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

Release 1

Daniel Kastl, Frédéric Junod

September 18, 2010

CONTENTS

1	Introduction	1
2	About	3
2.1	pgRouting	3
2.2	OpenStreetMap	4
2.3	osm2pgrouting	4
2.4	GeoExt	5
3	Installation and Requirements	7
3.1	Software	7
3.2	Data	8
3.3	Workshop	8
4	osm2pgrouting Import Tool	9
4.1	Create routing database	10
4.2	Run osm2pgrouting	11
5	Create a Network Topology	13
5.1	Load network data	13
5.2	Calculate topology	14
5.3	Add indices	15
6	Shortest Path Search	17
6.1	Dijkstra	17
6.2	A-Star	19
6.3	Shooting-Star	20
7	Advanced Routing Queries	23
7.1	Weighted costs	23
7.2	Restricted access	25
8	Server side script with PHP	27
8.1	Closest edge	27
8.2	Routing query	28
8.3	GeoJSON output	29
9	GeoExt Browser Client	31
9.1	Routing method selection	32
9.2	Select the start and final destination	32
9.3	Call and receive data from web service	34
9.4	Trigger the web service call	35
9.5	What's next ?	36
9.6	Full source code	37

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 authors of the previous [pgRouting](#) workshops, that have been held at FOSS4G events in Canada and South Africa and at local conferences in Japan.

License

This work is licensed under a [Creative Commons Attribution-Share Alike 3.0 License](https://creativecommons.org/licenses/by-sa/3.0/).



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

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 Dijkstra: 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 Known Text (WKT) and Well Known Binary (WKB).
- 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>

INSTALLATION AND REQUIREMENTS

For this workshop you need:

- A webserver like Apache with PHP support (and PHP PostgreSQL module)
- Preferable a Linux operating system like Ubuntu
- An editor like Gedit
- 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 latest Ubuntu 10.04.

3.1 Software

Installation of pgRouting on Ubuntu became very easy now because packages are available in a [Launchpad repository](#):

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

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

# Install pgRouting packages
sudo apt-get install gaul-devel \
    postgresql-8.4-pgrouting \
    postgresql-8.4-pgrouting-dd \
    postgresql-8.4-pgrouting-tsp

# Install osm2pgrouting package
sudo apt-get install osm2pgrouting

# Install workshop material (optional)
sudo apt-get install pgrouting-workshop
```

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

Note:

- “Multiverse” packages must be available as software sources. Currently only packages for Ubuntu 10.04 have been built, but further packages are likely to come if there is demand for them.
- 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/8.4/main/pg_hba.conf` and restart PostgreSQL server with `sudo service postgresql-8.4 restart`.

3.2 Data

The pgRouting workshop will make use of OpenStreetMap data of Barcelona, 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 as file barcelona.osm
wget --progress=dot:mega -O barcelona.osm \
    http://osmapi.hypercube.telascience.org/api/0.6/map \
        ?bbox=1.998653,41.307213,2.343693,41.495207
```

The API has a download size limitation, which can make it a bit inconvenient to download large areas with many features. An alternative is [JOSM Editor](#), which also makes API calls to download data, but it provides an user friendly interface. You can save the data as `.osm` file to use it in this workshop. JOSM is also available on the LiveDVD.

Note:

- OpenStreetMap API v0.6, see for more information http://wiki.openstreetmap.org/index.php/OSM_Protocol_Version_0.6
- Barcelona data is available at the LiveDVD in `/usr/local/share/osm/`

An alternative for very large areas is the download service of [CloudMade](#). The company offers extracts of maps from countries around the world. For data of Spain for example go to <http://download.cloudmade.com/europe/spain> and download the compressed `.osm.bz2` file:

```
wget --progress=dot:mega http://download.cloudmade.com/europe/spain/spain.osm.bz2
```

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

3.3 Workshop

If 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. It contains a compressed file with database dumps as well as a smaller network data of Barcelona downtown. To extract the file run `tar -xzf ~/Desktop/pgrouting-workshop/data.tar.gz`.

OSM2PGROUTING IMPORT TOOL

osm2pgrouting is a command line tool that makes it very 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 currently hosted on the pgRouting project site: <http://pgrouting.postlbs.org/wiki/tools/osm2pgrouting>

Note: There are some limitations though especially regarding network size. The current version of osm2pgrouting needs to load all data into memory, which makes it fast but also requires a lot of 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 Barcelona.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 OpenStreetMap 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/.

4.1 Create routing database

Before we can run osm2pgrouting we have to create PostgreSQL a database and load PostGIS and pgRouting functions into this database. Therefor open a terminal window and execute the following commands:

```
# become user "postgres" (or run as user "postgres")
sudo su postgres

# create routing database
createdb routing
createlang plpgsql routing

# add PostGIS functions
psql -d routing -f /usr/share/postgresql/8.4/contrib/postgis-1.5/postgis.sql
psql -d routing -f /usr/share/postgresql/8.4/contrib/postgis-1.5/spatial_ref_sys.sql

# add pgRouting core functions
```

```
psql -d routing -f /usr/share/postlbs/routing_core.sql
psql -d routing -f /usr/share/postlbs/routing_core_wrappers.sql
psql -d routing -f /usr/share/postlbs/routing_topology.sql
```

An alternative way with **PgAdmin III** and SQL commands. Start PgAdmin III (available on the LiveDVD), connect to any database and open the SQL Editor:

```
-- create routing database
CREATE DATABASE "routing";
```

Then connect to the `routing` database and open a new SQL Editor window:

```
-- add plpgsql and PostGIS/pgRouting functions
CREATE PROCEDURAL LANGUAGE plpgsql;
```

Next open `.sql` files with PostGIS/pgRouting functions as above and load them to the `routing` database.

Note: PostGIS `.sql` files can be on different locations. This depends on your version of PostGIS and PostgreSQL. The example above is valid for PostgreSQL/PostGIS version 1.5 installed on the LiveDVD.

4.2 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 `routing` database we created before. Furthermore we take `~/Desktop/pgrouting-workshop/data/sampledata.osm` as raw data. This file contains only OSM data from downtown Barcelona 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:

```
cd ~/Desktop/pgrouting-workshop/
osm2pgrouting -file "data/sampledata.osm" \
              -conf "/usr/share/osm2pgrouting/mapconfig.xml" \
              -dbname routing \
              -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
-clean		drop peviously created tables	no

Note: If you get permission denied error for postgres users you can set connection method to trust in `/etc/postgresql/8.4/main/pg_hba.conf` and restart PostgreSQL server with `sudo service postgresql-8.4 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:

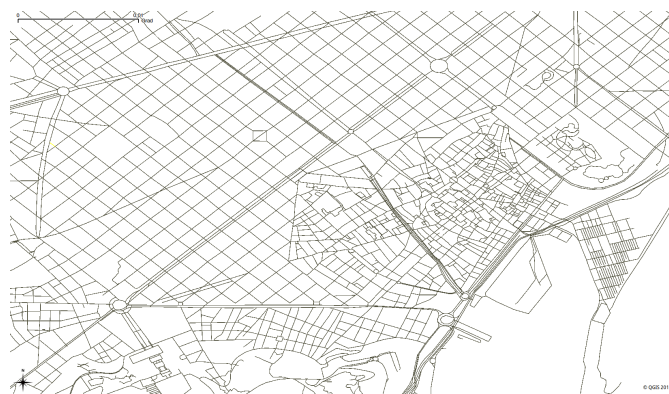
Run: `psql -U postgres -d routing -c "\d"`

If everything went well the result should look like this:

```
          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)
```

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.

5.1 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 smaller network data of Barcelona downtown. 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 described in the previous chapter. It will also load the Barcelona sample data with a minimum number of attributes, which you will usually find in any network data:

```
# Create a database  
createdb -U postgres pgrouting-workshop  
  
# Load database dump file  
psql -U postgres -d pgrouting-workshop \  
-f ~/Desktop/pgrouting-workshop/data/sampled_data_notopo.sql
```

Let's see which 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 | table | postgres
 public | spatial_ref_sys | table | postgres
 public | ways          | table | postgres
(4 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    | integer       | not null
 class_id | integer       |
 length | double precision |
 name   | character(200) |
 the_geom | geometry      |
Indexes:
    "ways_pkey" PRIMARY KEY, btree (gid)
    "geom_idx" gist (the_geom)
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)
```

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.

5.2 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 `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.:

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

-- Run topology function
SELECT assign_vertex_id('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".

5.3 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 look like this:

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

```
          List of relations
 Schema | Name                | Type  | Owner
-----+-----+-----+-----
 public | geography_columns   | view  | postgres
 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
(6 rows)
```

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

```
          Table "public.ways"
 Column | Type          | Modifiers
-----+-----+-----
 gid    | integer       | not null
 class_id | integer       |
 length | double precision |
 name   | character(200) |
 the_geom | geometry      |
```

```
source | integer |  
target | integer |
```

Indexes:

```
"ways_pkey" PRIMARY KEY, btree (gid)  
"geom_idx" gist (the_geom)  
"source_idx" btree (source)  
"target_idx" btree (target)
```

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 our first routing query with Dijkstra algorithm!

SHORTEST PATH SEARCH

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 the three different shortest path algorithms and which attributes are required. If you run `osm2pgrouting` tool to import *OpenStreetMap* data, the `ways` table contains all attributes already to run all shortest path function functions.

6.1 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;
```

Function with parameters

```
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”) ignore “has_reverse_cost” setting

6.1.1 Core

Each algorithm has its *core* function , 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',
    605, 359, false, false);
```

vertex_id	edge_id	cost
605	5575	0.0717467247513547
1679	2095	0.148344716070272
588	2094	0.0611856933258344
...
359	-1	0

(82 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', 605, 359);
```

gid	the_geom
168	MULTILINESTRING((2.1633077 41.3802886,2.1637094 41.3803008))
169	MULTILINESTRING((2.1637094 41.3803008,2.1638796 41.3803093))
170	MULTILINESTRING((2.1638796 41.3803093,2.1640527 41.3803265))
...	...
5575	MULTILINESTRING((2.1436976 41.3897581,2.143876 41.3903893))

(81 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', 605, 359, 0.1);
```

```
gid | the_geom
-----+-----
 168 | MULTILINESTRING((2.1633077 41.3802886,2.1637094 41.3803008))
 169 | MULTILINESTRING((2.1637094 41.3803008,2.1638796 41.3803093))
 170 | MULTILINESTRING((2.1638796 41.3803093,2.1640527 41.3803265))
 ... | ...
5575 | MULTILINESTRING((2.1436976 41.3897581,2.143876 41.3903893))
(81 rows)
```

Note: 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

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.

Prerequisites

For A-Star you need to prepare your network table and add latitude/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(ST_startpoint(the_geom));
UPDATE ways SET y1 = y(ST_startpoint(the_geom));

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

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

UPDATE ways SET x2 = x(ST_PointN(the_geom, ST_NumPoints(the_geom)));
UPDATE ways SET y2 = y(ST_PointN(the_geom, ST_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:

Function with parameters

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”) ignore “has_reverse_cost” setting

6.2.1 Core

```
SELECT * FROM shortest_path_astar('
    SELECT gid as id,
           source::integer,
           target::integer,
           length::double precision as cost,
           x1, y1, x2, y2
    FROM ways',
    605, 359, false, false);
```

```
vertex_id | edge_id |      cost
-----+-----+-----
         605 |      5575 | 0.0717467247513547
        1679 |      2095 | 0.148344716070272
         588 |      2094 | 0.0611856933258344
        ... |      ... | ...
         359 |       -1 | 0
(82 rows)
```

6.2.2 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', 605, 359, 0.1);
```

```
gid | the_geom
-----+-----
 2095 | MULTILINESTRING((2.1456208 41.3901317,2.143876 41.3903893))
 1721 | MULTILINESTRING((2.1494579 41.3890058,2.1482992 41.3898429))
 1719 | MULTILINESTRING((2.1517067 41.3873058,2.1505566 41.3881623))
    ... | ...
 3607 | MULTILINESTRING((2.1795052 41.3843643,2.1796184 41.3844328))
(81 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.
- 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

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.

Prerequisites

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

```
-- Add rule and to_cost column
ALTER TABLE ways ADD COLUMN to_cost double precision;
ALTER TABLE ways ADD COLUMN rule text;
```

Shooting-Star algorithm introduces two new attributes

Attribute	Description
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)

Function with parameters

```
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

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..

6.3.1 Core

An example of a Shooting Star query may look like this:

```
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',
    609, 366, false, false);
```

vertex_id	edge_id	cost
2026	609	0.132151952643718
2461	273	0.132231995120746
2459	272	0.034483403610109
...
2571	366	0.120471497765379

(81 rows)

Warning: Shooting Star 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.

6.3.2 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', 609, 366, 0.1, 'length', true, true);
```

gid	the_geom
609	MULTILINESTRING((2.1436976 41.3897581,2.1449097 41.3889929))
273	MULTILINESTRING((2.1460685 41.3898043,2.1449097 41.3889929))
272	MULTILINESTRING((2.1463431 41.3900361,2.1460685 41.3898043))
...	...
3607	MULTILINESTRING((2.1795052 41.3843643,2.1796184 41.3844328))

(81 rows)

Note: There is currently no wrapper function for Shooting-Star without bounding box, since bounding boxes are very useful to increase performance.

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.

ADVANCED ROUTING QUERIES

As explained in the previous chapter a shortest path query usually 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', 609, 366, true, true);
```

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").

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;`

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

Run: `SELECT * FROM classes;`

```
id | type_id | name | cost  
---+-----+-----+-----  
201 | 2 | lane |  
204 | 2 | opposite |  
203 | 2 | opposite_lane |  
202 | 2 | track |  
117 | 1 | bridleway |
```

```
113 |      1 | bus_guideway      |
118 |      1 | byway             |
115 |      1 | cicleway         |
116 |      1 | footway          |
108 |      1 | living_street    |
101 |      1 | motorway         |
103 |      1 | motorway_junction |
102 |      1 | motorway_link    |
114 |      1 | path             |
111 |      1 | pedestrian       |
106 |      1 | primary         |
107 |      1 | primary_link     |
107 |      1 | residential      |
100 |      1 | road             |
100 |      1 | unclassified     |
106 |      1 | secondary        |
109 |      1 | service          |
112 |      1 | services         |
119 |      1 | steps            |
107 |      1 | tertiary         |
110 |      1 | track            |
104 |      1 | trunk            |
105 |      1 | trunk_link       |
401 |      4 | roundabout       |
301 |      3 | grade1           |
302 |      3 | grade2           |
303 |      3 | grade3           |
304 |      3 | grade4           |
305 |      3 | grade5           |
```

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 * 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', 609, 366, true, true);
```

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 * 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 AND class_id != 111', 609, 366, true, true);
```

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

SERVER SIDE SCRIPT 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 or better GeoJSON and send it back to the web client.

Let's start with some PHP template and then place this file in a directory, which is accessible by Apache:

```
<?php

// Database connection settings
define("PG_DB" , "routing");
define("PG_HOST", "localhost");
define("PG_USER", "postgres");
define("PG_PORT", "5432");
define("TABLE", "ways");

// Retrieve start point
$start = split(' ', $_REQUEST['startpoint']);
$startPoint = array($start[0], $start[1]);

// Retrieve end point
$end = split(' ', $_REQUEST['finalpoint']);
$endPoint = array($end[0], $end[1]);

?>
```

8.1 Closest edge

Usually the start and end point, which we retrieved from the client, is not the start or end vertex of an edge. It is more convenient to look for the closest edge than for the closest vertex, because Shooting Star algorithm is “edge-based”. For “vertex-based” algorithms (Dijkstra, A-Star) we can choose arbitrary start or end of the selected edge.

```
<?php

// Find the nearest edge
$startEdge = findNearestEdge($startPoint);
```

```
$sendEdge = findNearestEdge($endPoint);

// FUNCTION findNearestEdge
function findNearestEdge($lonlat) {

    // Connect to database
    $con = pg_connect("dbname=".PG_DB." host=".PG_HOST." user=".PG_USER);

    $sql = "SELECT gid, source, target, the_geom,
              distance(the_geom, GeometryFromText(
                'POINT(".$lonlat[0]." ".$lonlat[1].")', 4326)) AS dist
            FROM ".TABLE."
            WHERE the_geom && setsrid(
                'BOX3D(".$lonlat[0]-0.1)."
                  ".$lonlat[1]-0.1).",
                  ".$lonlat[0]+0.1)."
                  ".$lonlat[1]+0.1).")'::box3d, 4326)
            ORDER BY dist LIMIT 1";

    $query = pg_query($con, $sql);

    $edge['gid']      = pg_fetch_result($query, 0, 0);
    $edge['source']   = pg_fetch_result($query, 0, 1);
    $edge['target']   = pg_fetch_result($query, 0, 2);
    $edge['the_geom'] = pg_fetch_result($query, 0, 3);

    // Close database connection
    pg_close($con);

    return $edge;
}

?>
```

8.2 Routing query

```
<?php

// Select the routing algorithm
switch($_REQUEST['method']) {

    case 'SPD' : // Shortest Path Dijkstra

        $sql = "SELECT rt.gid, ST_AsGeoJSON(rt.the_geom) AS geojson,
                    length(rt.the_geom) AS length, ".TABLE.".gid
                FROM ".TABLE.",
                    (SELECT gid, the_geom
                     FROM dijkstra_sp_delta(
                        '".TABLE."',
                        ".$startEdge['source'].",
                        ".$endEdge['target'].",
                        0.1)
                     ) as rt
                WHERE ".TABLE.".gid=rt.gid";

        break;

    case 'SPA' : // Shortest Path A*

        $sql = "SELECT rt.gid, ST_AsGeoJSON(rt.the_geom) AS geojson,
```



```
        length(rt.the_geom) AS length, ".TABLE.".gid
FROM ".TABLE.",
    (SELECT gid, the_geom
     FROM astar_sp_delta(
         '".TABLE."',
         ".$startEdge['source'].",
         ".$endEdge['target'].",
         0.1)
     ) as rt
WHERE ".TABLE.".gid=rt.gid;";

break;

case 'SPS' : // Shortest Path Shooting*

    $sql = "SELECT rt.gid, ST_AsGeoJSON(rt.the_geom) AS geojson,
        length(rt.the_geom) AS length, ".TABLE.".gid
    FROM ".TABLE.",
        (SELECT gid, the_geom
         FROM shootingstar_sp(
             '".TABLE."',
             ".$startEdge['gid'].",
             ".$endEdge['gid'].",
             0.1, 'length', true, true)
         ) as rt
    WHERE ".TABLE.".gid=rt.gid;";

    break;

} // close switch

// Connect to database
$dbcon = pg_connect("dbname=".PG_DB." host=".PG_HOST." user=".PG_USER);

// Perform database query
$query = pg_query($dbcon,$sql);

?>
```

8.3 GeoJSON output

OpenLayers allows to draw lines directly using GeoJSON format, so our script returns a GeoJSON FeatureCollection object:

```
<?php

// Return route as GeoJSON
$geojson = array(
    'type' => 'FeatureCollection',
    'features' => array()
);

// Add edges to GeoJSON array
while($edge=pg_fetch_assoc($query)) {

    $feature = array(
        'type' => 'Feature',
        'geometry' => json_decode($edge['geojson'], true),
        'crs' => array(
            'type' => 'EPSG',
            'properties' => array('code' => '4326')
        ),
    ),
```

```
        'properties' => array(
            'id' => $edge['id'],
            'length' => $edge['length']
        )
    );

    // Add feature array to feature collection array
    array_push($geojson['features'], $feature);
}

// Close database connection
pg_close($dbcon);

// Return routing result
header('Content-type: application/json', true);
echo json_encode($geojson);

?>
```

GEOEXT BROWSER CLIENT

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.

Let's start with a simple GeoExt example and extend it with routing functionality then:

```
<html>
<head>

<title>A Basic GeoExt Page</title>
<script src="Ext-3.2.1/adapter/ext/ext-base.js" type="text/javascript"></script>
<script src="Ext-3.2.1/ext-all.js" type="text/javascript"></script>
<link rel="stylesheet" type="text/css" href="Ext-3.2.1/resources/css/ext-all.css" />
<script src="OpenLayers-2.9.1/lib/OpenLayers.js" type="text/javascript"></script>
<script src="GeoExt-0.7/lib/GeoExt.js" type="text/javascript"></script>
<link rel="stylesheet" type="text/css"
      href="GeoExt-0.7/resources/css/geoext-all-debug.css" />

<script type="text/javascript">
    Ext.onReady(function() {
        var panel = new GeoExt.MapPanel({
            renderTo: 'gxmap',
            map: {
                layers: [new OpenLayers.Layer.OSM("Simple OSM Map")]
            },
            center: [245300, 5070600],
            zoom: 11,
            height: 400,
            width: 600,
            title: 'A Simple GeoExt Map'
        });
    });
</script>
</head>
<body>
<div id="gxmap"></div>
</body>
</html>
```

In the header we include all the javascript and css needed for the application, we also define a function to be run when the page is loaded (`Ext.onReady`).

This function creates a `GeoExt.MapPanel` with an `OpenStreetMap` layer centered to Barcelona. In this code, no `OpenLayers.Map` is explicitly created; the `GeoExt.MapPanel` do this under the hood: it takes the map options, the center and the zoom and create a map instance accordingly.

To allow our users to get directions, we need to provide:

- a way to select the routing algorithm (Shortest path Dijkstra, A* or Shooting*),

- a way to select the start and final destination.

Note: This chapter only show code snippets, the full source code of the page can be found in `pgrouting-workshop/web/routing-final.html` that should be on your desktop. The full listing can also be found at the end of this chapter.

9.1 Routing method selection

To select the routing method, we will use an `Ext.form.ComboBox`: it behaves just like a normal html select but we can more easily control it.

Just like the `GeoExt.MapPanel`, we need an html element to place our control, let's create a new div in the body (with 'method' as id):

```
<body>
  <div id="gxmap"></div>
  <div id="method"></div>
</body>
```

Then we create the combo itself:

```
var method = new Ext.form.ComboBox({
  renderTo: "method",
  triggerAction: "all",
  editable: false,
  forceSelection: true,
  store: [
    ["SPD", "Shortest Path Dijkstra"],
    ["SPA", "Shortest Path A*"],
    ["SPS", "Shortest Path Shooting*"]
  ]
});
```

In the `store` option, we set all the possible values for the routing method; the format is an array of options where an option is in the form `[key, name]`. The key will be send to the server (the php script in our case) and the value displayed in the combo.

The `renderTo` specify where the combo must be rendered, we use our new div here.

And finally, a default value is selected:

```
method.setValue("SPD");
```

This part only use ExtJS component: no OpenLayers or GeoExt code here.

9.2 Select the start and final destination

We want to allow the users to draw and move the start and final destination points. This is more or less the behavior of google maps and others: the user selects the points via a search box (address search) or by clicking the map. The system query the server and display the route on the map. The user can later move the start or final point and the route is updated.

In this workshop, we will only implement the input via the map (draw points and drag-and-drop) but it's possible to implement the search box feature by using a web service like [GeoNames](#) or any other [geocoding](#) service.

To do this we will need a tool to draw points (we will use the `OpenLayers.Control.DrawFeatures` control) and a tool to move points (`OpenLayers.Control.DragFeatures` will be perfect for this job). As their name suggests these controls comes from OpenLayers.

These two controls will need a place to draw and manipulate the points; we will also need an `OpenLayers.Layer.Vector` layer. In OpenLayers, a vector layer is a place where features (a geometry and attributes) can be drawn programmatically (in contrast, the OSM layer is a raster layer).

Because vector layers are cheap, we will use a second one to draw the route returned by the web service. The layers initialization is:

```
// create the layer where the route will be drawn
var route_layer = new OpenLayers.Layer.Vector("route", {
  styleMap: new OpenLayers.StyleMap(new OpenLayers.Style({
    strokeColor: "#ff9933",
    strokeWidth: 3
  })))
});
```

"route" is the layer name, any string can be used. `styleMap` gives the layer a bit of visual style with a custom stroke color and width (in pixel).

The second layer initialization is simply:

```
// create the layer where the start and final points will be drawn
var points_layer = new OpenLayers.Layer.Vector("points");
```

The two layers are added to the `OpenLayers.Map` object with:

```
// add the layers to the map
map.addLayers([points_layer, route_layer]);
```

Let's look at the control to draw the points: because this component has special behavior it's more easy to create a new class based on the standard `OpenLayers.Control.DrawFeatures` control. This new control (named `DrawPoints`) is saved in a separated javascript file (`web/DrawPoints.js`):

```
DrawPoints = OpenLayers.Class(OpenLayers.Control.DrawFeature, {

  // this control is active by default
  autoActivate: true,

  initialize: function(layer, options) {
    // only point can be drawn
    var handler = OpenLayers.Handler.Point;
    OpenLayers.Control.DrawFeature.prototype.initialize.apply(
      this, [layer, handler, options]
    );
  },

  drawFeature: function(geometry) {
    OpenLayers.Control.DrawFeature.prototype.drawFeature.apply(
      this, arguments
    );
    if (this.layer.features.length == 1) {
      // we just draw the startpoint
      // note: if we want to apply a special style to the
      // start point we should do this here
    } else if (this.layer.features.length == 2) {
      // we just draw the finalpoint
      // note: if we want to apply a special style to the
      // final point we should do this here

      // we have all what we need; we can deactivate ourself.
      this.deactivate();
    }
  }
});
```

```
    }  
  });
```

In the `initialize` function (that's the class constructor) we set that this control can only draw points (handler variable is `OpenLayers.Handler.Point`).

The special behavior is implemented in the `drawFeature` function: because we only need the start and final points the control deactivates itself when two points are drawn by counting how many features has the vector layer. Control deactivation is `this.deactivate()`.

Our control is then created with:

```
// create the control to draw the points (see the DrawPoints.js file)  
var draw_points = new DrawPoints(points_layer);
```

`points_layer` is the vector layer created earlier.

And now for the `DragFeature` control:

```
// create the control to move the points  
var drag_points = new OpenLayers.Control.DragFeature(points_layer, {  
  autoActivate: true  
});
```

Again, `points_layer` is the vector layer, `autoActivate: true` tells `OpenLayers` that we want this control to be automatically activated.

```
// add the controls to the map  
map.addControls([draw_points, drag_points]);
```

Adds the controls to the map.

9.3 Call and receive data from web service

The basic workflow to get a route from the web server is:

1. transform our points coordinates from EPSG:900913 to EPSG:4326
2. call the web service with the correct arguments (method name and two points coordinates)
3. parse the web service response: transform GeoJSON to `OpenLayers.Feature.Vector`
4. transform all the coordinates from EPSG:4326 to EPSG:900913
5. add this result to a vector layer

The first item is something new: our map uses the EPSG:900913 projection (because we use an OSM layer) but the web service expects coordinates in EPSG:4326: we have to re-project the data before sending them. This is not a big deal: we will simply use the [Proj4js](#) javascript library.

(The second item *call the web service* is covered in the next chapter.)

The routing web service in `pgrouting.php` returns a `GeoJSON FeatureCollection` object. A `FeatureCollection` is simply an array of features: one feature for each route segment. This is very convenient because `OpenLayers` and `GeoExt` have all what we need to handle this format. To make our live even easier, we are going to use the `GeoExt.data.FeatureStore`:

```
var store = new GeoExt.data.FeatureStore({  
  layer: route_layer,  
  fields: [  
    {name: "length"}  
  ],  
});
```

```
proxy: new GeoExt.data.ProtocolProxy({
  protocol: new OpenLayers.Protocol.HTTP({
    url: './php/pgrouting.php',
    format: new OpenLayers.Format.GeoJSON({
      internalProjection: epsg_900913,
      externalProjection: epsg_4326
    })
  })
});
```

A store is simply a container to store informations: we can push data into and get it back.

Let's explain all the options:

`layer`: the argument is a vector layer: by specifying a layer, the FeatureStore will automatically draw the data received into this layer. This is exactly what we need for the last item (*add this result to a vector layer*) in the list above.

`fields`: lists all the attributes returned along with the geometry: `pgrouting.php` returns the segment length so we set it here. Note that this information is not used in this workshop.

`proxy`: the proxy item specify where the data should be taken: in our case from a HTTP server. The proxy type is `GeoExt.data.ProtocolProxy`: this class connects the ExtJS world (the store) and the OpenLayers world (the protocol object).

`protocol`: this OpenLayers component is able to make HTTP requests to an `url` (our php script) and to parse the response (`format` option). By adding the `internalProjection` and `externalProjection` option, the coordinates re-projection is made by the format.

We now have all what we need to handle the data returned by the web service: the next chapter will explain how and when to call the service.

9.4 Trigger the web service call

We need to call the web service when:

- the two points are drawn
- one of the point is moved
- the routing method has changed

Our vector layer generates an event (called `featureadded`) when a new feature is added, we can listen to this event and call to `pgrouting` function (this function will be presented shortly):

```
draw_layer.events.on({
  featureadded: function() {
    pgrouting(store, draw_layer, method.getValue());
  }
});
```

Note: Before we continue some words about events: an event in OpenLayers (the same apply for ExtJS and other frameworks), is a system to allow a function to be called when *something* happened. For instance when a layer is added to the map or when the mouse is over a feature. Multiple functions can be connected to the same event.

No event is generated when a point is moved, hopefully we can give a function to the DragFeature control to be called we the point is moved:

```
drag_points.onComplete = function() {
  pgrouting(store, draw_layer, method.getValue());
};
```

For the *method* combo, we can add a listeners options to the constructor with a *select* argument (that's the event triggered when the user changes the value):

```
var method = new Ext.form.ComboBox({
    renderTo: "method",
    triggerAction: "all",
    editable: false,
    forceSelection: true,
    store: [
        ["SPD", "Shortest Path Dijkstra"],
        ["SPA", "Shortest Path A*"],
        ["SPS", "Shortest Path Shooting*"]
    ],
    listeners: {
        select: function() {
            pgrouting(store, draw_layer, method.getValue());
        }
    }
});
```

It's now time to present the pgrouting function:

```
// global projection objects (uses the proj4js lib)
var epsg_4326 = new OpenLayers.Projection("EPSG:4326"),
    epsg_900913 = new OpenLayers.Projection("EPSG:900913");

function pgrouting(store, layer, method) {
    if (layer.features.length == 2) {
        // erase the previous route
        store.removeAll();

        // transform the two geometries from EPSG:900913 to EPSG:4326
        var startpoint = layer.features[0].geometry.clone();
        startpoint.transform(epsg_900913, epsg_4326);
        var finalpoint = layer.features[1].geometry.clone();
        finalpoint.transform(epsg_900913, epsg_4326);

        // load to route
        store.load({
            params: {
                startpoint: startpoint.x + " " + startpoint.y,
                finalpoint: finalpoint.x + " " + finalpoint.y,
                method: method
            }
        });
    }
}
```

The pgrouting function calls the web service through the store argument.

At first, the function checks if two points are passed in argument. Then, `store.removeAll()` is called to erase a previous result from the layer (remember that the store and the vector layer are binded). The two points coordinates are then projected using `OpenLayers.Projection` instances.

Finally, `store.load()` is called with a `params` representing the `pgrouting.php` arguments (they are passed to the HTTP GET call).

9.5 What's next ?

Possible enhancements:

- use a geocoding service to get start/final point

- way point support
- nice icons for the start and final points
- driving directions (road map): we already have the segment length

9.6 Full source code

```
<html>
<head>

<title>A Basic GeoExt Page</title>
<script src="Ext-3.2.1/adaptor/ext/ext-base.js" type="text/javascript"></script>
<script src="Ext-3.2.1/ext-all.js" type="text/javascript"></script>
<link rel="stylesheet" type="text/css" href="Ext-3.2.1/resources/css/ext-all.css" />
<script src="OpenLayers-2.9.1/lib/OpenLayers.js" type="text/javascript"></script>
<script src="GeoExt-0.7/lib/GeoExt.js" type="text/javascript"></script>
<link rel="stylesheet" type="text/css"
      href="GeoExt-0.7/resources/css/geoext-all-debug.css" />

<script src="DrawPoints.js" type="text/javascript"></script>

<script src="proj4js-1.0.1/lib/proj4js.js" type="text/javascript"></script>

<script type="text/javascript">

    // global projection objects (uses the proj4js lib)
    var epsg_4326 = new OpenLayers.Projection("EPSG:4326"),
        epsg_900913 = new OpenLayers.Projection("EPSG:900913");

    function pgrouting(store, layer, method) {
        if (layer.features.length == 2) {
            // erase the previous route
            store.removeAll();

            // transform the two geometries from EPSG:900913 to EPSG:4326
            var startpoint = layer.features[0].geometry.clone();
            startpoint.transform(epsg_900913, epsg_4326);
            var finalpoint = layer.features[1].geometry.clone();
            finalpoint.transform(epsg_900913, epsg_4326);

            // load to route
            store.load({
                params: {
                    startpoint: startpoint.x + " " + startpoint.y,
                    finalpoint: finalpoint.x + " " + finalpoint.y,
                    method: method
                }
            });
        }
    }

    Ext.onReady(function() {
        // create the map panel
        var panel = new GeoExt.MapPanel({
            renderTo: "gxmap",
            map: {
                layers: [new OpenLayers.Layer.OSM("Simple OSM Map")]
            },
            center: [245300, 5070600],
            zoom: 11,
        });
    });
};
```

```
        height: 400,
        width: 600,
        title: "A Simple GeoExt Map"
    });
    var map = panel.map;

    // create the layer where the route will be drawn
    var route_layer = new OpenLayers.Layer.Vector("route", {
        styleMap: new OpenLayers.StyleMap(new OpenLayers.Style({
            strokeColor: "#ff9933",
            strokeWidth: 3
        })))
    });

    // create the layer where the start and final points will be drawn
    var points_layer = new OpenLayers.Layer.Vector("points");

    // when a new point is added to the layer, call the pgrouting function
    points_layer.events.on({
        featureadded: function() {
            pgrouting(store, points_layer, method.getValue());
        }
    });

    // add the layers to the map
    map.addLayers([points_layer, route_layer]);

    // create the control to draw the points (see the DrawPoints.js file)
    var draw_points = new DrawPoints(points_layer);

    // create the control to move the points
    var drag_points = new OpenLayers.Control.DragFeature(points_layer, {
        autoActivate: true
    });

    // when a point is moved, call the pgrouting function
    drag_points.onComplete = function() {
        pgrouting(store, points_layer, method.getValue());
    };

    // add the controls to the map
    map.addControls([draw_points, drag_points]);

    // create the store to query the web service
    var store = new GeoExt.data.FeatureStore({
        layer: route_layer,
        fields: [
            {name: "length"}
        ],
        proxy: new GeoExt.data.ProtocolProxy({
            protocol: new OpenLayers.Protocol.HTTP({
                url: "./php/pgrouting.php",
                format: new OpenLayers.Format.GeoJSON({
                    internalProjection: epsg_900913,
                    externalProjection: epsg_4326
                })
            })
        })
    });

    // create the method combo box
    var method = new Ext.form.ComboBox({
```

```
renderTo: "method",
triggerAction: "all",
editable: false,
forceSelection: true,
store: [
    ["SPD", "Shortest Path Dijkstra"],
    ["SPA", "Shortest Path A*"],
    ["SPS", "Shortest Path Shooting*"]
],
listeners: {
    select: function() {
        pgrouting(store, points_layer, method.getValue());
    }
}
});
// default method is Shortest Path Dijkstra
method.setValue("SPD");
});
</script>
</head>
<body>
<div id="gxmap"></div>
<div id="method"></div>
</body>
</html>
```