Rewrite of Turn Restricted Shortest Path Algorithm in PgRouting

Contact details

- **Name:** Vidhan Jain
- **Country:** India
- **Email:** vidhanj1307@gmail.com
- **Phone:** +91-8890873934
- **Location :** Jaipur, India, +5.30 GMT
- **Github:** vidhan13j07
- **Twitter:** **vidhanj13**

Title

Rewrite of Turn Restricted Shortest Path Algorithm in PgRouting.

Project Description in Brief

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. In real life scenario, the road network is modeled as a graph where the graph's vertices correspond to road junctions and the edges correspond to road segments, each weighted by the positive length of its road segment.

A turn models a movement from one edge element to another. Often turns are created to increase the cost of making the movement, or prohibit the turn entirely. For example, a turn feature representing a left-hand turn at an intersection could be assigned a cost of 30 seconds to model the average time it takes for the left-turn light to change to green. Similarly, a restriction attribute could read a field value from a turn feature to prohibit it. This is useful when the turning movement is posted as illegal (no left turns).

Current Status of The Software

TRSP is implemented in PgRouting but the code has issues which are not fixed as yet. Wrappers are added to the codebase that enhance the functionality of TRSP but fail to fix the issues. The following functionality is added through the use of wrappers:-

- pgr_dijkstraTRSP(one to one) (aka pgr_trsp)
- pgr_withPointsTRSP(one to one) (aka pgr_trsp)
- pgr_dijkstraViaTRSP() (aka pgr_trspViaVertices)
- pgr_withPointsViaTRSP() (aka pgr_trspViaEdges)

Here is the link to a wiki page that describes the issues associated with the current implementation of the pgr_trsp code in PgRouting.

TRSP for single source and multiple destinations as well as multiple sources and single destination is currently not implemented in PgRouting. Similarly, TRSP for multiple sources and multiple destinations is also not implemented in PgRouting.

Benefits

- TRSP will help in calculating the shortest path in graphs where certain restrictions are imposed such as no-left-turn, no-right-turn etc.

- Shortest path routing often returns results with a lot of turns(zigzag shape). In order to find routes with fewer turns, we can impose certain turn restrictions or penalties in the graph. Slowing down, turning and accelerating costs more fuel than driving straight ahead. Therefore, fuel-efficient routes form a possible use case.
- Another possible use case could be in case of high traffic on a particular road or highway, we can impose a turn restriction for a timeframe on that particular road or highway in order to ease the traffic.

Deliverables

The deliverables include:

❏ pgr_dijkstraTRSP(one to one)

```
pgr_dijkstraTRSP(TEXT edges_sql, TEXT restrict_sql, BIGINT start_vid, BIGINT
end_vid, BOOLEAN directed:=true)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost) OR EMPTY SET
```

Timeline

❖ **Community Bonding Period(May 5 - May 30)**
- Create a wiki page for TODO tasks and weekly reports.
- Look into the currently implemented code of TRSP.
- Discuss the design and functionality of *pgr_trsp* with the mentors.
- Get familiar with the PgRouting architecture.
- Go through pgTap for creating unit tests for PostgreSQL.
- Watch C++ videos for better understanding of C++ and STL.
- Investigate the handling of costs while conversion of original graph to Line graph.
- Discuss the design of restriction table with the mentors.

❖ **Official Coding Period(May 30 - August 21)**

★ **Official Coding Period Phase 1(May 30 - June 26)**

- **Week 1(May 30 - June 5)**

➔ Implement the basic code required for connecting the *pgr_dijkstraTRSP* function to the postgreSQL.

- **Week 2(June 6 - June 12)**
  - ➔ Implement the C code that reads and executes the queries from PostgreSQL.

- **Week 3(June 13 - June 19)**
  - ➔ Design and implement the function that transforms a given graph into its corresponding **Line graph**.

- **Week 4(June 20 - June 26)**
  - ➔ Create unit tests to test the transformed **Line graph**.
  - ➔ Implement a testing function that tests the graph transformation.
  - ➔ Fix bugs found in the implementation of the above function.
  - ➔ Work on the submissions required as part of the first evaluation.

★ **First evaluation period(June 26 - June 30)**

- Mentors evaluate me and I evaluate the mentors of Coding Period Phase 1.
- Deliver a working implementation of the transformation function into the Line Graph.

★ **Official Coding Period Phase 2(June 27 - July 24)**

- **Week 5-6(June 27 - July 10)**
  - ➔ Work on the feedback as provided after the first evaluation.
  - ➔ Implement the **dijkstra** algorithm on the **transformed Line Graph** where some possibilities can be:-
    - Using the simplified version of pgRouting dijkstra directly.
    - Using boost library's dijkstra directly.
    - Add additional functionality to the pgRouting simplified dijkstra.

- **Week 7-8(July 11 - July 24)**
  - ➔ Implement unit tests to verify the result of the dijkstra algorithm on the transformed Line Graph.
  - ➔ Fix bugs found in the implementation of the dijkstra algorithm on the transformed Line graph.

➔ Work on the submissions required as part of the second evaluation.

★ **Second evaluation period(July 24 - July 28)**

- Mentors evaluate me and I evaluate the mentors of coding period phase 2.
- Deliver a working implementation of the dijkstra on the Line graph.

★ **Phase 3(July 25 - August 21)**

- **Week 9(July 25 - July 31)**
  - ➔ Work on the feedback as provided after the second evaluation.
  - ➔ Implement the functionality that extracts the relevant result after applying dijkstra on the transformed Line graph.
- **Week 10(August 1 - August 7)**
  - ➔ Create tests and rigorously test the above functions and fix bugs if found any.
- **Week 11(August 8 - August 14)**
  - ➔ Create developer and user documentations.
  - ➔ Create documentation tests.
- **Week 12(August 15 - August 21)**
  - ➔ Work on the final submission report.
  - ➔ Prepare for the final delivery.

★ **Final evaluation period(August 21 - August 29)**

- Wrap up the project and submit final evaluation of my mentors of Coding Period Phase 3.
- Deliver a working implementation of *pgr_dijkstraTRSP.*

And if time permits, implement:-

- `pgr_dijkstraTRSP`(one to many)
- `pgr_dijkstraTRSP`(many to one)
- `pgr_dijkstraTRSP`(many to many)

Weekly Report Format:

- Work done in the week.
- Problems faced during the work.
- Work to be done next week.

Do you understand this is a serious commitment, equivalent to a full time paid summer internship or summer job?

Yes, I completely understand and am aware of my commitment levels. I am fully prepared for the work and will put in my best efforts.

Do you have any known time conflicts during the official coding period?

I do not have any known conflicts during the official coding period.

Related Work

Turn restriction shortest path algorithms are implemented and used in many organisations such as Open Source Routing Machine(http://wiki.openstreetmap.org/wiki/Open_Source_Routing_Machine), GraphHopper(https://github.com/graphhopper/graphhopper) and Mapbox(https://github.com/mapbox).

Biographical Information

I am a pre-final year undergraduate student pursuing my Bachelors of Technology in Computer Science and Engineering from The LNM Institute of Information Technology, Jaipur. I have been programming in C++ and STL since last 2 years. I am an active open source contributor. I have contributed to organizations such as Coala, Mozilla Marketplace, DuckduckGo in the past. I had also won a goodie for completing 4 Pull requests to various open source organizations by Digital Ocean in the Hacktoberfest. Apart from open source

development, I also like to take part in competitive programming competitions solving algorithmic problems hosted by Codeforces, Topcoder, HackerRank etc. I also enjoy Table Tennis and Badminton in my free time.

Studies

What is your School and degree?

School: The LNM Institute of Information Technology in India.
Degree: Bachelors in Computer Science and Engineering

Would your application contribute to your ongoing studies/degree? If so, how?

Yes, this application will contribute much to my ongoing studies. This project will be an added asset to my ongoing degree and would help me gain a better understanding on how GIS softwares work. I will get hands on experience on shortest path algorithms and pgRouting, and the experience and information I will obtain in the course of contributing to PgRouting will contribute a lot to my B Tech. project that I have to take up in my next semester.

Programming and GIS

Computing experience

Languages: C, C++, python, bash, java, javascript.
APIs: Google maps API.
Operating Systems: Linux(Ubuntu 14.04), Fedora 23, Windows 7,8 &10.
Frameworks: Flask, Django
Sport Programming: **Codeforces**, **Topcoder**, **HackerRank**, **Codechef**, **HackerEarth**

GIS experience as a user

I have used libraries such as pgRouting, osm2pgrouting etc.

GIS programming

I have contributed this Pull request to the pgRouting as part of its 2.4 release.

GSoC participation

- **Have you participated to GSoC before?**

  Yes I have applied to GSoc.

- **How many times, which year, which project?**

  I had submitted a proposal to WikiToLearn Rating project under KDE umbrella in GSoc'16 but was not selected.

- **Have you submitted/will you submit another proposal for GSoC 2017 to a different org?**

  I did not submit a proposal to any other organisation.

Proposal

A graph is a mathematical tool for representing entities and the connections between them. In the graph model of the road network, the entities are intersections and the roads between them form the connections. The weight of an edge denotes the cost to traverse it in the chosen mode of transport. This can be:-

- travel time for car routing
- distance for walking directions
- a combination of time and energy-consumption for electric vehicles.

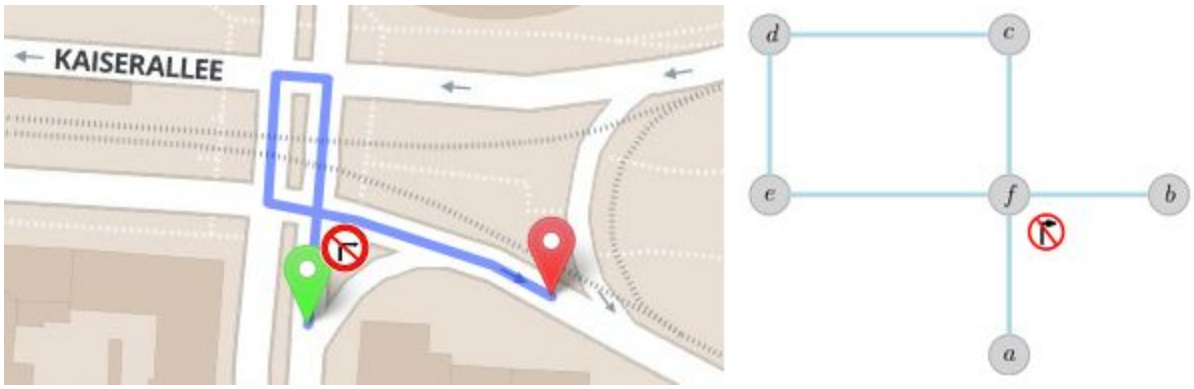
One of the basic algorithms for routing is Dijkstra's Algorithm, which solves the problem of finding a shortest path, i.e. a path of minimum weight.
Let us consider 2 nodes source A and destination B.
So, in order to route from A to B, for each iteration in Dijkstra's algorithm, we try to find and settle the nearest unsettled intersection where settling means assigning a definitive distance from the source A to the current node. Apart from that, this is the closest intersection to any previously settled nodes which is not settled itself. Intersections adjacent to settled ones are assigned a tentative distance that may decrease if a lower distance path is found during the search. In a sense, the algorithm searches radially around the source until it finds the destination.

More formally, following is the algorithmic approach:-

Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for the initial node and to infinity for all the other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes denoting the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited or if the smallest tentative distance among the nodes in the *unvisited set* is infinity, then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new **current node**, and go back to step 3.

Though, the above algorithm finds a path from node A to node B, but the algorithm has its shortcomings. The above algorithm does not account for the turn restrictions in the graph.



Consider the above real world example and its graph representation,
Nodes = {a, b, c, d, e, f}
Edges = {a-f, f-b, f-c, c-d, d-e, e-f}
There is a turn restriction imposed while routing from a-f to f-b.

Suppose we need to find a path from source node a to destination node b.

- The search starts from node a and settles it.
- The search then proceeds to node f and settles it.
- Since the path from f to b is forbidden as there is turn restriction involved therefore node b is not explored yet.

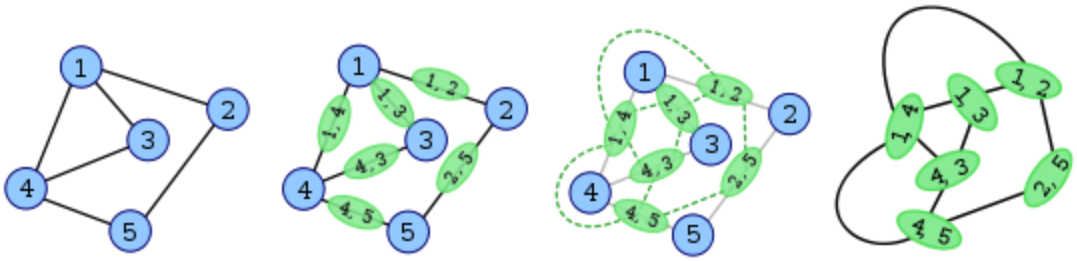
- The search then continues to node c then proceeds to node d and node e expanding each node.
- Now the search stops, since the node f has already been explored and settled with a lower weight.

Therefore, the node b remains unvisited as the road from f-b is never traversed. And if we allow the search to revisit the intersection after it is settled, we sacrifice efficiency and therefore practicality in our approach.

In order to approach the above problem, we model the above graph into a line graph.
Formally, Given a graph *G*, its Line graph *L*(*G*) is a graph such that:

- each vertex of *L*(*G*) represents an edge of *G.*
- two vertices of *L*(*G*) are adjacent if and only if their corresponding edges share a common endpoint in *G.*

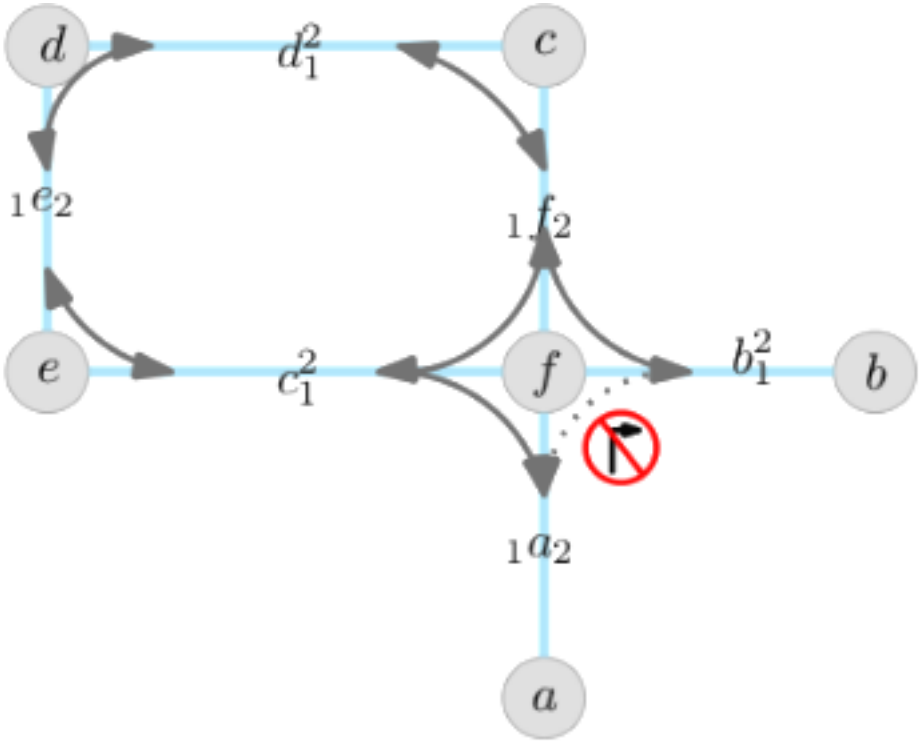
That is, it is the intersection graph of the edges of *G*, representing each edge by the set of its two endpoints.



The above figures show a graph (left, with blue vertices) and its line graph (right, with green vertices). Each vertex of the line graph is shown labeled with the pair of endpoints of the corresponding edge in the original graph. For instance, the green vertex on the right labeled 1,3 corresponds to the edge on the left between the blue vertices 1 and 3. Green vertex 1,3 is adjacent to three other green vertices: 1,4 and 1,2 (corresponding to edges sharing the endpoint 1 in the blue graph) and 4,3 (corresponding to an edge sharing the endpoint 3 in the blue graph).

Here is a C++ implementation, given a small graph(< 10 nodes), it converts the graph into the corresponding Line Graph.

Consider the real world example that we used above, we'll model that graph into its corresponding Line Graph.
Since the Dijkstra algorithm cannot guarantee a path when turn restriction is imposed so we try to model roads and turn from one to the other. Therefore, roads become our entity of interest and turns are the connections that we would like to model.



In the above figure, the grey colored arrow shaped edges in the picture now reflect turning from one road to the next. The weight of each expanded edge is the cost of its first edge plus the cost of the turn. In cases of turn restriction, we assign the cost of the turn to be infinite.
Applying Dijkstra on the above Line Graph, now, when going from a to b, the algorithm finds a path **a** → **f** → **c** → **d** → **e** → **f** → **b**.

Handling of Costs

While converting the graph to its corresponding Line Graph, the costs are handled as demonstrated here.

Implementation Details

The results below are derived based on the sample data from pgRouting.

- **One to One**

```
pgr_dijkstraTRSP(TEXT edges_sql, TEXT restrict_sql, BIGINT start_vid, BIGINT
end_vid, BOOLEAN directed:=true)
RETURNS SET OF (seq, path_seq, node, edge, cost, agg_cost) OR EMPTY SET
```

This signature deals with finding the shortest path from one **start_vid** to one **end_vid** along with **turn restrictions**.

```
SELECT * FROM pgr_dijkstraTRSP(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM
edge_table',
        'SELECT to_cost, target_id::int4, from_edge FROM restrictions', 2,
7,
        false
);

 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+---------
   1 |        1 |    2 |    4 |    1 |       0
   2 |        2 |    5 |   10 |    1 |       1
   3 |        3 |   10 |   12 |    1 |       2
   4 |        4 |   11 |   11 |    1 |       3
   5 |        5 |    6 |    8 |    1 |       4
   6 |        6 |    5 |    7 |    1 |       5
   7 |        7 |    8 |    6 |    1 |       6
   8 |        8 |    7 |   -1 |    0 |       7

(8 rows)
```

```
SELECT * FROM pgr_dijkstraTRSP(
        'SELECT id::INTEGER, source::INTEGER, target::INTEGER, cost FROM
edge_table',
        'SELECT to_cost, target_id::int4, from_edge FROM restrictions', 7,
```

```
12
  );


 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    7 |    6 |    1 |        0
   2 |        2 |    8 |    7 |    1 |        1
   3 |        3 |    5 |    8 |    1 |        2
   4 |        4 |    6 |    9 |    1 |        3
   5 |        5 |    9 |   15 |    1 |        4
   6 |        6 |   12 |   -1 |    0 |        5
(6 rows)
```

- **To be implemented if the time permits**

  ➢ **One to Many**

  Using this signature, the graph will be loaded once and a one to one *pgr_dijkstraTRSP* is performed where the starting vertex is fixed, and stopped when all end_vids are reached.

  ➢ **Many to One**

  Using this signature, the graph will be loaded once and a one to one *pgr_dijkstraTRSP* is performed where the destination vertex is fixed, and stopped when all start_vids finds path to end_vid.

  ➢ **Many to Many**

  Using this signature, the graph will be loaded once and a one to one *pgr_dijkstraTRSP* is performed for each source to each destination.

Description of the Signatures

Following Terminology is used below :-

ANY-INTEGER: SMALLINT, INTEGER, BIGINT

ANY-NUMERICAL: SMALLINT, INTEGER, BIGINT, REAL, FLOAT

- ## Description of the edges_sql query

  edges_sql: an SQL query, which should return a set of rows with the following columns:-

Column	Type	Description
id	ANY-INTEGER	Identifier of the edge.
source	ANY-INTEGER	Identifier of the first end point vertex of the edge.
target	ANY-INTEGER	Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL	Weight of the edge (source, target) ● When negative: edge (source, target) does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	(optional) Default value is -1. Weight of the edge (target, source), ● When negative: edge (target, source) does not exist, therefore it's not part of the graph.

- ## Description of the restrict_sql query

  restrict_sql: an SQL query, which should return a set of rows with the following columns:-

Column	Type	Description
to_cost	ANY_NUMERICAL	the cost to be associated with the turn.
target_id	ANY_INTEGER	the edge id of the turn restricted edge.
from_edge	ANY_INTEGER	the edge id of the edge from which the turn is restricted.

- **Description of the parameters of the signatures**

Parameter	Type	Description
edges_sql	TEXT	Edges SQL query as described above.
start_vid	ANY-INTEGER	Starting vertex identifier.
start_vids	ARRAY[ANY-INTEGER]	Array of starting vertex identifiers.
end_vid	ANY-INTEGER	Ending vertex identifier.
end_vids	ARRAY[ANY-INTEGER]	Array of ending vertex identifiers.
restrict_sql	TEXT	Restrict SQL as described above.
directed	BOOLEAN	(optional). ● When false the graph is considered as Undirected. ● Default is true which considers the graph as Directed.

- **Description of the return values**

Column	Type	Description
seq	INT	Row sequence.
path_seq	INT	Path sequence that indicates the relative position on the path.
start_vid	BIGINT	Identifier of the starting vertex. To be used when multiple starting vertices are in the query.
end_vid	BIGINT	Identifier of the ending vertex. To be used when multiple ending vertices are in the query.

node	BIGINT	Identifier of the node in the path from start_vid to end_vid.
edge	BIGINT	Identifier of the edge used to go from node to the next node in the path sequence. -1 for the last node of the path.
cost	FLOAT	Cost to traverse from node using edge to the next node in the path sequence.
agg_cost	FLOAT	Aggregate cost from start_v to node.

Future Work

Some ideas (not a part of the proposal) which can be implemented for the future, include the following:

- Addition of *pgr_aStarTRSP* that will use heuristics to find the shortest path in a turn restricted graph where some possible heuristics could be:-
    1. h(v) = abs(max(dx, dy))
    2. h(v) = abs(min(dx, dy))
    3. h(v) = dx * dx + dy * dy
    4. h(v) = sqrt(dx * dx + dy * dy)
    5. h(v) = abs(dx) + abs(dy)

    where,

    dx = x1 - x2, dy = y1 - y2

    (x1, y1) coordinates of source and (x2, y2) are coordinates of destination.

References

1. "Shortest path problem - Wikipedia." https://en.wikipedia.org/wiki/Shortest_path_problem.
2. "Dijkstra's algorithm - Wikipedia." https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
3. "Line graph - Wikipedia." https://en.wikipedia.org/wiki/Line_graph
4. "Turns in the network dataset—Help | ArcGIS Desktop." http://desktop.arcgis.com/en/arcmap/latest/extensions/network-analyst/turns-in-the-network-dataset.htm.
5. "Is there a way to add turn restrictions in A* and ... - GIS StackExchange.", http://gis.stackexchange.com/questions/16411/is-there-a-way-to-add-turn-restrictions-in-a-and-dijkstra.
6. "pgr_trsp - Turn Restriction Shortest Path (TRSP) — pgRouting Manual ....", http://docs.pgrouting.org/latest/en/pgr_trsp.html.
7. "pgr_dijkstra — pgRouting Manual (2.4).", http://docs.pgrouting.org/latest/en/pgr_dijkstra.html.

8. "Modeling Costs of Turns in Route Planning." http://geo.fsv.cvut.cz/gdata/2013/pin2/d/dokumentace/line_graph_teorie.pdf.
9. "Notes on pgr_trsp (2.3.2 release) · pgRouting/pgrouting Wiki · GitHub.", https://github.com/pgRouting/pgrouting/wiki/Notes-on-pgr_trsp--(2.3.2-release).
10. "Efficient Routing in Road Networks with Turn Costs - KIT." https://algo2.iti.kit.edu/1862.php.
11. "Route Planning in Road Networks with Turn Costs." http://lekv.de/pub/lv-turns-2008.pdf.
12. "Routino : Algorithm." https://www.routino.org/documentation/algorithm.html.
13. "Combining turning point detection and Dijkstra's ... - SAGE Journals.", http://journals.sagepub.com/doi/full/10.1177/1687814016683353.
14. "Graph Indexing of Road Networks for Shortest ... - VLDB Endowment." http://www.vldb.org/pvldb/vol4/p69-rice.pdf.
15. "MODELLING TURNING RESTRICTIONS IN TRAFFIC ... - isprs." http://www.isprs.org/proceedings/XXXIV/part4/pdfpapers/410.pdf.
16. "Smart Directions Powered by OSRM's Enhanced Graph Model | Mapbox.", https://www.mapbox.com/blog/smart-directions-with-osrm-graph-model/.