# GISCUP 2015: Notes on Routing with Polygonal Constraints

Martin Werner
Mobile and Distributed Systems Group
Ludwig-Maximilians University Munich, Munich, Germany
martin.werner@ifi.lmu.de

## ABSTRACT

This paper discusses the problem of street network navigation under polygonal constraints from an implementation perspective. It explains the problem chosen for the 4th ACM SIGSPATIAL Cup 2015 challenge. In this context, the paper shortly examines the most important choices taken for one submission to the challenge in the context of related work. It further highlights the structure of the submission including an interactive GUI for navigation algorithm research and hopes to trigger additional developments in this beautiful and fascinating research area.

## Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Path and Cirquit Problems; F.2.2 [**Nonnumerical Algorithms and Problems**]: Routing and Layout

## General Terms

Algorithm, Spatial Computing, Routing, Navigation, Constraints

## Keywords

Location-based Services, Navigation, Routing

## 1. INTRODUCTION

The routing problem (or shortest path problem) is a fundamental problem in graph theory. In this context, the task is usually to find shortest paths between two vertices of a graph, between one vertex and all other vertices of a graph or between all pairs of vertices of a graph.

Especially due to the availability of low-cost and small GPS receivers and digital road maps, these routing problems have made it into everyday life. Location-based services, navigation, and guidance are provided by calculating the shortest (fastest, most efficient) route between two points of interest. These instances of graph search problems are usually formulated in transportation networks in which vertices represent places and edges between vertices represent roads. While for the general shortest path problem, optimal and optimal efficient algorithms are known (e.g., A*), the specific structure and the fact that the graph does change only very slowly, allow for various techniques increasing the computational efficiency drastically.

Furthermore, the application area of navigation provides additional complexities to the problem, most notably constraints like turn restrictions and variable travel times provided by measurement (real time traffic information, RTTI) or time-of-day (rush hours, traffic jams, etc.). While most high performance approaches to routing are based on complex and time-consuming pre-computations, dynamic constraints can render this complete preprocessing useless for a specific shortest path query.

Additionally, there are only few (if any!) readable, concise and efficient implementations of navigation algorithms in the open source domain. This is one of the reasons, why I started working on this challenge. I hope to provide a readable and extendible yet performant implementation to the community.

The 4th GIS-focused algorithm competition, GISCUP 2015, co-located with ACM SIGSPATIAL GIS 2015, features a specific problem of shortest path routing. The question is, how to quickly calculate a shortest path avoiding dynamic polygonal constraints. My contribution has made it under the top three submission and this is an invited paper on this contribution.

*SIGSPATIAL'15 GIS CUP* , November 03-06, 2015, Bellevue, WA, USA
Copyright 2015 ACM . ISBN 978-1-4503-4048-9/15/11
DOI: http://dx.doi.org/10.1145/2835167.2835170 ...$15.00.

### 1.1 Problem Statement

As already stated, the problem of this competition was about street network routing with polygonal constraints. Concretely, the challenge was set up by providing a dataset extracted from OpenStreetMap. This set contained roughly 100,000 polygonal lines (linestrings) representing road segments in a Web Mercator (EPSG:3857) coordinate system often used in online map services. Additionally, a set of points is provided representing the turn points interconnecting different roads. Note that roads can cross each other without a connection at bridges and tunnels. Furthermore, concrete speed attributes per road segment were given to facilitate calculating the fastest route.

**Problem 1**
*Given this dataset and a set of polygons, find the fastest and shortest route, which never enters or crosses the interior (or boundary) of any of the given polygons.*

While this problem is very clear, the solution space is large as the following Section 2 on related work will show. As an implementation challenge, the real problem boils down to a careful and efficient implementation and to choosing the right perspective on this problem.

## 2. RELATED WORK

Efficient graph search algorithms are usually based on Dijkstra's algorithm. In this algorithm, a shortest path tree is expanded. In a widely-used description, the status of each vertex is managed as a color. White means that the vertex has not yet been used in the search, gray means, that the search has seen the vertex but not yet finished, and black means that a vertex has been fully explored and will be used in the search never again.

The algorithm of Dijkstra is driven by the idea of starting with all vertices white, and expanding shortest paths from already constructed shortest paths in an order defined by increasing path length. In this situation, the shortest path is stored by remembering with each vertex the predecessor on the shortest path to the search start. Additionally, the length of each shortest path is stored along with the vertex. A priority queue is used to organize the ordering of expanding shortest paths. A priority queue can usually insert an object with a specified priority and provide quick access to and removal of the element with lowest priority. There are numerous variants with strong impact on the overall performance of Dijkstra and its variants, see [1].

This basic algorithm can be extended in various directions in order to increase routing performance. The most important directions are towards exploring

- smaller graphs,
- fewer outgoing edges,
- better ordering of outgoing edges.

A recent overview has been given by Bast et. al in a survey [1]. For the purpose of this paper, I want to shortly describe only the most important directions of research each of which is realized in many variants in literature.

The most classical extension to Dijkstra's algorithm is known as $A^*$ and is based on guiding the search towards the direction of the goal. In a specific setting (e.g., some properties of the distances involved), this algorithm is optimal and optimally efficient. "Optimal" in this context means that the algorithm always finds the shortest path and does not accept any detours and "optimally efficient" means that any optimal algorithm has to visit a superset of the vertices visited by $A^*$.

However, one can do some of this work in a preprocessing step, often. So it is no fundamental contradiction that many algorithms are faster than $A^*$. But all of them have in common non-trivial preprocessing.

A classical extension based on preprocessing is given by ALT in which landmarks play a fundamental role. A landmark in this context is a specific vertex of the graph such that the distance to and from this landmark for any other vertex is precomputed. Let $L$ be a landmark. By triangle inequality, the following equations are valid:

$$d_{\text{from } L}(w) - d_{\text{from } L}(v) \leq d(v,w) \quad (1)$$
$$d_{\text{to } L}(v) - d_{\text{to } L}(w) \leq d(v,w), \quad (2)$$

where $d_{\text{from } L}$ is the distance from $L$ and $d_{\text{to } L}$ is the distance to $L$. Both distances can be precomputed for each landmark and can be used to estimate the distance to the goal. With a good distribution of landmarks, these bounds are actually more useful than the distance calculations of $A^*$ why this algoritm can reach astonishing performance. Furthermore, this algorithm is compatible with increasing edge lengths due to the fact that the bounds keep valid. More details on this algorithm and its variants can be found in [5].

Other algorithms try to calculate hierarchies, most notably highway hierarchies [9, 10] and contraction hierarchies [4, 2]. In these algorithms, the graph search complexity is reduced by introducing hierarchies of shortcuts in order to have a significantly smaller search space. The main drawback of this method with respect to the problem statement is that the hierarchy is not automatically compatible with polygonal constraints and update strategies seem to be costly and complex. A third approach for speeding up shortest path calculation is given by edge flags in which the navigation graph is divided into cells and for each pair of cells and each edge, a flag maintains whether this is an edge of any shortest path between the two cells. This approach essentially needs an all pair shortest path preprocessing between pairs of cells [7, 6, 8]. With respect to the problem statement, it is quite hard to update this data structure with constraints: imagine two cities connected with two bridges via a river, one in the middle of the city and the other one far away. All shortest paths will use the near bridge. If a constraint now affects this bridge, all precomputed information is useless and in fact misleading, as all searches will be guided to the near bridge.

There are many other approaches and a lot of details that had to be left out in this short paper, but in general update strategies for the mentioned (and other) speedup techniques allowing for increasing and decreasing edge weights are an important and promising research area, today.

## 3. PROBLEM DISCUSSION

With the mentioned background on speedup strategies for vehicle routing, let us shortly discuss the problem statement again, now in the light of all those choices that have to be made: Dijkstra's algorithm and its possible optimizations, especially with respect to the central priority queue, is important. However, I feel that a reasonable implementation of the priority queue should be used instead of blowing up the code complexity for minor improvements. Dijkstra's algorithm is so simple in implementation, that it makes us available a ground truth for the numerous programming errors that might come up when implementing more complicated algorithms.

The second algorithm which must be implemented is $A^*$. This is due to the fact that the shape of the $A^*$ search space reveals many problematic instances in which also other algorithms might have problems. For the given dataset, the bridges have an interesting behavior with respect to $A^*$. From a performance perspective, however, a careful implementation of $A^*$ was even slightly slower than a run of Dijkstra. This is due to the chosen projection in which distance calculation is non-trivial and due to the very small dataset.

With these two algorithms in place, I chose to avoid complex update strategies for constraints, as the dataset is small enough to assume that even with a thorough challenge to the algorithm (e.g., some thousand shortest paths on the same set of constraints) these updates will not pay off. This left me with the final choice of implementing landmark search (ALT) for this challenge. In this way, the preprocessing data is valid even with constraints in place though the usefulness is reduced. The elements of ALT are essentially Dijkstra searches for calculating the landmark distances and a modified Dijkstra algorithm for calculating the shortest path.

This approach was implemented in a specific way as explained in the next section and made it under the top three submissions of the ACM SIGSPATIAL Cup 2015. I want to emphasize that the used algorithm is in no way better than all other variants described in the related work section. The choice is just based on those intuitive observations pointed out in this section, mainly based on the size and projection of the dataset and the ratio of route calculations per constraint set.

## 4. NOTES ON THE IMPLEMENTATION

As the ACM SIGSPATIAL Cup is an implementation challenge, I want to take the opportunity to comment on the framework and implementation aspects of the challenge as well. The following describes some ideas, which might be most useful together with the source code, which will be published as well.

### 4.1 Data Acquisition and Geometry Support

The dataset was formatted as a set of three relevant ESRI shapefiles, which were easy to read with shapelib library [11]. However, due to the numerous shapefiles of the first epoch of the challenge (the dataset was exchanged once due to numerous questions with respect to the first variant), this library was used in conjunction with a code generator creating a class representing a shapefile as well as the set of its attributes in C++. Given a shapefile, this class could be generated, reading the information into a data structure.

In order to manage this data, I used a combination of `boost::geometry` and `std::vector`. In this way, I was able to keep a dumb memory representation of the original dataset in memory as well as an equivalent representation allowing for simple geometric operations. Furthermore, `boost::geometry::index::rtree` provides a peer-reviewed and quite performant implementation of the $R^*$-tree spatial index supporting nearest neighbor as well as range queries and intersections quite easily.

With these choices, it is as easy as using queries like `intersects(poly, road)` to check, whether polygons and roads intersect. Furthermore, and especially with respect to the developed GUI, one can select nearest roads and vertices in single lines reading
`vertex_rtree.query(nearest(point(x, y), 1), ...);`.

This serves as an example why generic programming matters and how powerful it can become. For all this to work, no one has prescribed a specific library or data type. One is still free on everything as long as it supports some resonable axioms (e.g., some given C++ concepts).

### 4.2 Graph Search and Routing

For the graph search part, the Boost Graph Library (BGL) is chosen. Though there are several other libraries, the choice is based on the availability of bidirectional graphs in which an adjacency list models incoming as well as outgoing edges and with the possibility of exchanging the roles of incoming and outgoing edges by generic programming. In this way, one is able to calculate the distances **from** and **to** landmark nodes with the same algorithm and without additional complexities due to the constant `reverse_graph` adapter of BGL. Finally, I am used to generic programming and the BGL and can afford those very long and partly unclear error messages occuring regularly in generic programming.

While implementing the search algorithms, I made use of an important trick in order to conserve runtime: sometimes, a graph search algorithm would have to update the priority of a given vertex in the priority queue. This is a very costly operation, as it is not easy to find the correct element if it does not have lowest priority. Instead of updating the priority queue, one introduces a temporary value for each vertex in which the priority of the last insertion of the vertex into the priority queue is stored. In this way, the same vertex can exist more than once inside the priority queue. However, after retrieving this element from the queue, one checks, that it is not outdated by looking at the last priority value that has been used to insert this element. This trick is only sensible in application scenarios, where the amount of multiple entries in the queue keeps reasonable.

### 4.3 Library Structure

The library was implemented as a header-only library and the graph search algorithms have actually been exported into their own files. This allows for having several interfaces: the benchmark program as required by the challenge, several debug programs to assess reasons for various behaviors, as well as a completely decoupled GUI allowing for interactive search.

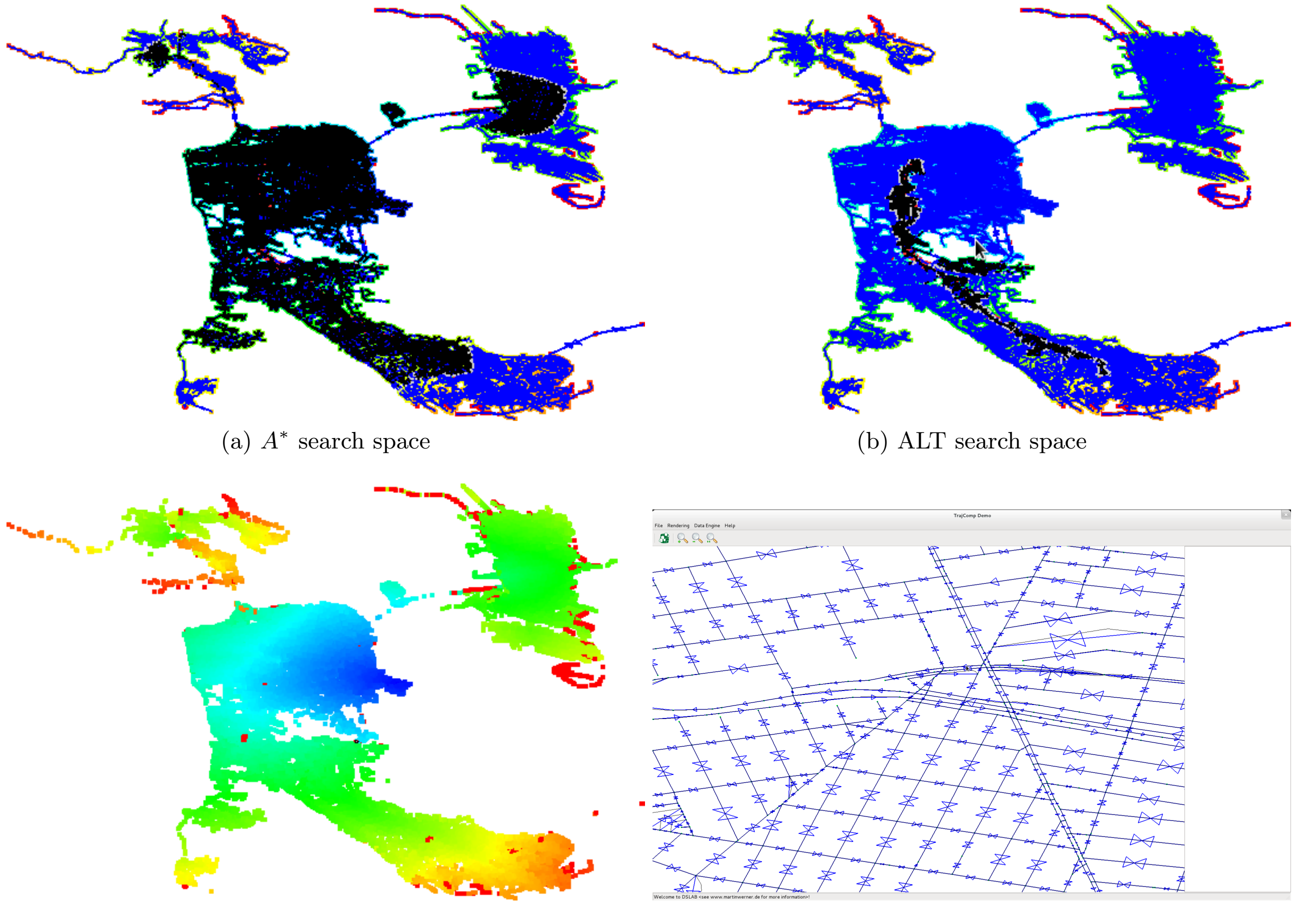
### 4.4 GUI

The GUI has been implemented using dslab [3], which is a cross-platform data science environment essentially providing a framework for writing interactive OpenGL applications across Windows, Linux and Mac. The central implementation `giscup.hpp` contains conditional source code in order to allow for rendering and nearest neighbor search. It is possible to show the street network, the derived graph, the constraint set, the search space, data of a specific landmark and much more. Most notably, it is possible to interactively paint polygonal constraints which are directly applied and it is possible to search with respect to time and distance in this graph.

## 5. CONCLUSION

With the GISCUP 2015 Challenge on routing under polygonal constraints, the ACM SIGSPATIAL GIS conference is providing a great incentive for thinking about and implementing shortest path algorithms. My submission was created with readable code in mind. Only few tricks regarding performance have been used and it is great that the optimization system of GNU compilers together with C++11 and boost are able to create sufficiently fast code without specific, unreadable, non-portable tricks.

The real-world problem of navigation under polygonal constraints is, however, not solved. Not by me and most probably not by other submission to the challenge. This is due to the fact that real-world deployments would have to handle routing graphs that are by numbers of magnitudes larger than the given graph, and constraints that are more complex. In my opinion, the challenge highlighted an important, largely unexplored problem of the present by asking


TrajComp Demo
File Rendering Data Engine Help
Welcome to DSLAB <see www.martinwerner.de for more information>!

(a) $A^*$ search space (b) ALT search space

(c) Landmark Data (d) Graph Structure and Roads

**Figure 1: Screenshots of the GUI**

for submissions regarding a toy problem (small graph, no turn restrictions) of the same kind.

At this point, I want to acknowledge the hard work of the organizing committee for this challenge. It is very hard to create and manage an implementation challenge in such a complicated domain. Thanks for managing the challenge with such a large amount of devotion.

I would love to see this topic develop in the next months and years; my source code will be published to foster quick exchange on the topic and to facilitate real progress. Let us join forces after this great challenge in order to provide a scalable and flexible environment for navigation research!

## 6. REFERENCES

[1] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.

[2] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *ALENEX*, volume 9. SIAM, 2009.

[3] DSLAB Data Science Lab. Online, to be published.

[4] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.

[5] A. V. Goldberg and C. Harrelson. Computing the shortest path: $a^*$ search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[6] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Experimental and Efficient Algorithms*, pages 126–138. Springer, 2005.

[7] U. Lauther. An extremely fast, exact algorithm for finding shor test paths in static networks with geographical background. 2004.

[8] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2–8, 2007.

[9] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Algorithms–Esa 2005*, pages 568–579. Springer, 2005.

[10] P. Sanders and D. Schultes. Engineering highway hierarchies. In *Algorithms–ESA 2006*, pages 804–816. Springer, 2006.

[11] Shapefile C Library. http://shapelib.maptools.org/.