Restricted areas obeying path search algorithm without preprocessing *

František Kolovský
University of West Bohemia
Faculty of Applied Sciences
Univerzitní 8
306 14 Pilsen, Czech Republic
kolovsky@students.zcu.cz

Jan Ježek
University of West Bohemia
Faculty of Applied Sciences
Univerzitní 8
306 14 Pilsen, Czech Republic
jezekjan@kma.zcu.cz

ABSTRACT
Finding the shortest path in a given road network is a well-know problem. In addition to the commonly used restrictions for the searched path, such as speed limits, there might be additional restrictions considered such as polygonal obstacles. This paper describes the shortest path 'obstacle-wise' search algorithm and implementation that was submitted to the ACM SIGSPATIAL Cup 2015. In this paper the implementation and evaluation of various state-of-the-art algorithms dedicated for shortest path search is described. This algorithms were customized to obey polygonal obstacles. The resulting algorithms were extensively evaluated by measuring query performance as well as result quality. The test results show the variant based on Bidirectional Dijkstra's to present the fastest exact solution.

Categories and Subject Descriptors
G.2.2 [**Graph Theory**]: Graph algorithms

General Terms
Algorithm, Performance

Keywords
shortest path search, A*, Dijkstra, Bidirectional Dijkstra, Shortest path with restricted zones

1. INTRODUCTION
This paper discusses the solution designed for the GISCUP 2015 [1], co-located with ACM SIGSPATIAL GIS 2015. The challenge was to deliver a program that will find the shortest path in a given road network while the path will obey turn restrictions, speed restrictions and polygonal obstacles. More concretely the task was to calculate two paths between two given points: the shortest path and the fastest path. The priority of the solution is given to the path correctness and the calculation speed.

The proposed approach is based on a customization of so-called *relaxation* function (a function commonly used by the state-of-the-art path search techniques). Such a function was extended by a mechanism that obey the obstacles while searching for the path. This customized method has been incorporated to the well known search techniques (Dijkstra, Bidirectional Dijkstra and A*). Mentioned algorithms where implemented and tested. Based on the test results that were performed, the most appropriate approach for a given task and evaluation criteria was selected. Integration of restricted polygon recognition into the *relaxation* function enable efficient path search without any preprocessing of the input graph.

In the first part of this paper basic algorithms for the shortest path search are concisely introduced. The next part describes our solution. The design considerations with respect to the given task are discussed. Finally, the implementation and the testing results important for selecting the best option are described.

2. SOLUTION
We implemented and evaluated various state-of-the art techniques. These techniques include: Dijkstra's algorithm, Bidirectional Dijkstra's algorithm, A* algorithm and Bidirectional A* including various concepts for the edge cost. These techniques were extended to fulfil the requirement for obeying the restricted zones. This section summarizes these algorithms and discusses the ideas for incorporating the restricted zones constraint.

2.1 Problem formulation
Let $G = (V, E)$ denote a graph, where V is a set of vertexes and E is a set of edges. The edge is a pair of vertexes $(i, j) \in E$, where i is a source vertex and j is a target vertex. For our scenario we consider a directed graph, where $(i, j) \neq$

(j, i). Every edge has defined a cost as $c : E \to \mathbb{R}_0^+$ for its overcoming.

In this paper the following expressions will be used:

$G = (V, E)$	graph (directed)
c_{ij}	cost from i to j
n	number of vertexes
m	number of edges
P	path (sequence of vertexes)
s,d	source and destination vertex
$dist : V \to \mathbb{R}_0^+$	distance
$prev : V \to V$	previous vertex

The shortest path problem is defined as:

$$min \sum_{i \in P} c(i, i+1),$$

where we search for the minimal path from the source vertex to the destination vertex.

2.2 Algorithms for shortest path search

For the shortest path search problem, there are available many existing algorithms. For the given task, the focus was on those algorithms that do not require any prepossessing. In particular the focus was on Dijkstra, A* and Bidirectional Dijkstra.

2.2.1 Dijkstra

One of the basic algorithm for the shortest path search is Dijkstra's algorithm [2]. This algorithm finds the shortest path from the source vertex to all other vertexes (one-to-all).

In the first step of this algorithm the value of *dist* is set to ∞ for all the vertexes ($\forall i \in V : dist(i) = \infty$) and $dist(s) = 0$ for the source vertex. Algorithm starts in the source vertex s and all s neighbours (j) are traversed and the *relaxation* operation is performed. The *relaxation* operation calculates new distance value in vertex j like $dist(i) + c(i, j)$ and compares this value with an old distance value in j ($dist(j)$). If the new value is smaller then the old $dist(i) + c(i, j) < dist(j)$, the distance value in j is updated ($dist(j) = dist(i) + c(i, j)$). Afterwards the vertex with minimal distance d is selected (without vertexes which have been used) and entire process is repeated.

This algorithm has complexity $m + nlog(n)$ [2]. Section from line 7 to line 10 expresses the *relaxation*.

Algorithm 1 : Dijkstra's algorithm

```
1: ∀i ∈ V : dist(i) = ∞
2: dist(s) = 0
3: Q = Q ∪ s
4: i = s
5: while i ≠ d do
6:    for ∀ neighbours j do
7:       if dist(i) + c(i, j) < dist(j) then
8:          dist(j) = dist(i) + c(i, j)
9:          prev(j) = i
10:         Q = Q ∪ j
11:      end if
12:   end for
13:   i = min(Q)
14:   Q = Q ∖ i
15: end while
```

2.2.2 A*

A* is a speed-up technique for Dijkstra's algorithm. The algorithm is based on a thought that prefers direction to the goal [4]. This algorithm is also called Goal-directed.

Every vertex has a defined *potential* p ($p : V \to \mathbb{R}_0^+$). The potential is defined as an areal distance from the vertex to destination vertex:

$$p(i) = |pos(i) - pos(d)|,$$

where $pos : V \to \mathbb{R}^2$.

Afterwards, the cost of the edge for A* is defined as $ca(i, j) = c(i, j) + p(j) - p(i)$. After that, the classic Dijkstra's algorithm with A* cost (ca) can be used. Finally, the cost of path can be computed as [5]:

$$dist(d) = \sum_{i \in V} ca(i, i+1) = -p(s) + \sum_{i \in V} c(i, i+1)$$

$$c(P) = dist(d) + p(s).$$

The constant k: $ca(i, j) = c(i, j) + k(p(j) - p(i))$ $k \in (0, 1)$ can be added. Such a constant can be set to prioritize algorithm accuracy or speed. This algorithm is usually faster then Dijkstra, but is not exact.

2.2.3 Bidirectional Dijkstra

Bidirectional Dijkstra algorithm is exact speed-up technique [3]. The algorithm starts the search from the source vertex as well as from the destination vertex. Search from the destination vertex is called *backward* and from source *forward*. The forward search is the classical Dijkstra. The backward search uses the inverse edges: $E_i = ((i, j)|(j, i) \in E)$.

Both searches meet approximately in the middle of the source and destination vertexes. Final cost of the path is $dist_f(q) + dist_b(q)$, where q is a meeting vertex, $dist_f$ is the forward distance and $dist_b$ is the backward distance. Searches can meet in more vertexes, where in that case one with the minimal sum of distances is chosen. The Main advantage of this algorithm is its exactitude [3].

2.3 Algorithm combination

Other approach is a combination of these methods such as A* combined with Bidirectional Dijkstra.

2.3.1 Bidirectional A*

We can use A* cost (ca) for Bidirectional Dijkstra's algorithm. This step can significantly reduce the overall search space, but the real advantage of this property is hard to predicate (this is demonstrated later in the section focused on tests).

Fig.1 depicts 5 variants of the search space for each method. In some scenarios it might be also beneficial to use the edge cost for the forward and backward search differently. These methods are:

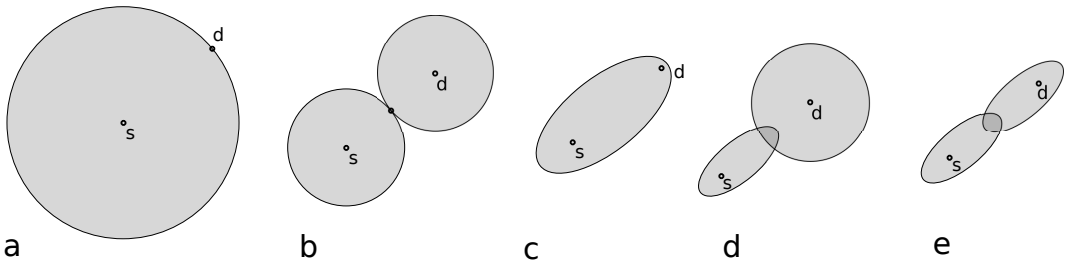

Figure 1: Search space a) Dijkstra b) Bidirectional c)A* d) Bidirectional Dij. and A* e)Bidirectional A*

a) Dijksta's algorithm

b) Bidirectional Dijkstra's algorithm (Bi-Dijkstra)

c) A* algorithm

d) Bidirectional A* with A* cost for forward search and with Dijkstra's cost for backward cost (Bi-A*-Dij)

e) Bidirectional A* with both searches having A* cost (Bi-A*)

2.4 Restricted areas constraint

The given task requires the searched path to avoid restricted areas. Restricted areas have the form of polygons (including non-convex polygons). The mentioned search algorithms were customized in order to test whether or not a particular edge is located in the restricted zone. More concretely the *relaxation* function was extended to evaluate if a particular edge intersects a restricted area (by calling *intersectsRS* function in 2). This approach enables to evaluate just the edges that are traversed during a particular shortest path search.

Algorithm 2 : Relaxation edges with passable test

```
1: for ∀ neighbours j do
2:    if intersectsRS((i,j)) then
3:       continue
4:    end if
5:    if dist(i) + c(i, j) < dist(j) then
6:       dist(j) = dist(i) + c(i, j)
7:       prev(j) = i
8:       Q = Q ∪ j
9:    end if
10: end for
```

Another possibility that might be considered is to delete the edges that intersect the restricted zones before the search algorithm starts. In this case the evaluation of all the edges of the graph will be required even-though for particular search space it might be not necessary (there might be even no restricted zones between source and destination vertex).

The choice of an appropriate approach (in the sense of algorithm speed) depends on concrete properties of the given graph, restricted zones positions as well as the number of searches and location of source and destination vertexes. If there is only one or a few shortest path searches performed on the graph (such as given in the task), the approach based on the extended relax function seems to be an appropriate option.

2.4.1 Algorithm for *intersectsRS* function

Algorithm 3 evaluates if an edge crosses a restricted area. The first step of the algorithm tests if a bounding box of the edge and the restricted area's bounding box intersect. If this is true, the intersection of the entire geometries is tested. For the test of bounding boxes the precomputed bounding boxes that are contained in the Shapefile format (format of the provided input data) are utilized.

Algorithm 3 : *intersectsRS*

```
1: e is input edge
2: for ∀ restricted polygons polyg do
3:    if bbox(e) intersect with bbox(polyg) then
4:       if a intersect with polyg then
5:          return true
6:       end if
7:    end if
8:    return false
9: end for
```

The complexity of the *intersectsRS* algorithm is linear with respect to the number of given restricted areas. In that sense it is obviously not the most optimal solution for a large number of restricted zones. In such a case it would be preferable to use an appropriate spatial data structure (e.g. R-tree), however for a small number of polygons the described solution might be sufficient.

2.5 Implementation and tests

2.5.1 Implementation in C

We implemented the described solution in the C programming language. For the graph adjacency a list representation is used. Fig. 2 depicts a schema of the graph representation. The graph vertexes are stored as an array. For an efficient searching for a vertex by its *id*, there is an index array, where the *id* is used as an array's index and the item value is a pointer to the vertex array. If the range of vertexes *ids* is too wide, we get a sparse array, that might exceed the primary memory. In that case there is a fallback solution and the hash table is used. The vertex structure contains the pointer to a linked list of vertex's edges having this vertex as a source (red arrows in Fig. 2) or as a destination (blue arrows in Fig. 2).

2.5.2 Tests

Tests where performed on a computer with Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz and 8 GB RAM. Using gcc 4.9.2 and optimization was set to "-O3".

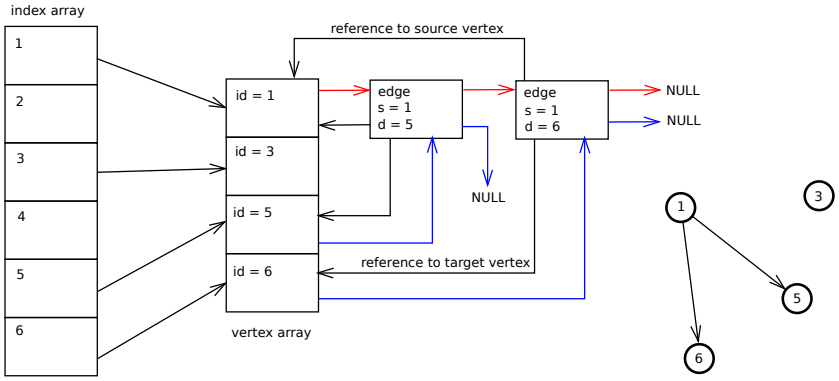

Figure 2: Graph implementation in C

For every of the above mentioned methods (Dijkstra, Bi-Dijkstra, A*, Bi-A*-Dij, Bi-A*) we computed 100 shortest path searches. Graph has 67 000 edges (m), 29 000 vertexes (n) and 5 restricted zones (every zone contains about 5 edges). Uniformly distributed source and destination vertexes were generated and two configurations were evaluated. In the first case, the edge length as a value for the edge cost was used (searching for real shortest path). In second case the edge travel time as the edge cost value was used (searching for fastest path). We tested all the methods, while for the Bidirectional A* method we use:

Method	forward k	backward k
Bi-A*	0.5	0.5
Bi-A*-Dij	0.5	0

Method	execution time [μs]	maximum error [%]	average error [%]
Dijkstra	8747	0.0	0.0
Bi-Dijkstra	7327	0.0	0.0
A*	3478	25.3	9.4
Bi-A*-Dij	3462	24.3	8.3
Bi-A*	2797	28.3	10.1

Table 1: Result of the shortest path search

Table 1 depicts the results of the test, where the edge length has been used as an edge cost. The first column is the execution time in microseconds. The second column is the maximal error (ratio between correct length and calculated length). The third is the average error. Execution time corresponds with the size of a search space. Table 2 depicts the same test, where the edge travel time has been used as an edge cost. Overall errors are less for the length-derived cost, because the potential (p) is based on an air distance and therefore more helpful in this scenario. For the time-derived edge cost the errors are much higher.

3. CONCLUSIONS

We customized state-of-the-art techniques focused on the shortest path search to obey restricted zones. We made several implementation considerations based on the size of the network, the number of restricted areas as well as expected number of given queries. By experiments we evaluate these design choices and select the one featuring the best performance while keeping the found path exact. All the algorithms were implemented in C language. The fastest method is Bi-A* and the slowest is Dijkstra. Algorithm's speed in agreement with theoretical expectations. The solution based on Bidirectional Dijkstra's algorithm was delivered for the ACM SIGSPATIAL Cup 2015 as the fastest exact method.

Method	execution time [μs]	maximum error [%]	average error [%]
Dijkstra	8961	0.0	0.0
Bi-Dijkstra	6800	0.0	0.0
A*	2995	40.0	19.8
Bi-A*-Dij	3017	39.4	18.5
Bi-A*	2386	40.0	22.2

Table 2: Result of the fastest path search

4. ACKNOWLEDGMENTS

We would like give thanks to the ACM SIGSPATIAL Cup 2015 organizers for the interesting competition. František Kolovský was supported by the OpenTransportNet project (620533) co-funded by the Union's ICT Policy Support Programme as part of the Competitiveness and Innovation Framework Programme. Jan Ježek was supported by the LO1506 project of the Czech Ministry of Education, Youth and Sports.

5. REFERENCES

[1] Acm sigspatial cup 2015. `http://research.csc.ncsu.edu/stac/GISCUP2015/index.php`. Accessed: 2015-10-05.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[3] A. V. Goldberg. Point-to-point shortest path algorithms with preprocessing. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 88–102. Springer, 2007.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[5] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In *STACS 2007*, pages 23–36. Springer, 2007.