Hidden markov model based map-matching

Supervisor Amnir Hadachi

Joosep Kibal
Institute of Computer science
university of Tartu
truesigh@ut.ee

Abstract – Map matching is the process of aligning a sequence of observed user positions with the road network on a digital map. It is used in many applications such as moving object management, traffic flow analysis and driving directions. A number of map-matching algorithms have been developed by researchers around the world using different techniques such as topological analysis of spatial road network data, probabilistic theory, Kalman filter, fuzzy logic, and belief theory. In this project I started implementing the Hidden Markov model based map-matching algorithm described in[1]. The aim of the project was to try to create a map-matching solution suitable for real-time applications.

1. INTRODUCTION

In Intelligent Transportation Systems (ITS) "Floating car" or "probe" data collection is a set of relatively low-cost methods for obtaining travel time and speed data for vehicles traveling along streets, highways, motorways (freeways), and other transport routes there are manly three different methods used to gather raw data [2]:

- Triangulation method. In developed countries high proportion of cars contain mobile phones. The phones transmit their presence to mobile phone networks. As the car moves so does signal of the phone and by using triangulation, pattern matching or cell-sector statistics the data is converted to traffic flow information.
- Vehicle re-identification. Vehicle re-identification methods require sets of detectors mounted along the road. A unique serial number for a device in the vehicle is detected at one location and then detected again (re-identified) further down the road. Travel times and speed are calculated by comparing the time at which a specific device is detected by pairs of sensors.
- GPS based methods. Vehicles are equipped with GPS systems that have two-way communication with a traffic data provider. Positioning readings are used to calculate vehicle speeds. Modern solutions may use GPS equipped smart-phones.

The main advantages of Floating Car Data (FCD) are that it is less expensive than sensors or cameras, it has more coverage, is faster to set up and works well in all weather conditions.[2]

The map-matching is the procedure of comparing the vehicle tracking data and the digital road map, with the purpose of matching the vehicular positions to the road on which the vehicle actually had driven. When using FDC the GPS data is not precise so it is possible that one GPS location can be matched to several road segments. Also there can be a sampling error caused by the sampling rate [1]. The hidden Markov model based map-matching algorithm should provide accurate results quickly so that it is suitable for real time applications

2. RELATED WORKS

Several map-matching algorithms are surveyed by Quddus in [3]. Also a great table for some of the different algorithms can be seen from [4] Some of the examples include: Point-to-Curve matching with heading (White et al. 2000), Curve-to-Curve matching (Bernstein and Kornhauser (1996) White et al. (2000) Taylor et al. (2001)), Similarity criteria by weighting system (Greenfeld, J.S. (2002)).

3. ROAD NETWORK DATA AND STRUCTURE IN THE SOLUTION

- Node

In digital road map, the road is kept as a line object which is essentially a series of points. If the points are connected then a road network can be shown. Connectivity nodes are intersections of roads, the beginning and the end of a road and the points where vehicles can turn. Other points which does not belong to connectivity node are called common nodes.

- Road segment

If there is a directional pathway between two adjacent nodes then this path is defined as a road segment. Each segment has two nodes which are beginning node and ending node of the segment.

- Link

If there is a directional path between two adjacent connectivity nodes, then this path is defined as a link and the connectivity nodes are respectively the beginning node and ending node of link. Each link can be connected to one or many road segments. From figure 1 we can see how the road network is formed. We

can see that nodes are connected to each other by directional links and if two nodes are connected only by one link then the road is one-directional road but if they are connected by two links which have the opposite direction then the road would be two-directional. From figure 1 we can see that in this example all the nodes are connected by only one link to each other so the road is one-directional.

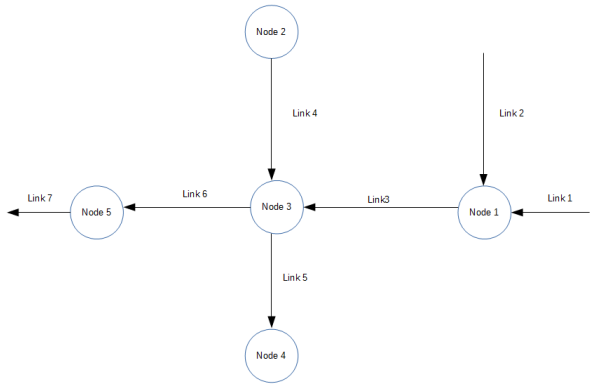


Figure 1 Topology of the graph

Several tools were explored for importing the map data:

- osm2pgsql
- osm2pgrouting
- osmosis
- osm2po
- shp2pgsql

For all of these tools postgresql[5] database with postgis extensions were necessary. Osm2pgsql[6] is a command-line based program that converts OpenStreetMap data to postGIS-enabled PostgreSQL databases. The problem with osm2pgsql was that it had only official Linux support and Cygwin[7] was required to build it on Windows. It required several dependencies which were no easily installed on Windows so it was discarded from selection. Shp2pgsql[8] converts a shape-file into a series of SQL commands that can be loaded into a database–it does **not** perform the actual loading. The output of this command may be captured in a SQL file, or piped directly to the psql command, which will execute the commands against a target database. Shp2pgsql was not suitable for routing due to incompleteness of osm shape-files so broken roads could affect the routing. Osm2po[9] converter parses OpenStreetMap's XML-Data and makes it routable. It can convert large sets like the whole map of Europe and runs both on Windows and Linux platform. Osm2po was one of the two better choices for parsing the map data into database. It also had a built in routing service which was faster than PG-routing but slower than osrm (more on routing in section 4. Routing). Osmosis[10] Osmosis is a command line Java application for processing OSM data. The tool consists of pluggable components that can be chained to perform a larger operation. For example, it has components for reading/writing databases and files, deriving/applying changes to data sources, and sorting data, etc. The downsize of osmosis is that it creates a it creates one row for each attribute that belongs to a geometry. So, if our geometry has 5 attributes, there would be 5 rows with the structure: ID, Attribute. If we have a big amount of data then this would be undesirable. The last tool and also the tool used for importing osm data to postgresql was osm2pgrouting. Osm2pgrouting[11] is good for creating a routing topology but the downside is that it basically works only with roads. Also different from osmosis osm2pgrouting does not have offficial binaries for Windows platform but can easily be built with the help of cygwin. It required some additional libraries like boot, expat, cmake and also that postgresql, postgis and pgrouting were installed to import the osm data. It could be run from the terminal with the following command: `osm2pgrouting -file your-OSM-XML-File.osm -conf mapconfig.xml -dbname routing -user postgres -clean` where the mapconfig file defined the acceptable highway, cycle, track and junction types. One of the reasons for choosing osm2pgrouting instead of osm2po was because it created better tables for pgrouting topology. And with osm2po with larger maps some edge connection information was corrupted also Osm2po is also not open sourced, it is freeware.

4. ROUTING

Two different algorithms with four different tool were tested to see which one was the most suitable for real-time routing approach: A* (and Dijkstra) and contradicting hierarchies. Osm2po, Routino[12] and Pg-routing used some version of A* (or Dijkstra) and OSRM uses contradicting hierarchies. PG-routing extends the PostGIS / PostgreSQL geospatial database to provide geospatial routing functionality. The advantages of PG-routing also include that the data is easily accessible via many clients including: QGIS and uDig through JDBC, ODBC, or directly using Pl/pgSQL. The clients can either be PCs or mobile devices. And Data changes can be reflected instantaneously through the routing engine. This was the first approach tried and the routing functionality worked well but the routing queries took around a second per query. This is definitely not suitable for real time applications since for each GPS point we need to perform around 3-7 routing queries. Later I managed to make it faster including a bounding box for the routing query and this scaled the timings down to couple of hundred milliseconds but I had already switched to OSRM routing by then. Pg routing uses Dijkstra and A* for finding shortest path. Routino uses modified A* algorithm and it discards most of the nodes while keeping only interesting nodes (e.g Nodes which have at least a junction of 3 ways or a junction of two ways with different properties.) it gave similar results to PG-routing in regards of time used without optimization. Since Routino did not improve the computation time I decided to try something different. I decided to try **OSRM (Open Source Routing Machine)**[13] as the routing component of the application. It computes shortest paths in a graph. It was designed to run well

with map data from the OpenStreetMap Project. OSRM does not use an A* variant to compute shortest path, but Contraction Hierarchies. This results in very fast query times, usually below 100ms for data sets like Europe, making OSRM a good candidate for responsive web-based routing applications and websites. C**ontraction hierarchies**[14] is a technique to speed up shortest-path routing by first creating precomputed "contracted" versions of the connection graph. it generates a multi-layered node hierarchy in the preprocessing stage. When preprocessing of the original graph is done, we have a CH graph which consists of the original graph with node ordering added and with shortcut edges introduced. For querying a bi-directional Dijkstra is used – the algorithm searches from both the starting node and the end node. If the shortest path exists, those two searches will meet at some node v. The algorithm finds the shortest path due to preprocessing stage.

OSRM-backend (the name of the OSRM routing component) processing flow composes of several steps. The main flow is as follows: import raw osm data, compute routes, serve data. *Osrm-extract* generates .osrm and .osrm.restrictions it takes a profile field as an argument to specifie the roadtypes used for the routing. I used a default car profile for testing the osrm routing functionality. It defined several restrictions which made the routing use ways only vehicles can access. This helped to reduce routing errors due to the fact that some pedestrian roads would be used which cars cannot travel. The profile file also defined default speed limits for different types of roads. For example motorway limit is 90 km/h residential road is 25 km/h etc. *Osrm-prepare* reads the extracted data and prepares it for routing. What it basically does is that it will read the Intermediate OSRM format files generated from extraction and creates OSRM server data files (edges, nodes etc). And the most important step is *osrm-routed.* It loads the prepared data and starts a routing server. As I used local server for routing then default configuration was suitable. After running the command osrm server will respond to routing requests received on a specific IP and port, and return computed routes. The service I need for routing is called viaroute. It provides shortest path queries with multiple via locations. Example query as below:

http.///{server}/viaroute
loc={lat,lon}&loc={lat,lon}<&loc={lat,lon} ...>

where loc is the previouly matched gps canditate coordinates on roads and last loc is the current gps candite point. This allows to get all shortest paths or fastest paths for markov model.

5. IMPLEMENTED MAP-MATCHING ALGORITHM

The first approach was to start implementing my own Markov model in java. This was deemed too time consuming since creating an effective and fast Markov model approach would take too much time. The next approach was to see if there were already some open sourced hidden Markov models implemented. Some of the implementations I looked at were JAHMM[15] and hmm-lib[16]. Both of them support the use of Viterbi[17] algorithm for calculating the most likely sequence of hidden states. JAHMM also includes forward – backward algorithm[18]. I decided to try to use hmm-lib and see if I can use it for my implementation. The Viterbi algorithm is a dynamic programming algorithm for finding the most likely sequence of hidden states – called the Viterbi path – that results in a sequence of observed events, especially in the context of Markov information sources and hidden Markov models. It can be used to quickly find the most suitable matched gps points on the road. The complexity of the algorithm is O(T * |S|) where S is the state space and T is the number of outputs. We give the observation space, state space, the sequence of observations, transition matrix, emission matrix and initial probabilities as input and get the most likely hidden state sequence as output. In my approach each position measurement triggers an iteration of the algorithm and updates the position estimates.

The current approach is to first determine map position candidates on road segments that have minimum geodesic distance to the measurement gps point. The road segments must overlap with a square around the gps point. The emission probability is calculated via the following formula:

$$p(z_t|s_t) \sim \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\left\{-\frac{||z_t - s_t||^2}{2\sigma_z^2}\right\}$$

where theta z is the standard deviation of gps measurements and z_t - s_t is the distance between the gps point and the matched point on the road segment. For finding the transition probabilities calculating the route from previous positioning candidate to current candidate is required. In [1] they found that transition probabilities have been experimentally determined to fit a negative exponential distribution:

$$p(s_t|s_{t-1}) \sim \lambda \exp\left\{\lambda\left(||z_t - z_{t-1}|| - |\langle s_t - s_{t-1}\rangle|\right)\right\}$$

where it remains to find the best parameterization (estimate of λ) for a specific sampling set, i.e. the sequence of position measurements. Then use the Viterbi algorithm to calculate the most suitable sequence for current states (gps points) and start the next iteration until all gps points are matched.

6. DISPLAYING THE MATCHED POINTS

Several different approaches were explored when deciding which tool to use for displaying the map matched GPS points (and the possible routes between points). The first approach was to use GeoServer [21] as the software server to view the routes and use Openlayers to display the map. This solution was considered because it had good integration with the PG-routing router. First I had to write a pl/pgsql wrapper to make it easier to use PG-Routing functions with different input parameters (taking the start and end point's coordinates) and transform the result into format which is easier to read by application (see EXTRAS for

Dijkstra wrapper). Also to simplify things a secondary wrapper for returning a route between points A and B had to be created in order to display the map in QGIS or GeoServer (see EXTRAS). The route wrapper basically does 5 things:

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

What it is lacking by default is that there is no bounding-box for the query (this I discovered later and it improved the time a lot as well) and it cannot handle one way streets.

Next step was to install the GeoServer and create a WMS layer in it (Sql view). The main part of the view is to add the previously created wrapper sql for the layer sql part.

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

Also in SQL view parameters : Guess parameters from SQL must be selected. And for each parameter the default value must be zero and I tried with the following validation regular expression: `^-?[\d.]+$` , which should only validate numbers. And in the attributes values the type of the st_makeline is LineString. Also the coordinates reference should be in `EPSG:3857` instead of `EPSG:4326` . The last part was to create a simple Openlayers map and add the WMS GET parameters variable to the map code which holds the WMS GET parameters that will be sent to GeoServer. And for executing database queries The viewparams property is then 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. And the response from the GeoServer is then drawn as a new layer. This approach was dropped when PG-ROUTING was deemed too slow for real time applications. Additionally before limiting the layer query SQL statement it queried the whole table making it unusable with larger maps (whole Europe for example, with Estonia seemed to be working fast ~100 – 200 ms).

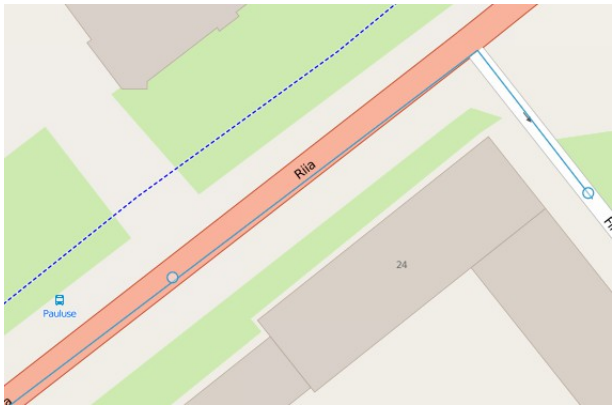
The second platform considered was Mapnik[22]. Mapnik is basically a collection of geographic objects like maps, layers, datasources, features, and geometries. For example OpenStreetMap runs on Mapnik service. A simple python based map was implemented and it seemed to work really well. But Mapnik was discarded for a number of reasons: lack of up-to-date Java bindings and the complexity of setting it up. Since I did not want to convert my whole program to C++ or Python at this stage I decided to not use Mapnik for displaying the map-matched route due to the previously mentioned reasons.

Openlayers 3 and leaflet were the main map services considered for displaying the map. Openlayers has really good integration with PG-routing and Mapnik map server where as leaflet has better integration with OSRM routing service.

Leaflet has several open source implementations that already utilize the OSRM API for example Leaflet Routing Machine or Project OSRM Frontend which is purely Javascript written frontend to enable OSRM based routing. Although they provide OSRM server api support and enable to make routing calls from the frontend, this was not required in the current solution since only one route per vehicle is required to be displayed at the end of the map-matching process. Final map framework to be used was Openlayers3.

Openlayers is really similar to Leaflet but it has more features is abit more complex and has a greater size ~ 1 mb vs Leaflet ~100 kb. So we may ask why not use Leaflet? For me it was simple – Openlayers just has more mature community and a lot more documentation and examples available for it. Leaflet has tutorials about creating a routing interface for OSRM router but this is not needed in this project.

The proposed map solution would initially only read in the matched route from a file and display it on the map. Openlayers streams the map from web so no additional map files are required for it to work. Main functions required to draw the routes onto the map are ol.geom.LineString for drawing the most likely route and ol.geom.Point for drawing the matched points. The points and linestrings are created on Openlayers feature which are added to the map layer and made visible. The map page itself is a simple html page which consists of Openlayers JavaScript file and The map JavaScript file which has the necessary functions for displaying the matched route and points.



Openlayers mapblue line is the route, dots represents the matche points

7. MAP DATA STRUCTURE

The following data structures were considered when trying to improve the finding candidate map segments for first GPS point matching: R-tree[19] and Quad-tree[20]. During las project kd-tree and interval-tree were considered as well but found not suitable for the following reasons:

Kd-tree was not suitable in my implementation since I was using map segments as my data. Each map segment has a start coordinates and end coordinates. Since Kd-tree allows for fast nearest-neighbor searches for points then it would allow us to find the nearest intersections to the GPS point. But if we imagine a case where the GPS point is near map segment but the starting and ending points of the segment are further away than some other intersection point which is not suitable in our case then we either do not find any suitable map segments to start mapping or start mapping from the wrong map segment which gives us false information. When looking into interval/ segment trees I found that 2 dimensional tree would be too time consuming to implement and was not too certain if it would be a good choice.

In this project I experimented with R-tree and quad-tree.

R-tree was considered since R-tree provides spatial access methods and is used in indexing multidimensional information. The main idea behind R-tree is to group together and represent nearby objects in within a minimum bounding rectangle in the higher level of the tree. Since Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. R-tree seemed o be suited for storing map segments and for speeding up finding the candidate segments for the first GPS coordinate. The obstacle in implementing R-tree was that most of the existing implementations in Java were used to store simple data structures namely arrays of integers depicting either lines or rectangles. Then I found an implementation which could be quickly integrated with my project and did not require any external libraries. Currently my project is not using R-tree but I have implemented it and am trying to adapt it for my needs. There are three issues remaining: dividing map into rectangles and assigning a limit to the maximum number of elements in one rectangle. Dividing the map into rectangles can be done by simply taking the bounding box of the map and dividing it into rectangles of equal size. For assigning the rectangle capacity limit I'm currently just trying to get the maximum number of map segments that fall into one rectangle. The last issue is to decide what to do with map segments that fall into several rectangles. Meaning that the start of a segment is in one rectangle and the end is in another one. Currently the idea is to put the segment into both of the rectangles. This should enable me to use nearest-neighbor search that gives n nearest map segments to the first GPS point in reasonably fast time. The complexity of searching a R-tree is $O(M \log_M n)$ where M is the maximum number of elements in a page(rectangle) in the R-tree.

8. FUTURE DEVELOPMENTS

The main point is to get the algorithm working since most of the time was used on preprocessing and setting up so that it should be possible to consider real time application. One other thing is to add visualization feature to display the map-matched route on map (already working on it too – Openlayers 3 and Leaflet are two different libraries being considered for that). The next major step would be to add server side logic and some kind of job balancer to run multiple instances of the program as to make it more effective.

9. CONCLUSION

As all of the different components (map , router, visualization, matching) are not yet merged into a full working solution, I cannot comment on the accuracy of the hidden Markov model based map matching but what I can see is that using OSRM for routing and R-tree for storing the road segments of map data seems quite efficient. When I implemented the Vector based approach I used simple arrays to hold road data and it took seconds to find the first point and all of the nearby road segments. Now finding the n nearest road segments takes a couple of milliseconds on the scale of Estonia. Using OSRM router it takes 20-30 milliseconds to calculate a shortest path using *via* function. I think this can definitely be improved when running the routing server on native platform. Currently it runs on a virtual 32 bit Ubuntu and the rest of the program is running in Windows 8. Also OSRM provides c++ api functionality and it is probably faster to use that than to perform HTTP requests against the server in Java. The reason why I did not use it is because I^am not proficient in C++ and I switched to OSRM router after a lot of preproccesing was already done in Java. With currently selected data structures and router I believe that it is possible to create a map-matching solution which can run in real time.

10. REFERENCES

1. Kazi Khaled Al-Zahid, Birgit Engelmann, Andreas Hildisch, Stefan Holder, Olexiy Lazarevych, Daniel Mohr, Felix Sedlmeier, Sebastian Mattheis, Richard Zinck. Putting the car on the map: A scalable map matching system for the Open Source Community

2. Floating Car Data: http://en.wikipedia.org/wiki/Intelligent_transportation_system#Floating_car_data.2Ffloating_cellular_data

3. Mohammed A. Quddus, Washington Y. Ochieng , Robert B. Noland. Current map-matching algorithms for transport applications: State-of-the art and future research directions

4. Quddus, Mohammed A. Ochieng, Washington Y. Zhao, Lin Noland, Robert B. A general map matching algorithm for transport telematics applications

5. PostgreSQL open source object-relational

database system: http://www.postgresql.org/

6. osm2pgsql a command-line based program that converts OpenStreetMap data to postGIS-enabled PostgreSQL databases http://wiki.openstreetmap.org/wiki/Osm2pgsql
7. Cygwin - a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows. https://www.cygwin.com/
8. Tool for importing shapefiles to postgresql database: http://man.cx/shp2pgsql%281%29
9. osm2po a converter and a routing engine. http://osm2po.de/
10. Osmosis is a command line Java application for processing OSM data: http://wiki.openstreetmap.org/wiki/Osmosis
11. osm2pgrouting is a command line tool that allows to import OpenStreetMap data into a pgRouting database http://pgrouting.org/docs/tools/osm2pgrouting.html
12. Routino is an application for finding a route between two points http://wiki.openstreetmap.org/wiki/Routino
13. A high performance routing engine written in C++11 designed to run on OpenStreetMap data. http://project-osrm.org/
14. contraction hierarchies is a technique to speed up shortest-path routing https://en.wikipedia.org/wiki/Contraction_hierarchies
15. A Java implementation of Hidden Markov Model https://code.google.com/p/jahmm/
16. Hmm library in Java https://github.com/bmwcarit/hmm-lib
17. Viterbi algorithm https://en.wikipedia.org/wiki/Viterbi_algorithm
18. https://code.google.com/p/jahmm/wiki/Algorithms
19. Kd-tree: http://en.wikipedia.org/wiki/K-d_tree
20. Quadtree https://en.wikipedia.org/wiki/Quadtree
21. GeoServer an open source server for sharing geospatial data http://geoserver.org/
22. Mapnik is an open source toolkit for developing mapping applications. https://github.com/mapnik/mapnik
23. Leaflet is a JavaScript library for mobile-friendly interactive maps http://leafletjs.com/

EXTRAS

```
--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;
```

Dijkstra wrapper

```
--
--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 ways_vertices_pgr
                        ORDER BY the_geom <-> ST_GeometryFromText(''POINT('
                        || x1 || ' ' || y1 || ')'',4326) LIMIT 1' INTO rec;
        source := rec.id;

        EXECUTE 'SELECT id::integer FROM ways_vertices_pgr
                        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;
```

Route wrapper