871
      8137 |    1976 |  0.0812786367080268
      5453 |     758 |  0.0421747270358272
      5456 |    3366 |  0.0104935732514831
     11086 |    3367 |   0.113400030221047
      4416 |     306 |   0.111600379959229
      4419 |     307 |  0.0880411972519595
      4422 |    4880 |  0.0208599114366633
      5101 |     612 |  0.0906859882381495
      5102 |    5787 |     80089.8820919459
(11 rows)
```

That is usually called *SHORTEST* path, which means that a length of an edge is its cost.

Costs can be anything (“Weighted costs”)

But in real networks we have different limitations or preferences for different road types for example. In other words, we want to calculate CHEAPEST path - a path with a minimal cost. There is no limitation in what we take as costs.

When we convert data from OSM format using the osm2pgrouting tool, we get these two additional tables for road types and classes:

```
\d classes

  id |   name
-----+------------
   2 | cycleway
   1 | highway
   4 | junction
   3 | tracktype
```

```
\d types

 id   | type_id |         name         |  cost
-----+---------+--------------------+--------
 201  |       2 | lane                 |   1
 204  |       2 | opposite             |   1
 203  |       2 | opposite_lane        |   1
 202  |       2 | track                |   1
 117  |       1 | bridleway            |   1
 113  |       1 | bus_guideway         |   1
 118  |       1 | byway                |   1
 115  |       1 | cicleway             |   1
 116  |       1 | footway              |   1
 108  |       1 | living_street        |   1
 101  |       1 | motorway             |   0.2
 103  |       1 | motorway_junction    |   0.2
 102  |       1 | motorway_link        |   0.2
 114  |       1 | path                 |   100
 111  |       1 | pedestrian           |   100
 106  |       1 | primary              |   100
 107  |       1 | primary_link         |   100
 107  |       1 | residential          |   100
 100  |       1 | road                 |   0.7
 100  |       1 | unclassified         |   0.7
 106  |       1 | secondary            |   10
 109  |       1 | service              |   10
 112  |       1 | services             |   10
 119  |       1 | steps                |   10
 107  |       1 | tertiary             |   10
 110  |       1 | track                |   10
 104  |       1 | trunk                |   10
 105  |       1 | trunk_link           |   10
 401  |       4 | roundabout           |   10
 301  |       3 | grade1               |   15
 302  |       3 | grade2               |   15
 303  |       3 | grade3               |   15
 304  |       3 | grade4               |   15
 305  |       3 | grade5               |   15
```

Road class is linked with the ways table by class_id field. Cost values for classes table are assigned arbitrary.

```
UPDATE classes SET cost=15 WHERE id>300;
```

For better performance it is worth to create an index on id field of classes table.

```
CREATE INDEX class_idx ON ways (id);
```

The idea behind these two tables is to specify a factor to be multiplied with the cost of each link (usually length):

```
SELECT * FROM shortest_path_shooting_star(
        'SELECT gid as id, class_id, source, target, length*c.cost as cost,
                x1, y1, x2, y2, rule, to_cost, reverse_cost*c.cost as reverse_cost
        FROM ways w, classes c
        WHERE class_id=c.id', 1955, 5787, true, true);
```

Query result:

```
 vertex_id | edge_id |         cost
-----------+---------+---------------------
      8134 |    1955 | 0.00666732825367195
      5459 |    1956 |   0.043965289417901
```

```
      8137 |    1992 |   0.126646230936747
      5464 |     762 |   0.827868704808978
      5467 |     763 |    0.16765902528648
      ... |      ... |                ...
      9790 |    5785 | 0.00142107468268373
      8548 |    5786 | 0.00066608685984761
      16214 |    5787 |  0.0160179764183892
(69 rows)
```

We can see that the shortest path result is completely different from the example before. We call this "weighted costs".

Another example is to restrict access to roads of a certain type:

```
UPDATE classes SET cost=100000 WHERE name LIKE 'motorway%';
```

Through subqueries you can "mix" your costs as you like and this will change the results of your routing request immediately. Cost changes will affect the next shortest path search, and there is no need to rebuild your network.

CHAPTER

EIGHT

SERVER SIDE SCRIPTS WITH PHP

We will use a PHP script to make the routing query and send the result back to the web client.

The following steps are necessary:

Retrieve the start and end point coordinates. Find the closest edge to start/end point. Take either the start or end vertex of this edge (for Dijkstra/ A-Star) or the complete edge (Shooting-Star) as start of the route and end respectively. Make the Shortest Path database query. Transform the query result to XML and send it back to the web client.

CHAPTER

NINE

GEOEXT BROWSER CLIENT