Workshop - FOSS4G routing with pgRouting

Release 4

Daniel Kastl, Frédéric Junod

July 03, 2014

Contents

CHAPTER 1

Introduction

Abstract

pgRouting adds routing functionality to PostGIS. This introductory workshop will show you how. It gives a practical example of how to use the new pgRouting release with OpenStreetMap road network data. It explains the steps to prepare the data, make routing queries, assign costs, write a custom function 'plpgsql' function and use the new OpenLayers 3 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 extendable 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. You will learn how to convert the data into the required format and how to calibrate the data with "cost" attributes. Furthermore we will tell you what else pgRouting provides beside "Dijkstra", "A-Star" and "Shooting-Star" and what has been added recently to the library. 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 OpenLayers 3. 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. Writing a custom PostgreSQL stored procedure in 'plpgsql' will allow us to make shortest path queries through Geoserver in a convenient way.

Prerequisites

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

Presenters and Authors

- *Daniel Kastl* is founder and CEO of Georepublic and works in Germany and Japan. He is moderating and promoting the pgRouting community and development since the beginning of the project, and he's an active OSM contributor in Japan.

- *Frédéric Junod* works at the Swiss office of Camptocamp for about six 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.

License

Supported by

Camptocamp

Georepublic

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

adds routing and other network analysis functionality. A predecessor of pgRouting – pgDijkstra, written by Sylvain Pasche from Camptocamp, was later extended by Orkney and renamed to pgRouting. The project is now supported and maintained by Georepublic, iMaptools and a broad user community.

pgRouting is an OSGeo Labs project of the OSGeo Foundation and included on OSGeo Live.

pgRouting provides functions for:

- All Pairs Shortest Path, Johnson's Algorithm [1]
- All Pairs Shortest Path, Floyd-Warshall Algorithm [1]
- Shortest Path A*
- Bi-directional Dijkstra Shortest Path [1]
- Bi-directional A* Shortest Path [1]
- Shortest Path Dijkstra
- Driving Distance
- K-Shortest Path, Multiple Alternative Paths [1]
- K-Dijkstra, One to Many Shortest Path [1]
- Traveling Sales Person
- Turn Restriction Shortest Path (TRSP) [1]

- Shortest Path Shooting Star [2]

Advantages of the database routing approach are:

- Data and attributes can be modified by many clients, like QGIS" and uDig through JDBC, ODBC, or directly using Pl/pgSQL. The clients can either be PCs or mobile devices.
- 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 and is supported by a growing community of individuals, businesses and organizations.

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

[1] **New** in pgRouting 2.0.0
[2] Discontinued in pgRouting 2.0.0

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://www.pgrouting.org/docs/tools/osm2pgrouting.html

2.4 OpenLayers 3

OpenLayers 3 brings geospatial data to any modern desktop or mobile web browser. ol3 is a complete rewrite, featuring WebGL and 3D. Like OpenLayers 2, it supports a huge variety of data formats and layer types. But unlike OpenLayers 2, it is built from scratch relying on latest browser technologies like HTML5, WebGL and CSS3.

OpenLayers 3 website: http://www.ol3js.org

CHAPTER 3

Installation and Requirements

For this workshop you need:

- Preferable a Linux operating system like Ubuntu
- An editor like Gedit or Mousepad
- Geoserver for the routing application
- Internet connection

All required tools are available on the OSGeo LiveDVD, so the following reference is a quick summary of how to install it on your own computer running Ubuntu 12.04 or later.

3.1 pgRouting

pgRouting on Ubuntu can be done using packages from a Launchpad repository:

All you need to do is to open a terminal window and run:

```
# Add pgRouting launchpad repository
sudo apt-add-repository -y ppa:ubuntugis/ppa
sudo apt-add-repository -y ppa:georepublic/pgrouting
sudo apt-get update

# Install pgRouting package
sudo apt-get install postgresql-9.3-pgrouting

# Install osm2pgrouting package
sudo apt-get install osm2pgrouting

# Install workshop material (optional, but maybe slightly outdated)
sudo apt-get install pgrouting-workshop

# For workshops at conferences and events:
# Download and install from http://trac.osgeo.org/osgeo/wiki/Live_GIS_Workshop_Install
wget --no-check-certificate https://launchpad.net/~georepublic/+archive/pgrouting/+files/pgroutin
sudo dpkg -i pgrouting-workshop_[version]_all.deb
```

This will also install all required packages such as PostgreSQL and PostGIS if not installed yet.

Note:

- Once pgRouting 2.0 has been released it will be available in the `stable` repository on Launchpad.

- To be up-to-date with changes and improvements you might run `sudo apt-get update & sudo apt-get upgrade` from time to time, especially if you use an older version of the LiveDVD.
- To avoid permission denied errors for local users you can set connection method to `trust` in `/etc/postgresql/<version>/main/pg_hba.conf` and restart PostgreSQL server with `sudo service postgresql restart`.

```
local   all             postgres                                trust
local   all             all                                     trust
host    all             all             127.0.0.1/32            trust
host    all             all             ::1/128                 trust
```

`pg_hba.conf` can be only edited with "superuser" rights, ie. from the terminal window with

```
sudo nano /etc/postgresql/9.3/main/pg_hba.conf
```

To close the editor again hit `CTRL-X`.

3.2 Workshop

When you installed the workshop package you will find all documents in `/usr/share/pgrouting/workshop/`.

We recommend to copy the files to your home directory and make a symbolic link to your webserver's root folder:

```
cp -R /usr/share/pgrouting/workshop ~/Desktop/pgrouting-workshop
sudo ln -s ~/Desktop/pgrouting-workshop /var/www/pgrouting-workshop
```

You can then find all workshop files in the `pgrouting-workshop` folder and access to

- Web directory: http://localhost/pgrouting-workshop/web/
- Online manual: http://localhost/pgrouting-workshop/docs/html/

Note: Additional sample data is available in the workshop `data` directory. To extract the file run `tar -xzf ~/Desktop/pgrouting-workshop/data.tar.gz`.

3.3 Add pgRouting Functions to database

Since **version 2.0** pgRouting functions can be easily installed as extension. This requires:

- PostgreSQL 9.1 or higher
- PostGIS 2.x installed as extension

If these requirements are met, then open a terminal window and execute the following commands (or run these commands in pgAdmin 3:

```
# login as user "postgres"
psql -U postgres

# create routing database
CREATE DATABASE routing;
\c routing

# add PostGIS functions
CREATE EXTENSION postgis;

# add pgRouting core functions
CREATE EXTENSION pgrouting;
```

Note: If you're looking for the SQL files containing pgRouting function, you can find them in `/usr/share/postgresql/<version>/contrib/pgrouting-2.0/`:

```
-rw-r--r-- 1 root root  4126 Jun 18 22:30 pgrouting_dd_legacy.sql
-rw-r--r-- 1 root root 43642 Jun 18 22:30 pgrouting_legacy.sql
-rw-r--r-- 1 root root 40152 Jun 18 22:30 pgrouting.sql
```

3.4 Data

The pgRouting workshop will make use of OpenStreetMap data, which is already available on the LiveDVD. If you don't use the LiveDVD or want to download the latest data or the data of your choice, you can make use of OpenStreetMap's API from your terminal window:

```
# Download using Overpass XAPI (larger extracts possible than with default OSM API)
BBOX="-1.2,52.93,-1.1,52.985"
wget --progress=dot:mega -O "sampledata.osm" "http://www.overpass-api.de/api/xapi?*[bbox=${BBOX}]
```

More information how to get OSM data:

- OpenStreetMap download information in http://wiki.openstreetmap.org/wiki/Downloading_data
- OpenStreetMap data is available at the LiveDVD in `/usr/local/share/osm/`

An alternative for very large areas is the download services of Geofabrik. Download a country extract and unpack the data like this:

```
wget --progress=dot:mega http://download.geofabrik.de/[path/to/file].osm.bz2
bunzip2 [file].osm.bz2
```

Warning: Data of a whole country might be too big for the LiveDVD as well as processing time might take very long.

CHAPTER 4

Create a Network Topology

osm2pgrouting is a convenient tool, but it's also a *black box*. There are several cases where *osm2pgrouting* can't be used. Obviously if the data isn't OpenStreetMap data. Some network data already comes with a network topology that can be used with pgRouting out-of-the-box. Often network data is stored in Shape file format (`.shp`) and we can use PostGIS' `shape2postgresql` converter to import the data into a PostgreSQL database. But what to do then?

In this chapter you will learn how to create a network topology from scratch. For that we will start with data that contains the minimum attributes needed for routing and show how to proceed step-by-step to build routable data for pgRouting.

4.1 Load network data

At first we will load a database dump from the workshop `data` directory. This directory contains a compressed file with database dumps as well as a small size network data. If you haven't uncompressed the data yet, extract the file by

```
cd ~/Desktop/pgrouting-workshop/
tar -xvzf data.tar.gz
```

The following command will import the database dump. It will add PostGIS and pgRouting functions to a database, in the same way as decribed in the previous chapter. It will also load the sample data with a minimum number of attributes, which you will usually find in any network data:

```
# Optional: Drop database
dropdb -U postgres pgrouting-workshop

# Load database dump file
psql -U postgres -f ~/Desktop/pgrouting-workshop/data/sampledata_notopo.sql
```

Let's see wich tables have been created:

Run: `psql -U postgres -d pgrouting-workshop -c "\d"`

```
              List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | geography_columns | view  | postgres
 public | geometry_columns  | view  | postgres
 public | raster_columns    | view  | postgres
 public | raster_overviews  | view  | postgres
 public | spatial_ref_sys   | table | postgres
 public | ways              | table | postgres
(7 rows)
```

The table containing the road network data has the name `ways`. It consists of the following attributes:

Run: `psql -U postgres -d pgrouting-workshop -c "\d ways"`

```
              Table "public.ways"
  Column  |           Type             | Modifiers
----------+----------------------------+-----------
 gid      | bigint                     |
 class_id | integer                    | not null
 length   | double precision           |
 name     | character(200)             |
 osm_id   | bigint                     |
 the_geom | geometry(LineString,4326) |
Indexes:
    "ways_gid_idx" UNIQUE, btree (gid)
    "geom_idx" gist (the_geom)
```

It is common that road network data provides at least the following information:

- Road link ID (gid)
- Road class (class_id)
- Road link length (length)
- Road name (name)
- Road geometry (the_geom)

This allows to display the road network as a PostGIS layer in GIS software, for example in QGIS. Though it is not sufficient for routing, because it doesn't contain network topology information.

For the next steps we need to start the PostgreSQL command line tool

```
psql -U postgres pgrouting-workshop
```

... or use PgAdmin III.

4.2 Calculate 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 information about source and target ID of each road link.

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

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

First we have to add source and target column, then we run the `pgr_createTopology` function ... and wait. Depending on the network size this process may take from minutes to hours. It will also require enough memory (RAM or SWAP partition) to store temporary data.

```
-- Add "source" and "target" column
ALTER TABLE ways ADD COLUMN "source" integer;
ALTER TABLE ways ADD COLUMN "target" integer;

-- Run topology function
SELECT pgr_createTopology('ways', 0.00001, 'the_geom', 'gid');
```

Note: Execute `psql -U postgres -d pgrouting-workshop` in your terminal to connect to the database and start the PostgreSQL shell. Leave the shell with `\q` command.

Warning: The dimension of the tolerance parameter depends on your data projection. Usually it's either "degrees" or "meters".

4.3 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 to `source` and `target` column.

```
CREATE INDEX source_idx ON ways("source");
CREATE INDEX target_idx ON ways("target");
```

After these steps our routing database looks like this:

Run: `\d`

```
                List of relations
 Schema |        Name          |   Type   |  Owner
--------+----------------------+----------+----------
 public | geography_columns    | view     | postgres
 public | geometry_columns     | view     | postgres
 public | raster_columns       | view     | postgres
 public | raster_overviews     | view     | postgres
 public | spatial_ref_sys      | table    | postgres
 public | vertices_tmp         | table    | postgres
 public | vertices_tmp_id_seq | sequence | postgres
 public | ways                 | table    | postgres
(9 rows)
```

- `geography_columns` should contain a record for each table with "geometry" attribute and its SRID.
- `vertices_tmp` contains a list of all network nodes.

Run: `\d ways`

```
                Table "public.ways"
  Column   |            Type             | Modifiers
-----------+-----------------------------+-----------
 gid       | integer                     |
 class_id  | integer                     | not null
 length    | double precision            |
 name      | text                        |
 osm_id    | bigint                      |
 the_geom  | geometry(LineString,4326)   |
 source    | integer                     |
 target    | integer                     |
Indexes:
    "ways_gid_idx" UNIQUE, btree (gid)
    "geom_idx" gist (the_geom)
    "source_idx" btree (source)
    "target_idx" btree (target)
```

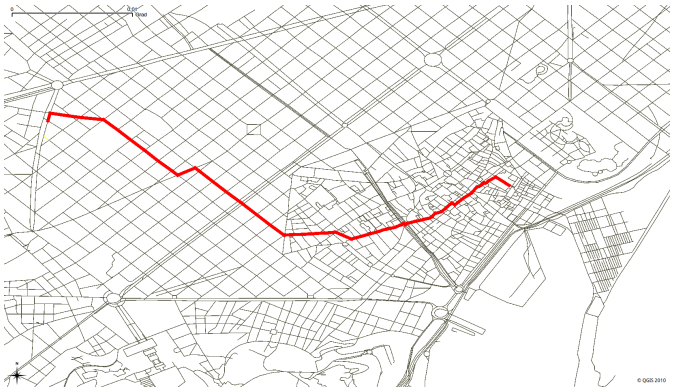
- `source` and `target` columns are now updated with node IDs.
- `name` may contain the street name or be empty.
- `length` is the road link length in kilometers.

Now we are ready for our first routing query with *Dijkstra algorithm*!

CHAPTER 5

pgRouting Algorithms

pgRouting was first called *pgDijkstra*, because it implemented only shortest path search with *Dijkstra* algorithm. Later other functions were added and the library was renamed.



This chapter will explain selected pgRouting algorithms and which attributes are required.

Note: If you run *osm2pgrouting* tool to import *OpenStreetMap* data, the `ways` table contains all attributes already to run all shortest path functions. The `ways` table of the `pgrouting-workshop` database of the *previous chapter* is missing several attributes instead, which are listed as **Prerequisites** in this chapter.

5.1 Shortest Path Dijkstra

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

Prerequisites

To be able to use `reverse cost` you need to add an additional cost column. We can set reverse cost as length.

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

Description

Returns a set of `pgr_costResult` (seq, id1, id2, cost) rows, that make up a path.

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

Parameters

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

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

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

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

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

source `int4` id of the start point

target `int4` id of the end point

directed `true` if the graph is directed

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

Returns set of `pgr_costResult`:

seq row sequence

id1 node ID

id2 edge ID (`-1` for the last row)

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

Note:

- Many pgRouting functions have `sql::text` as one of their arguments. While this may look confusing at first, it makes the functions very flexible as the user can pass any `SELECT` statement as function argument as long as the returned result contains the required number of attributes and the correct attribute names.
- Dijkstra algorithm does not require the network geometry.
- The function does not return a geometry, but only an ordered list of nodes.

Example query

`pgr_costResult` is a common result type used by several pgRouting functions. In the case of `pgr_dijkstra` the first column is a sequential ID, followed by node ID, edge ID and cost to pass this edge.

```
SELECT seq, id1 AS node, id2 AS edge, cost FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
```

```
                    FROM ways',
                30, 60, false, false);
```

Query result

```
 seq | node | edge |        cost
-----+------+------+---------------------
   0 |   30 |   53 |  0.0591267653820616
   1 |   44 |   52 |  0.0665408320949312
   2 |   14 |   15 |  0.0809556879332114
   ...
   6 |   10 | 6869 |  0.0164274192597773
   7 |   59 |   72 |  0.0109385169537801
   8 |   60 |   -1 |                    0
(9 rows)
```

Note:

- With more complex SQL statements, using JOINs for example, the result may be in a wrong order. In that case `ORDER BY seq` will ensure that the path is in the right order again.
- The returned cost attribute represents the cost specified in the `sql::text` argument. In this example cost is `length` in unit "kilometers". Cost may be time, distance or any combination of both or any other attributes or a custom formula.

5.2 Shortest Path A*

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

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 = ST_x(ST_PointN(the_geom, 1));
UPDATE ways SET y1 = ST_y(ST_PointN(the_geom, 1));

UPDATE ways SET x2 = ST_x(ST_PointN(the_geom, ST_NumPoints(the_geom)));
UPDATE ways SET y2 = ST_y(ST_PointN(the_geom, ST_NumPoints(the_geom)));
```

Note:

- A bug in a previous version of PostGIS didn't allow the use of `ST_startpoint` or `ST_endpoint`.
- From PostGIS 2.x `ST_startpoint` and `ST_endpoint` are only valid for `LINESTRING` geometry type and will fail with `MULTILINESTING`.

Therefor a slightly more difficult looking query is used. If the network data really contains multi-geomtery linestrings the query might give the wrong start and end point. But in general data has been imported as `MULTILINESTING` even if it only contains `LINESTRING` geometries.

Description

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.

Returns a set of `pgr_costResult` (seq, id1, id2, cost) rows, that make up a path.

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

Parameters

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

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

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

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

x1 `x` coordinate of the start point of the edge

y1 `y` coordinate of the start point of the edge

x2 `x` coordinate of the end point of the edge

y2 `y` coordinate of the end point of the edge

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

source `int4` id of the start point

target `int4` id of the end point

directed `true` if the graph is directed

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

Returns set of `pgr_costResult`:

seq row sequence

id1 node ID

id2 edge ID (`-1` for the last row)

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

Example query

```
SELECT seq, id1 AS node, id2 AS edge, cost FROM pgr_astar('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost,
```

```
                        x1, y1, x2, y2
                       FROM ways',
                30, 60, false, false);
```

Query result

```
 seq | node | edge |        cost
-----+------+------+--------------------
   0 |   30 |   53 |  0.0591267653820616
   1 |   44 |   52 |  0.0665408320949312
   2 |   14 |   15 |  0.0809556879332114
   ...
   6 |   10 | 6869 |  0.0164274192597773
   7 |   59 |   72 |  0.0109385169537801
   8 |   60 |   -1 |                    0
(9 rows)
```

Note:

- The result of Dijkstra and A-Star are the same, which should be the case.
- A-Star is supposed to be faster than Dijkstra algorithm as the network size is getting larger. But in case of pgRouting the algorithm speed advantage does not matter really compared the time required to select the network data and build the graph.

5.3 Multiple Shortest Paths with kDijkstra

The kDijkstra functions are very similar to the Dijkstra function but they allow to set multiple destinations with a single function call.

Prerequisites

kDijkstra doesn't require additional attributes to Dijkstra algorithm.

Description

If the main goal is to calculate the total cost, for example to calculate multiple routes for a distance matrix, then `pgr_kdijkstraCost` returns a more compact result. In case the paths are important `pgr_kdijkstraPath` function returns a result similar to A* or Dijkstra for each destination.

Both functions return a set of `pgr_costResult` (seq, id1, id2, cost) rows, that summarize the path cost or return the paths.

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

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

Parameters

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

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

id `int4` identifier of the edge

source `int4` identifier of the source vertex

target `int4` identifier of the target vertex

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

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

source `int4` id of the start point

targets `int4[]` an array of ids of the end points

directed `true` if the graph is directed

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

`pgr_kdijkstraCost` returns set of `pgr_costResult`:

seq row sequence

id1 path vertex source id (this will always be source start point in the query).

id2 path vertex target id

cost cost to traverse the path from `id1` to `id2`. Cost will be -1.0 if there is no path to that target vertex id.

`pgr_kdijkstraPath` returns set of `pgr_costResult`:

seq row sequence

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

id2 path edge id

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

Example query `pgr_kdijkstraCost`

```
SELECT seq, id1 AS source, id2 AS target, cost FROM pgr_kdijkstraCost('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                10, array[60,70,80], false, false);
```

Query result

```
 seq | source | target |       cost
-----+--------+--------+------------------
   0 |     10 |     60 | 13.4770181770774
   1 |     10 |     70 | 16.9231630493294
   2 |     10 |     80 | 17.7035050077573
(3 rows)
```

Example query `pgr_kdijkstraPath`

```
SELECT seq, id1 AS path, id2 AS edge, cost FROM pgr_kdijkstraPath('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                10, array[60,70,80], false, false);
```

Query result

```
 seq | path | edge |        cost
-----+------+------+---------------------
   0 |   60 | 3163 |   0.427103399132954
   1 |   60 | 2098 |   0.441091435851107
...
  40 |   60 |   56 |  0.0452819891352444
  41 |   70 | 3163 |   0.427103399132954
  42 |   70 | 2098 |   0.441091435851107
...
 147 |   80 |  226 |  0.0730263299529259
 148 |   80 |  227 |  0.0741906229622583
(149 rows)
```

There are many other functions available with the new pgRouting 2.0 release, but most of them work in a similar way, and it would take too much time to mention them all in this workshop. For the complete list of pgRouting functions see the API documentation: http://docs.pgrouting.org/

CHAPTER 6

osm2pgrouting Import Tool

osm2pgrouting is a command line tool that allows 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 currently hosted on the pgRouting project site: http://www.pgrouting.org/docs/tools/osm2pgrouting.html

---

Note: There are some limitations, especially regarding the network size. The current version of osm2pgrouting needs to load all data into memory, which makes it fast but also requires a lot or memory for large datasets. An alternative tool to osm2pgrouting without the network size limitation is **osm2po** (http://osm2po.de). It's available under "Freeware License".

Raw OpenStreetMap data contains much more features and information than need for routing. Also the format is not suitable for pgRouting out-of-the-box. An `.osm` XML file consists of three major feature types:

- nodes
- ways
- relations

The data of sampledata.osm for example looks like this:

```
<?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>
```

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

When using osm2pgrouting, we take only nodes and ways of types and classes specified in `mapconfig.xml` file that will be imported into the routing database:

```
<?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>
```

The default `mapconfig.xml` is installed in `/usr/share/osm2pgrouting/`.

6.1 Create routing database

Before we can run osm2pgrouting we have to create a database and load PostGIS and pgRouting functions into this database. If you have installed the template databases as described in the previous chapter, creating a pgRouting-ready database is done with a single command. Open a terminal window and run:

```
# Optional: Drop database
dropdb -U postgres pgrouting-workshop

# Create a new routing database
createdb -U postgres pgrouting-workshop
psql -U postgres -d pgrouting-workshop -c "CREATE EXTENSION postgis;"
psql -U postgres -d pgrouting-workshop -c "CREATE EXTENSION pgrouting;"
```

... and you're done.

Alternativly you can use **PgAdmin III** and SQL commands. Start PgAdmin III (available on the LiveDVD), connect to any database and open the SQL Editor and then run the following SQL command:

```
-- create pgrouting-workshop database
CREATE DATABASE "pgrouting-workshop";
\c pgrouting-workshop

CREATE EXTENSION postgis;
CREATE EXTENSION pgrouting;
```

6.2 Run osm2pgrouting

The next step is to run `osm2pgrouting` converter, which is a command line tool, so you need to open a terminal window.

We take the default `mapconfig.xml` configuration file and the `pgrouting-workshop` database we created before. Furthermore we take `~/Desktop/pgrouting-workshop/data/sampledata.osm` as raw data. This file contains only OSM data for a small area to speed up data processing time.

The workshop data is available as compressed file, which needs to be extracted first either using file manager or with this command:

```
cd ~/Desktop/pgrouting-workshop/
tar -xvzf data.tar.gz
```

Then run the converter:

```
osm2pgrouting -file "data/sampledata.osm" \
                          -conf "/usr/share/osm2pgrouting/mapconfig.xml" \
                          -dbname pgrouting-workshop \
                          -user postgres \
                          -clean
```

List of all possible parameters:

Parameter	Value	Description	Required
-file	<file>	name of your osm xml file	yes
-dbname	<dbname>	name of your database	yes
-user	<user>	name of the user, which have write access to the database	yes
-conf	<file>	name of your configuration xml file	yes
-host	<host>	host of your postgresql database (default: 127.0.0.1)	no
-port	<port>	port of your database (default: 5432)	no
-passwd	<passwd>	password for database access	no
-prefixtables	<prefix>	add at the beginning of table names	no
-skipnodes		don't import the nodes table	no
-clean		drop peviously created tables	no

Note:

- There might be an updated version of osm2pgrouting available. To update the package run:

```
sudo apt-get update
sudo apt-get install --only-upgrade osm2pgrouting
```

- If you get permission denied error for postgres users you can set connection method to `trust` in `/etc/postgresql/<version>/main/pg_hba.conf` and restart PostgreSQL server with `sudo service postgresql restart`.

Depending on the size of your network the calculation and import may take a while. After it's finished connect to your database and check the tables that should have been created:

Run: `psql -U postgres -d pgrouting-workshop -c "\d"`

If everything went well the result should look like this:

```
                List of relations
 Schema |         Name          |   Type   |  Owner
--------+-----------------------+----------+----------
 public | classes               | table    | postgres
 public | geography_columns     | view     | postgres
 public | geometry_columns      | view     | postgres
 public | nodes                 | table    | postgres
 public | raster_columns        | view     | postgres
 public | raster_overviews      | view     | postgres
 public | relation_ways         | table    | postgres
 public | relations             | 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 | way_tag               | table    | postgres
 public | ways                  | table    | postgres
(14 rows)
```

Note: osm2pgrouting creates more tables and imports more attributes than we will use in this workshop. Some of them have been just added recently and are still lacking proper documentation.

CHAPTER 7

Advanced Routing Queries

Note: This chapter may be skipped depending on available time, or you can come back here again later

As explained in the *chapter about routing algorithms* a shortest path query usualy looks like this:

```
SELECT seq, id1 AS node, id2 AS edge, cost FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                30, 60, false, false);
```

This is usually called **shortest** path, which means that a length of an edge is its cost. But cost doesn't need to be length, cost can be almost anything, for example time, slope, surface, road type, etc.. Or it can be a combination of multiple parameters ("Weighted costs").

Note: If you want to proceed with a routing database containing pgRouting functions, sample data and all required attributes, you can load the following database dump file.

```
# Optional: Drop database
dropdb -U postgres pgrouting-workshop

# Load database dump file
psql -U postgres -f ~/Desktop/pgrouting-workshop/data/sampledata_routing.sql
```

7.1 Weighted costs

In "real" networks there are different limitations or preferences for different road types for example. In other words, we don't want to get the *shortest* but the **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 two additional tables for road `types` and road `classes`:

Note: We switch now to the database we previously generated with osm2pgrouting. From within PostgreSQL shell this is possible with the `\c routing` command.

Run: `SELECT * FROM types ORDER BY id;`

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

Run: `SELECT * FROM classes ORDER BY id;`

```
 id  | type_id |       name         | cost | priority | default_maxspeed
-----+---------+--------------------+------+----------+------------------
 100 |       1 | road               |      |        1 |               50
 101 |       1 | motorway           |      |        1 |               50
 102 |       1 | motorway_link      |      |        1 |               50
 103 |       1 | motorway_junction  |      |        1 |               50
 104 |       1 | trunk              |      |        1 |               50
 105 |       1 | trunk_link         |      |        1 |               50
 106 |       1 | primary            |      |        1 |               50
 107 |       1 | primary_link       |      |        1 |               50
 108 |       1 | secondary          |      |        1 |               50
 109 |       1 | tertiary           |      |        1 |               50
 110 |       1 | residential        |      |        1 |               50
 111 |       1 | living_street      |      |        1 |               50
 112 |       1 | service            |      |        1 |               50
 113 |       1 | track              |      |        1 |               50
 114 |       1 | pedestrian         |      |        1 |               50
 115 |       1 | services           |      |        1 |               50
 116 |       1 | bus_guideway       |      |        1 |               50
 117 |       1 | path               |      |        1 |               50
 118 |       1 | cycleway           |      |        1 |               50
 119 |       1 | footway            |      |        1 |               50
 120 |       1 | bridleway          |      |        1 |               50
 121 |       1 | byway              |      |        1 |               50
 122 |       1 | steps              |      |        1 |               50
 123 |       1 | unclassified       |      |        1 |               50
 124 |       1 | secondary_link     |      |        1 |               50
 125 |       1 | tertiary_link      |      |        1 |               50
 201 |       2 | lane               |      |        1 |               50
 202 |       2 | track              |      |        1 |               50
 203 |       2 | opposite_lane      |      |        1 |               50
 204 |       2 | opposite           |      |        1 |               50
 301 |       3 | grade1             |      |        1 |               50
 302 |       3 | grade2             |      |        1 |               50
 303 |       3 | grade3             |      |        1 |               50
 304 |       3 | grade4             |      |        1 |               50
 305 |       3 | grade5             |      |        1 |               50
 401 |       4 | roundabout         |      |        1 |               50
(36 rows)
```

The road class is linked with the ways table by `class_id` field. After importing data the `cost` attribute is not set yet. Its values can be changed with an `UPDATE` query. In this example cost values for the classes table are assigned arbitrary, so we execute:

```
UPDATE classes SET cost=1 ;
UPDATE classes SET cost=2.0 WHERE name IN ('pedestrian','steps','footway');
UPDATE classes SET cost=1.5 WHERE name IN ('cicleway','living_street','path');
UPDATE classes SET cost=0.8 WHERE name IN ('secondary','tertiary');
UPDATE classes SET cost=0.6 WHERE name IN ('primary','primary_link');
```

```
UPDATE classes SET cost=0.4 WHERE name IN ('trunk','trunk_link');
UPDATE classes SET cost=0.3 WHERE name IN ('motorway','motorway_junction','motorway_link');
```

For better performance, especially if the network data is large, it is better to create an index on the `class_id` field of the ways table and eventually on the `id` field of the `types` table.

```
CREATE INDEX ways_class_idx ON ways (class_id);
CREATE INDEX classes_idx ON classes (id);
```

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

```
SELECT seq, id1 AS node, id2 AS edge, cost FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length * c.cost AS cost
                        FROM ways, classes c
                        WHERE class_id = c.id',
                30, 60, false, false);
```

7.2 Restricted access

Another possibility is to restrict access to roads of a certain type by either setting a very high cost for road links with a certain attribute or by not selecting certain road links at all:

```
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.

Of course certain road classes can be excluded in the `WHERE` clause of the query as well, for example exclude "living_street" class:

```
SELECT seq, id1 AS node, id2 AS edge, cost FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length * c.cost AS cost
                        FROM ways, classes c
                        WHERE class_id = c.id AND class_id != 111',
                30, 60, false, false);
```

Of course pgRouting allows you all kind of SQL that is possible with PostgreSQL/PostGIS.

CHAPTER 8

Writing a pl/pgsql Wrapper

Many pgRouting functions provide a "low-level" interface to algorithms and for example return ordered ID's rather than routes with geometries. "Wrapper functions" therefor offer different input parameters as well as transform the returned result into a format, that can be easier read or consumed by applications.

The downside of wrapper functions is, that they often make assumptions that make them only useful for specific use cases or network data. Therefor pgRouting has decided to only support low-level functions and let the user write their own wrapper functions for their own use cases.

The following wrappers are examples for common transformations:

8.1 Return route with network geometry

To return a route with the line geometry of it's path segments it's not necessary to write a wrapper function. It's sufficient to link the result pack to the original road network table:

Shortest Path Dijkstra

```
SELECT seq, id1 AS node, id2 AS edge, cost FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                30, 60, false, false);
```

Result with Geometries

```
SELECT seq, id1 AS node, id2 AS edge, cost, b.the_geom FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                30, 60, false, false) a LEFT JOIN ways b ON (a.id2 = b.gid);
```

Note: The last record of this JOIN doesn't contain a geometry value since the shortest path function returns `-1` for the last record to indicate the end of the route.

8.2 Visualize the result

Instead of looking at rows, columns and numbers on the terminal screen it's more interesting to visualize the route on a map. Here a few ways to do so:

- **Store the result as table** with `CREATE TABLE <table name> AS SELECT ...` and show the result in QGIS, for example:

```
CREATE TABLE route AS SELECT seq, id1 AS node, id2 AS edge, cost, b.the_geom FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                30, 60, false, false) a LEFT JOIN ways b ON (a.id2 = b.gid);
```

- **Use QGIS SQL where clause when adding a PostGIS layer:**
  - Create a database connection and add the "ways" table as a background layer.
  - Add another layer of the "ways" table but select `Build query` before adding it.
  - Then type the following into the **SQL where clause** field:

```
"gid" IN ( SELECT id2 AS gid FROM pgr_dijkstra('
                SELECT gid AS id,
                         source::integer,
                         target::integer,
                         length::double precision AS cost
                        FROM ways',
                30, 60, false, false) a LEFT JOIN ways b ON (a.id2 = b.gid)
        )
```

- Use the **QGIS DB Manager** Plugin.
- See the next chapter how to configure a WMS server with Geoserver.

8.3 Simplified input parameters and geometry output

The following function simplifies (and sets default values) when it calls the shortest path Dijkstra function.

Note: The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different argument types can share a name (this is called overloading).

Dijkstra Wrapper

```
--DROP FUNCTION pgr_dijkstra(varchar,int,int);

CREATE OR REPLACE FUNCTION pgr_dijkstra(
                IN tbl varchar,
                IN source integer,
                IN target integer,
                OUT seq integer,
                OUT gid integer,
                OUT geom geometry
        )
        RETURNS SETOF record AS
$BODY$
DECLARE
```

```
        sql     text;
        rec     record;
BEGIN
        seq     := 0;
        sql     := 'SELECT gid,the_geom FROM ' ||
                        'pgr_dijkstra(''SELECT gid as id, source::int, target::int, '
                                        || 'length::float AS cost FROM '
                                        || quote_ident(tbl) || ''', '
                                        || quote_literal(source) || ', '
                                        || quote_literal(target) || ' , false, false), '
                                || quote_ident(tbl) || ' WHERE id2 = gid ORDER BY seq';

        FOR rec IN EXECUTE sql
        LOOP
                seq     := seq + 1;
                gid     := rec.gid;
                geom    := rec.the_geom;
                RETURN NEXT;
        END LOOP;
        RETURN;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE STRICT;
```

Example query

```
SELECT * FROM pgr_dijkstra('ways',30,60);
```

8.4 Route between lat/lon points and return ordered geometry with heading

The following function takes lat/lon points as input parameters and returns a route that can be displayed in QGIS or WMS services such as Mapserver and Geoserver:

Input parameters

- Table name
- `x1`, `y1` for start point and `x2`, `y2` for end point

Output columns

- Sequence (for example to order the results afterwards)
- Gid (for example to link the result back to the original table)
- Street name
- Heading in degree (simplified as it calculates the Azimuth between start and end node of a link)
- Costs as length in kilometer
- The road link geometry

What the function does internally:

1. Finds the nearest nodes to start and end point coordinates
2. Runs shortest path Dijkstra query

3. Flips the geometry if necessary, that target node of the previous road link is the source of the following road link
4. Calculates the azimuth from start to end node of each road link
5. Returns the result as a set of records

```
--DROP FUNCTION pgr_fromAtoB(varchar, double precision, double precision,
--                           double precision, double precision);

CREATE OR REPLACE FUNCTION pgr_fromAtoB(
                IN tbl varchar,
                IN x1 double precision,
                IN y1 double precision,
                IN x2 double precision,
                IN y2 double precision,
                OUT seq integer,
                OUT gid integer,
                OUT name text,
                OUT heading double precision,
                OUT cost double precision,
                OUT geom geometry
        )
        RETURNS SETOF record AS
$BODY$
DECLARE
        sql     text;
        rec     record;
        source  integer;
        target  integer;
        point   integer;

BEGIN
        -- Find nearest node
        EXECUTE 'SELECT id::integer FROM vertices_tmp
                        ORDER BY the_geom <-> ST_GeometryFromText(''POINT('
                        || x1 || ' ' || y1 || ')'',4326) LIMIT 1' INTO rec;
        source := rec.id;

        EXECUTE 'SELECT id::integer FROM vertices_tmp
                        ORDER BY the_geom <-> ST_GeometryFromText(''POINT('
                        || x2 || ' ' || y2 || ')'',4326) LIMIT 1' INTO rec;
        target := rec.id;

        -- Shortest path query (TODO: limit extent by BBOX)
        seq := 0;
        sql := 'SELECT gid, the_geom, name, cost, source, target,
                                ST_Reverse(the_geom) AS flip_geom FROM ' ||
                        'pgr_dijkstra(''SELECT gid as id, source::int, target::int, '
                                        || 'length::float AS cost FROM '
                                        || quote_ident(tbl) || ''', '
                                        || source || ', ' || target
                                        || ' , false, false), '
                                || quote_ident(tbl) || ' WHERE id2 = gid ORDER BY seq';

        -- Remember start point
        point := source;

        FOR rec IN EXECUTE sql
        LOOP
                -- Flip geometry (if required)
                IF ( point != rec.source ) THEN
                        rec.the_geom := rec.flip_geom;
                        point := rec.source;
```

```
                ELSE
                        point := rec.target;
                END IF;

                -- Calculate heading (simplified)
                EXECUTE 'SELECT degrees( ST_Azimuth(
                                ST_StartPoint(''' || rec.the_geom::text || '''),
                                ST_EndPoint(''' || rec.the_geom::text || ''') ) )'
                        INTO heading;

                -- Return record
                seq     := seq + 1;
                gid     := rec.gid;
                name    := rec.name;
                cost    := rec.cost;
                geom    := rec.the_geom;
                RETURN NEXT;
        END LOOP;
        RETURN;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE STRICT;
```

What the function does not do:

- It does not restrict the selected road network by BBOX (necessary for large networks)
- It does not return road classes and several other attributes
- It does not take into account one-way streets
- There is no error handling

Example query

```
SELECT * FROM pgr_fromAtoB('ways',-1.18600,52.96701,-1.11762,52.93691);
```

To store the query result as a table run

```
CREATE TABLE temp_route AS
        SELECT * FROM pgr_fromAtoB('ways',-1.18600,52.96701,-1.11762,52.93691);
--DROP TABLE temp_route;
```

We can now install this function into the database:

```
psql -U postgres -d pgrouting-workshop ~/Desktop/pgrouting-workshop/data/fromAtoB.sql
```

CHAPTER 9

WMS server with GeoServer

Now that we have a pl/pgsql wrapper, we will make it available as a WMS layer using GeoServer.

The installation of GeoServer is out of scope of this workshop but if you're using the OSGeo LiveDVD the server is already installed and only needs to be started.

9.1 Connect to the administration page

In order to create the WMS layer, we need to connect to the administration page of GeoServer. On the OSGeo LiveDVD, open the *Applications* menu on the desktop and then *Geoservers* > *GeoServer* > *Start GeoServer*.

Once the server is up, the administration page should open, click the *Login* button and enter the admin credentials:

- Username: `admin`
- Password: `geoserver`

9.2 Create the layer

Now that we are logged in as an administrator we can start create our layer. In GeoServer terminology we will create an SQL View (see the official documentation for more info).

The first thing to do is to create a new **Workspace** for pgrouting: in the left menu of the page, inside the *Data* section, click *Workspaces* and then *Add new workspace*.

Fill the form with:

- Name: `pgrouting`
- Namespace URI: `http://pgrouting.org/`

And press the submit button.

Next step: create a new **Store** linked to the workspace. Still in the left menu, click *Stores* and then *Add new Store*. Here we can choose the type of data source to configure. Choose *PostGIS*.

Fill the form with:

- Basic Store Info:
- Workspace (select): `pgrouting`
- Data Source Name: `pgrouting`
- Connection Parameters:

- host: `localhost`
- port: `5432`
- database: `pgrouting-workshop`
- schema: `public`
- user: `postgres`
- password: blank

The rest of the values can be left untouched.

Finally, our next task is to create the **Layer**. Click the *Layers* menu and then *Add a new resource*. Select the newly created workspace and store pair: `pgrouting:pgrouting`.

Inside the text, click *Configure new SQL view*.

Name the view `pgrouting` and fill the *SQL statement* with:

```
SELECT ST_MakeLine(route.geom) FROM (
    SELECT geom FROM pgr_fromAtoB('ways', %x1%, %y1%, %x2%, %y2%
  ) ORDER BY seq) AS route
```

In the *SQL view parameters*, click *Guess parameters from SQL*; the list displayed represents the parameters from the SQL above.

For each parameter:

- Set the default value to: *0*
- Change the validation regular expression to `^-?[\d.]+$`

The regular expression will only match numbers.

In the *Attributes* list:

- Hit *Refresh*, one attribute should appear (called *st_makeline*)
- Change the type to `LineString` and the SRID of the geometry column from `-1` to `4326`

Save the form.

Finally, we need to setup the rest of the layer.

The only thing to do in this screen is to make sure that the coordinate reference system is correct: the geometries in the database are in *EPSG:4326* but we want to display them in *EPSG:3857* because the OpenLayers map where the layer will be dispayed is in this projection.

Scoll down to the *coordinate reference system* section and change the **Declared SRS** to `EPSG:3857` and the **SRS handling** to `Reproject native to declared`.

Click the *Compute from data* and *Compute from native bounds* link to automatically set the layer bounds. Click the *Save* at the bottom of the page to create the layer.

CHAPTER 10

OpenLayers 3 Browser Client

The goal of this chapter is to create a simple web based user interface to pgRouting based on OpenLayers 3. The user will be able to choose the start and destination location of the routing by clicking on the map.

The general workflow of this application is to wait until we have the start and destination points, then we send these values to the WMS server who will query the database for a routing result. The result is represented as an image by the WMS server and returned to our application and displayed.

10.1 OpenLayers 3 introduction

OpenLayers 3 is a complete rewrite of OpenLayers 2, it uses modern javascript and HTML5 technologies such as Canvas and WebGL.

The new code is based on the Google Closure Tools, this allows us to use a comprehensive and well-tested library (the Closure Library, also used to build Gmail, Google Maps and most of the Google web applications). But the most powerful tool is the Closure Compiler; a Java based compiler who can remove dead code, optimize and minimize Javascript. These tools are completely optional for the OpenLayers library users: they only need to download the compiled code and use it, that's what we will do now.

Let's explore some key concepts of OpenLayers 3:

At the heart of the library we have the map (`ol.Map` class), responsible for managing the layers, the controls, the view and the renderer.

Each map has a renderer who is responsible to draw the layers into the HTML element. They are three different type of renderer:

- `ol.renderer.dom` a DOM based renderer who uses a grid of html img tag. This type of system is also used in OpenLayers 2 or Leaflet. This is the slowest and least tested of the renderer, don't use it ...
- `ol.renderer.canvas` a Canvas based renderer, uses a single canvas tag and combine all the tiles from the layers into it. This system is also used by the mobile version of HERE from Nokia (http://m.here.com).
- `ol.renderer.webgl` same as the canvas renderer but uses WebGL. WebGL is also used by the new version of Google Maps. At the moment only the 2d navigation is supported.

The view (`ol.View` class) represents what's displayed in the map: this geographic center of the map, the resolution but also the map rotation. Unlike others library, these values are separated from the map object; one advantage is to allows two maps to share the same view (for example in this example)

10.2 Creating a minimal map

Let's start our first OpenLayers 3 map: open a text editor and copy this code into a file named `ol3.html`. You can save this file into the Desktop and open it with a web browser.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>ol3 pgRouting client</title>
5          <meta charset="utf-8">
6          <link href="ol3/ol.css" rel="stylesheet">
7          <style>
8              #ol-map {
9                  width: 100%;
10                 height: 400px;
11             }
12         </style>
13         <script src="ol3/ol.js"></script>
14     </head>
15     <body>
16         <div id="ol-map"></div>
17
18         <script type="text/javascript">
19
20             var map = new ol.Map({
21                 target: 'ol-map',
22                 layers: [
23                     new ol.layer.Tile({
24                         source: new ol.source.OSM()
25                     })
26                 ],
27                 view: new ol.View2D({
28                     center: [-127998, 6974591],
29                     zoom: 10
30                 })
31             });
32
33         </script>
34     </body>
35 </html>
```

This code displays a map with an OpenStreetMap layer centered to a predefined location.

At the moment there's not routing related code; only standard navigation.

Line by line we have:

- line 6: include the default OpenLayers CSS file.
- line 8 to 11: give the map a size: 400px height and the entire page width.
- line 13: include the OpenLayers code. All the functions and javascript classes starting with `ol` comes from there.
- line 16: create a div with a `ol-map` identifier. The map will be displayed inside this div.

The rest of the file (inside the `script` tag) will contain our javascript code to query the server for a routing and display the result.

Once the page is open in a web browser, try to open the javascript console and interact with the `map` object:

```
map.getView().getCenter();
map.getView().setCenter([-29686, 6700403]);

map.getView().setRotation(Math.PI);
```

Note: If you inspect an OpenLayers object using the console, you can see that most of the properties and functions have a short (and cryptic) name; that's because the Google Closure Compiler renames the original names. The goal of this compilation is to product the smaller library as possible.

10.3 WMS GET parameters

Add this code a the end of the `script` tag:

```
var params = {
  LAYERS: 'pgrouting:pgrouting',
  FORMAT: 'image/png'
};
```

The `params` object holds the WMS GET parameters that will be sent to GeoServer. Here we set the values that will never change: the layer name and the output format.

10.4 Select the start and final destination

We want to allow the users to set the start and destination position by clicking on the map.

Add this code a the end of the `script` tag:

```
var startPoint = new ol.Feature();
var finalPoint = new ol.Feature();
new ol.FeatureOverlay({features: [startPoint, finalPoint], map: map});
```

It's creates a feature overlay with two points, which get a reference to the map because they need to update their position according to the view (in short: they move when the map moves).

Add this code a the end of the `script` tag:

```
var transform = ol.proj.getTransform('EPSG:3857', 'EPSG:4326');

map.on('click', function(event) {

  if (startPoint.getGeometry() == null) {
    // first click
    startPoint.setGeometry(new ol.geom.Point(event.coordinate));
  }

  else if (finalPoint.getGeometry() == null) {
    // second click
    finalPoint.setGeometry(new ol.geom.Point(event.coordinate));

    // transform the coordinates from the map projection (EPSG:3857)
    // into the server projection (EPSG:4326)
    var startCoord = transform(startPoint.getGeometry().getCoordinates());
    var finalCoord = transform(finalPoint.getGeometry().getCoordinates());
    var viewparams = [
      'x1:' + startCoord[0], 'y1:' + startCoord[1],
      'x2:' + finalCoord[0], 'y2:' + finalCoord[1]
    ];
    params.viewparams = viewparams.join(';');

    // we now have the two points, create the result layer and add it to the map
    result = new ol.layer.Image({
      source: new ol.source.ImageWMS({
        url: 'http://localhost:8082/geoserver/pgrouting/wms',
```

```
        params: params
      })
    });
    map.addLayer(result);
  }
});
```

When the map is clicked, this function is executed. The geographical position of the cursor is stored into the feature objects and shows them on the map

Once we have the start and destination points (after two clicks); the two pairs of coordinates are transformed from the map projection (`EPSG:3857`) into the server projection (`EPSG:4326`) using the `transform` function.

The `viewparams` property is set on WMS GET parameters object. The value of this property has a special meaning: GeoServer will substitute the value before executing the SQL query for the layer. For example, if we have:

```
SELECT * FROM ways WHERE maxspeed_forward BETWEEN %min% AND %max%
```

And `viewparams` is `viewparams=min:20;max:120` then the query sent to PostGIS will be:

```
SELECT * FROM ways WHERE maxspeed_forward BETWEEN 20 AND 120
```

Finally, a new OpenLayers WMS layer is created and added to the map, the param object is passed to it.

10.5 Clear the results

Add this code between the map's html `div` and the `script` tag:

```
<button id="clear">clear</button>
```

This create a button to allow the user to clear the routing points and start a new routing query.

Add this code a the end of the `script` tag:

```
document.getElementById('clear').addEventListener('click', function(event) {
  // hide the overlays
  startPoint.setGeometry(null);
  finalPoint.setGeometry(null);

  // remove the result layer
  map.removeLayer(result);
});
```

When the button is clicked, this function is executed. The routing points and the result layer are hidden.