An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags.

Ulrich Lauther
Siemens AG, CT SE 6
Otto-Hahn Ring 6, 81730 München, Germany
ulrich.lauther@siemens.com

November 3, 2006

Abstract

We present an efficient algorithm for fast and exact calculation of shortest paths in graphs with geometrical information in nodes (coordinates), e.g. road networks. The method is based on preprocessing and therefore best suited for static graphs, i.e., graphs with fixed topology and edge costs.

In the preprocessing phase, the network is divided into regions and edge flags are calculated that indicate whether an edge belongs to a shortest path into a given region. In the path calculation step, only those edges need to be investigated that carry the appropriate flag.

We compared this method to a classical Dijksta implementation using USA road networks with travel times and report on speedup, preprocessing time, and memory needed to store edge flags.

1 Introduction

A huge amount of research work has been done concerning fast shortest path algorithms, but still efficient exact algorithms are based on Dijkstra's algorithm [1], combined with efficient data structures for implementing the priority queue [2, 3, 4], which is a central part of an efficent implementation.

Given a directed or undirected graph $G = (V, E)$ with n nodes V and m edges E, edges $e = (v, w)$, positive edge weights $w(e)$, and two special nodes s (source) and t (sink), calculating a shortest path (i.e., a path with minimum total edge weight) between s and t can be solved efficiently using Dijkstra's algorithm. Depending on data structures used and additional assumptions on the density of the graph and the distribution of edge weights, the worst case complexity lies between $O(n^2)$ (for dense graphs) and $O(m)$ (for a sparse graph with limited, integral edge weights [2]).

In a typical geographical application, e.g., finding a shortest path in a road map, the expected runtime of a carefull implementation will be $O(d^2 \log d)$ if d is the number of edges in the shortest path; as we observe a circular expansion of the algorithm around the source node, $O(d^2)$ nodes will be touched and each one will be inserted into and retrieved from a priority queue with size $O(d^2)$ once. (Updates of the priority queue are of the same order, as the node degree in road networks is limited and small). "Carefull implementation" implies for instance that - other than usually seen in textbooks - an initialization phase processing all nodes in the network is avoided.

Often these runtimes are prohibitive, e.g., in autonomous automotive navigation systems with limited resources (memory and CPU power). In practice, some speed-up is often achieved by compromising accuracy: heuristics are employed that limit the search space and hopefully find reasonable solutions. However, the latter is not guaranteed and not always achieved in practice. Typical examples of such heuristics are the layer concept found in autonomous car navigation systems and heuristic variants of the A^*-algorithm.

Much less effort has been spent on solutions which use preprocessing, i.e., trade off-line preprocessing time for runtime in the target application. Of course, preprocessing is not trivial: we cannot afford to calculate (and store!) all shortest paths that might be requested in the application.

This paper describes a very successful preprocessing strategy and its efficient implementation. Here, *successful* means a big speed-up in the target application and acceptable memory needs, *efficient* implies acceptable preprocessing runtime. Moreover, the resulting algorithms are exact, i.e. the shortest (cheapest) path is guaranteed to be found.

At least in our implementation, the suggested method needs not just an abstract graph, but also some geometrical information: coordinates of nodes. In addition, we assume positive length (or cost) of edges. No further assumptions (e.g., planarity) are made.

2 Previous Work

The work reported here was done in the late nineties already, but published at that time only in the form of patent applications [5, 6]. A scientific publication was made available in 2004 [7], but contained quite limited experimental results.

Other preprocessing approaches include Ertl's work [8] using edge radii, the work by Goldberg and Harrelson [9], who use so called *land marks* that give upper bounds for the travel time to the target node and can be used in A^* search, Wagner et. al.'s [10] work on using geometric containers that enclose subtrees of shortest paths trees to reduce the search space, and finally hierarchical methods [11, 12] that are similar to the heuristics currently used in production navigation systems, but - unlike these heuristics - guarantee optimum path calculations.

To the best of my knowledge, all but one of these approaches are - in comparison to the method described here - inferior in terms of speed-up achieved and/or memory needed to store results of preprocessing. The only competitive method seems to be the hierarchical approach presented by Sanders and Schultes [12], but that method needs major modifications of the shortest path implementation in the target that has to deal with the hierarchy imposed on the network.

3 Basic Idea: Edge Flags

When we drive through a road network in real life, we usually do not calculate shortest paths at all; we follow signposts. These do not need to be very specific: driving from Munich to Hamburg we do not (initially) care whether we are heading for the Binnenalster or the Reeperbahn, we just follow the signs to Hamburg.

This principle can be transferred to a software solution.

Let the road map be represented by an undirected graph $G = (V, E)$ with n nodes and m edges. Edges carry two weights (e.g., length or estimated travel time), one for the forward direction, one for the backward direction. (A one-way street would have an infinite cost in one direction).

We first partition the whole network into r regions i. These regions should be geometrically connected, but the only hard requirement is that each node v of the network belongs to exactly one region, and that (in the application) we have fast access to a node's region (for instance by way of a region-id per node). We then define two edge flags for each edge $e = (v, w)$ and region i, $vflag_{e,i}$ and $wflag_{e,i}$. Let us (for now) assume that node v does not belong to region i. Then edge flag $vflag_{e,i}$ is set iff there is a shortest path from node v over edge e into region i. $wflag_{e,i}$ is defined in a similar way. Thus we have $m * r * 2$ different flags, and we need 1 bit to store each of these flags. These flags are our road signs; how they can be calculated will be discussed later. The generation of regions is not discusse in this paper, see [7] for details.

4 How to Use Edge Flags

Utilization of edge flags can be added to any existing shortest path algorithm; below is the Dijkstra algorithm with edge flag enhancements printed in bold face. We use pseudo-code, which is very near to the actual implementation based on our C++ class library TURBO [13]. So we need just two additional lines, one to retrieve the region index of the target node, the other one for skipping edges which cannot be on the shortest path to the target. This simple trick gives an enormous reduction of nodes that need to be scanned, and thus a corresponding speed-up for the shortest path calculation.

(Note than in a real application the initialization of all nodes would be done outside the Dijkstra-procedure only once; inside the procedure, we would keep track of touched nodes and re-initialize only these before the procedure returns.)

5 The Problem of Cones

Coming back to our previous example, as we approach Hamburg we need to make up our mind, whether to head for Binnenalster or Reeperbahn. The signs pointing to Hamburg are not very helpful anymore. When we use edge flags as described above and animate the implementation, we observe that in the initial part of the route very few nodes are touched that do not belong to the final shortest path. This is so because usually only *one* edge out of a node v has its flag on for a far away region. When we are near the region, the situation changes. As edge flags describe shortest routes to *alls* nodes within the region, now many edges out of a node will have its flag set and we observe a broad cone of touched nodes in front of the target region.

```
int dijkstra(Graph& g, Node* source, Node* target) {

  int target_region = target->region();

  // initialize all nodes:
  Node* v;
  forall_nodes(v,g) v->dist = ∞;

  // initialize priority queue:
  priority_queue q;
  source->dist = 0;
  q.insert(source);

  while ((v = q.del_min())) { // while q not empty
    if (v == target) return v->dist;
    Node* w;
    Edge* e;
    forall_adj_edges(w,v,e,g) { // scan node v
      if (! flagged(v,e,target_region)) continue;
      if (w->dist > v->dist + e->length) {
        w->dist = v->dist + e->length;
        if (q.member(w)) q.update(w);
        else             q.insert(w);
      }
    }
  }
  return ∞;
} // dijkstra
```

Figure 1: Edge-flags-enhanced Dijkstra code.

This cone can easily be avoided: if we had started the shortest path calculation at the target, we would not have had a cone there, but one in front of the source region. We can combine the two approaches (and their benefits) by using a bidirectional shortest path algorithm, expanding from source and target simultaneously. The two expansions will meet somewhere in the middle between source and target and hopefully before the cones start to develop. This is at least true when source and target are far from each other; if they are near, we do not have a problem with computation time in the first place.

There is a small problem with this approach: when we expand from the target, we have to account for the fact that we are driving (one-way) roads

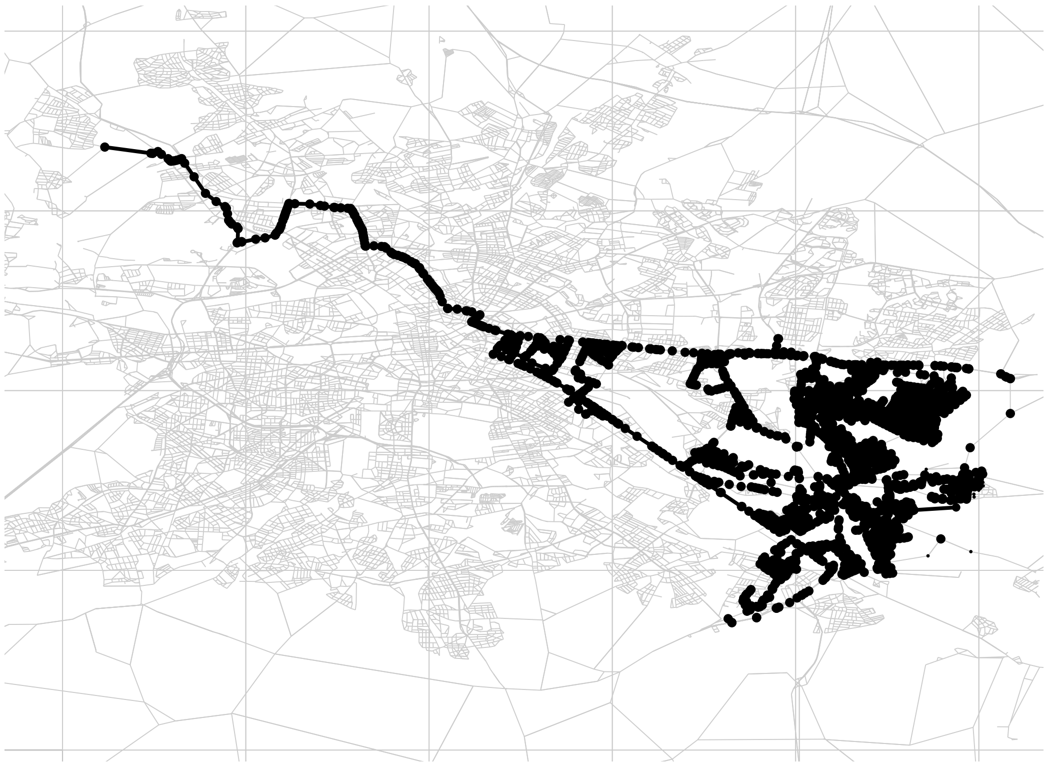

Figure 2: Cone in Front of the Target Region

in the wrong direction. To compensate for this, we have to use the ”wrong” cost value for each edge traversed. However, the edge flags as defined above have not been calculated for this situation. As a consequence, we need *four* flags per bidirectional edge and region, two for the nominal driving direction and two for the wrong one.

We also have to make sure that the two paths developing from source and target meet in a common node. To guarantee this, we have to make sure that *all* edges (v,w) on a shortest path from v into region i are flagged, not just one out of v (which would be sufficient in a unidirectional implementation).

A further complication arises, when we are inside the target region. For a thorough discussion of this special case see [7].

6 How to Calculate Edge Flags

The key question is, of course, how to calculate the edge flags in an efficient way. This is where the problem gets interesting (or where the fun begins). In this paper, we will only give a rough idea, without going into all details.

6.1 A Simple but dead slow method

The simplest method is as follows: for each node v in a region i we calculate a tree of shortest paths, with node v as its root. We can use Dijkstra's algorithm for this purpose. When a tree has been calculated, each node carries a distance label, which gives the node's distance to the root. Then we look at all edges $e = (v, w)$: if the difference between the two distances $distance(v)$ and $distance(w)$ corresponds to the length (or cost) of e, this edge is in some shortest path out of region i and we can set the corresponding edge flag. Flags from different trees within the same region are or-ed together. (Actually we are interested in shortest paths *into* region i, not out of it; we fix this be using the "wrong" edge costs). Note that not only edges which form the shortest-path tree are flagged, but all edges without "slack"; thus we have to inspect *all* edges after each tree calculation. If we have done all these tree calculations (two for each node, as we need four flags, see above) and edge inspections (about 2 weeks later for a road map of Germany containing 1.2 million edges) we can write the collected flags to disk and are done.

6.2 A somewhat faster method

We can classify the edges into two types: *internal* edges and *interface* edges. Internal edges are those that connect two nodes which belong to the same region. Interface edges connect nodes that belong to different (usually geometrically adjacent) regions. Based on the classification of edges we can classify nodes into *interior* nodes and *exported* nodes. Exported nodes are nodes which are incident to at least one interface edge. All others are interior nodes. Obviously, when we try to reach a target node in some region i (the target region) from the outside, we have to traverse some interface edge and thus cross an exported node of that region. Therefore, it is sufficient to apply the tree calculation process just discussed to exported nodes. After nodes have been assigned to regions (see [7] for details), it is easy to identify exported nodes and to store them in a list for each region. We just look at all nodes of a region and check outgoing edges; if the edge connects into another region, both nodes can be marked as exported nodes and put into the respective list. If we now apply our flag calculation method to exported nodes only, we can cut down runtime for the mentioned data set from 20 weeks (estimated) to about 35 hours. (Running times in this paragraph were measured on a somewhat outdates 1 GHz Pentium III).

6.3 A fast but wrong method

One might be tempted to suggest the following clever way to achieve an even faster solution: Instead expanding from one exported node of a region at a time, expand from all of them at the same time. This is easy to implement: we initialize the priority queue of Dijkstra's algorithm with all the exported nodes, after setting their distance to zero. Then we keep expanding nodes until the queue is empty. This is equivalent to expanding from some virtual inner node of the region.

This method is fast, but unfortunately wrong. Resulting flags do not reflect shortest paths to *any* interior node of a region (or equivalently to *any* exported node), but to the *nearest* one. Thus, in the application, we would generate paths that pass thru the exported node of the target region that is nearest to the source node and from there run along a shortest path to the target node. This is not necessarily a shortest path between source and target.

6.4 The fast way

An other obvious attempt to achieve faster tree calculations uses the observation that trees rooted at neighboring exported nodes will usually show much similarity. A method that exploits this fact has been developed and gives indeed another big speed-up. However, details of this procedure are out of the scope of this paper.

7 Saving space

As described, we would need $r * 4$ bits per edge when we use r regions. Some of these flags are redundant (in case of one way streets). For some edges all flags are zero and need not be stored individually. And different edges may share the same set of flags. Using these observations we can greatly reduce the space needed for edge flags.

8 Experimental Results

Experiments have been made with road networks provided by the 9th DIMACS challenge. We report on runtimes for preprocessing, speed-up, and efficiency (see definition below) for point-to-point queries, and memory needs for storing edge flags. For comparison we use a straight forward Dijkstra implementation using a radix-queue [4] for maintaining labeled but not yet scanned nodes.

The first table compares the runtimes and number of touched nodes of our own and the DIMACS-provided implementation for road networks with travel times and random point-to-point pairs as generated by the provided software. Runtimes are given in milliseconds/path and were measured - as all other results reported in this paragraph - on an AMD Opteron Processor 252 with 2.6 GHz running in 32-bit mode under Linux. We used the GNU-compiler gcc version 4.0.2 with optimization flag -O4.

	nodes	edges	runtime		own/D	touched		own/D
			DIMACS	own		DIMACS	own	
NY	264346	733846	36.1	69.8	1.9	132293.5	133172.4	1.0
BAY	321270	800172	45.5	84.1	1.9	164933.4	165576.0	1.0
COL	435666	1057066	66.5	110.0	1.7	218853.0	219496.0	1.0
FLA	1070376	2712798	151.6	260.1	1.7	530726.8	531580.2	1.0
NW	1207944	2840206	196.8	318.3	1.6	604467.0	605707.3	1.0
NE	1524452	3897634	252.1	394.3	1.6	785022.8	786515.6	1.0
CAL	1890814	4657740	285.4	494.8	1.7	937315.4	938574.5	1.0
LKS	2758118	6885656	450.2	764.8	1.7	1405326.0	1406694.2	1.0
E	3598622	8778112	644.8	1010.6	1.6	1858975.8	1860879.9	1.0
W	6262103	15248144	1324.8	1923.1	1.5	3141784.9	3144926.7	1.0

Table 1: Comparison of conventional Point-to-Point implementations, using networks with travel times

We see that our Dijkstra-implemetation is by a factor of 1.5 to 1.9 slower than the DIMACS-provided software - possibly due to our list based data structure used for storing and traversing the graphs. As the implementation using edge flags uses the same graph data structure, we use our own Dijkstra-implementation for comparisons in what follows.

8.1 Preprocessing

The following table shows preprocessing time and memory needs:

Instance	nodes	edges	run time	memory
BAY	21270	800172	136.9	6.1
NY	264346	733846	121.7	7.6
COL	435666	1057066	199.7	6.3
FLA	1070376	2712798	288.8	4.9
NW	1207944	2840206	671.5	5.9
NE	1524452	3897634	798.8	5.7
CAL	1890814	4657740	1119.0	5.4
LKS	2758118	6885656	1578.3	5.5
E	3598622	8778112	2090.0	5.2
W	6262103	15248144	6223.4	5.3

Table 2: Running times and memory needs for preprocessing

Runing times are given in CPU-seconds, memory is the space needed per edge in Bytes which includes a 4-Byte pointer to a bit string and the average size of the bit string itself. This is the additional memory needed in the application, resulting from preprocessing; memory needs *during* preprocessing are much higher.

The number of regions was set to 200 for all runs.

The graph in Fig. 2 shows how the running time increases with the number of nodes.

8.2 Random Point Pairs

In this paragraph we show results for the DIMACS-provided 1000 random point pairs per network (with travel times as edge weights).

For the evaluation of preprocessing methods, not only the speed-up achieved is interesting, but also the total cost (in terms of runtime) for preprocessing plus path calculations. Preprocessing pays off when the sum of preprocessing time t_{pre} and production time $n\ t_{\text{fast}}$ for calculating n shortest paths is lower than that of just using the slow algorithm $n\ t_{\text{slow}}$. The break even point is achieved for

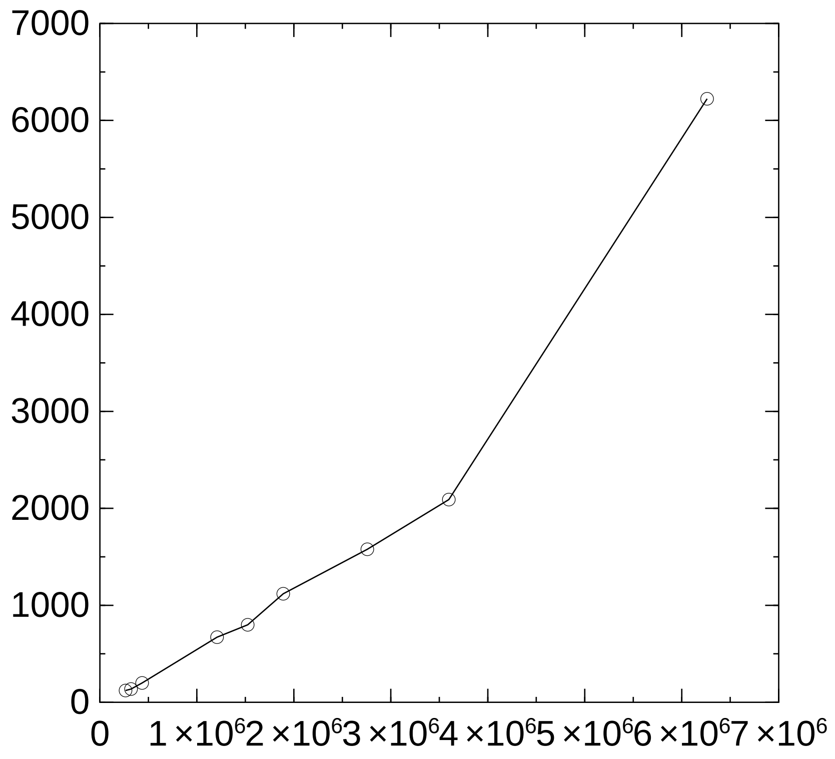

Figure 3: Running times for preprocessing as a function of number of nodes

$$n = \frac{t_{pre}}{t_{\text{slow}} - t_{\text{fast}}}$$

or

$$n \approx \frac{t_{\text{pre}}}{t_{\text{slow}}} \qquad \text{for} \qquad t_{\text{fast}} \ll t_{\text{slow}}$$

(This assumes that preprocessing and shortest path calculations are done on the same host. If the target host is slower, break even is reached earlier).

Another interesting measure is the efficiency as defined in [9], the number of nodes on the shortest path divided by the number of nodes scanned, given in percent. This number would ideally be 100, and actually we achieve this value for long paths.

The following table shows running times measured for our own plain vanilla Dijkstra-implementation and those achieved using edge flags, the speed-up resulting from preprocessing, the break-even path number, and the efficiency in percent.

Instance	nodes	pre	slow	fast	slow/fast	pre/slow	eff
NY	264346	121.7	69.8	0.293	238.3	1742.6	67.5
BAY	321270	136.9	84.1	0.374	225.0	1627.5	65.8
COL	435666	199.7	110.0	0.721	152.6	1815.5	59.4
FLA	1070376	288.8	260.1	1.421	183.1	1110.3	38.9
NW	1207944	671.5	318.3	1.151	276.6	2109.6	60.6
NE	1524452	798.8	394.3	1.086	363.1	2026.0	54.2
CAL	1890814	1119.0	494.8	1.695	291.9	2261.4	45.7
LKS	2758118	1578.3	764.8	1.952	391.8	2063.6	62.1
E	3598622	2090.0	1010.6	2.929	345.0	2068.0	37.6
W	6262103	6223.4	1923.1	3.679	522.7	3236.1	50.7

Table 3: Preprocessing time [sec] and average path calculation times [msec] with (fast) / without (slow) preprocessing for roadmaps with travel times.

Preprocessing time is given in seconds, path calculation times in milliseconds per path.

We see a speedup factor between about 150 to 500, a break-even number around 2000, and an efficieny around 50%.

The plot of Fig. 4 shows how path calculation time and path length are related. Path lengths are measured as Dijkstra rank of the target node, which is equal to the number of scanned nodes when the target node is about to be scanned in a plain vanilla Dijkstra implementation.

We see that most times are below 10 msec (the average is 3.7 msec) and that higher times are needed for path length in the middle range. Short paths are trivially fast and long paths gain most from preprocessing.

8.3 Local Point Pairs

Next we show results for more local point to point path calculations.

Here we give also the number of touched nodes (per path) for the two versions.

The *locality* shown in the table is the logarithm to the basis 2 of the Dijkstra rank. As to be expected, the speed-up factor grows with the length

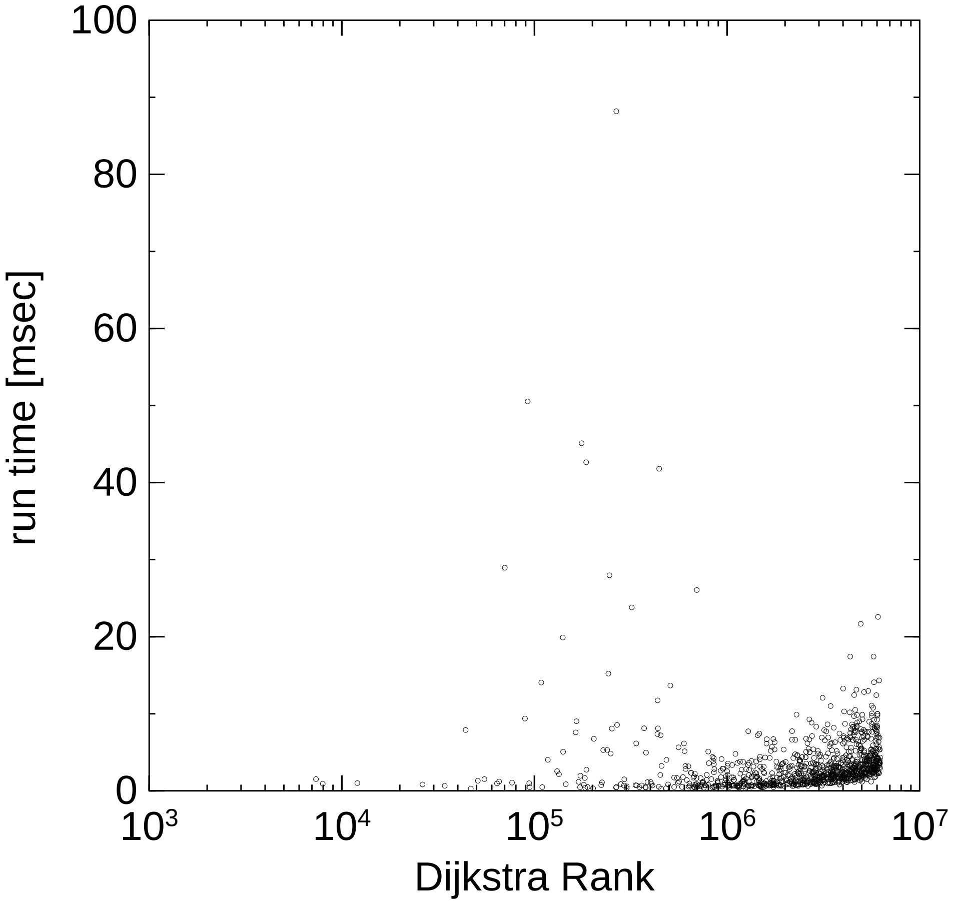

Figure 4: Path calculation times vs. Dijkstra rank for 1000 random node pairs in the Western USA roadmap with travel times

of paths calculated, with the exception of the most local paths where the edge flags help to keep the path calculations local to a region. Accordingly, the efficiency is low for local paths and gows up to 77.1% as paths get longer.

The speed-up factor more or less corresponds to the numbers of touched nodes in both cases.

Again we show a plot of calculation times versus path length, Fig. 5. Here too, the majority of calculation times are below 10 msec.

We may present the same data as a box plot (Fig. 6).

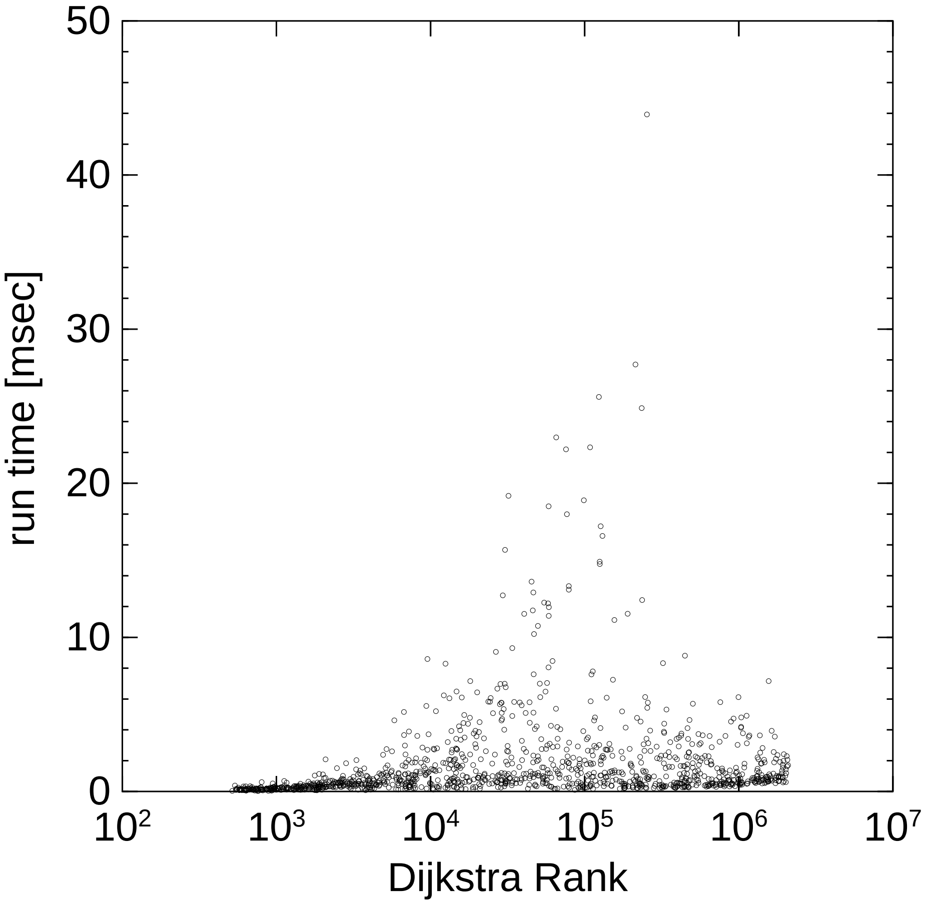

Figure 5: Path calculation times vs. Dijkstra rank for 100 random node pairs per locality range in the Great Lakes roadmap with travel times

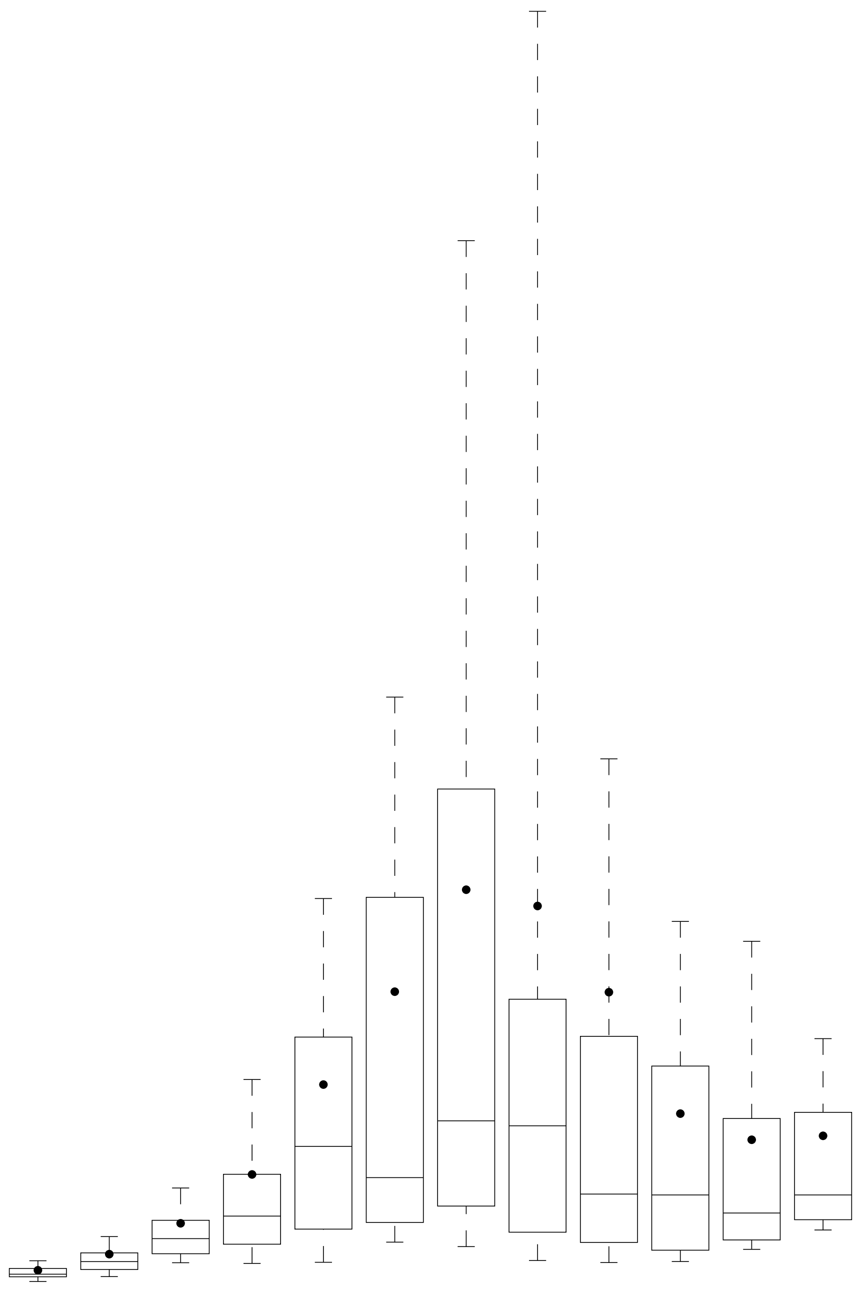

Figure 6: Box and whiskers plot of path calculation times vs. Dijkstra rank for 100 random node pairs per locality range in the Great Lakes roadmap with travel times, showing 10%, 25%, 75%, and 90% quantiles, median, and average running time (dot).

	running time			touched nodes			
locality	slow	fast	slow/fast	slow	fast	slow/fast	eff
9	2.370	0.230	10.3	831.7	484.1	1.7	8.5
10	2.740	0.810	3.4	1647.0	1720.4	1.0	6.3
11	3.319	0.420	7.9	3159.9	903.3	3.5	4.6
12	4.739	1.300	3.6	6431.4	2774.5	2.3	3.7
13	7.999	2.420	3.3	12881.1	4911.2	2.6	3.1
14	15.038	2.930	5.1	25281.0	6041.5	4.2	3.2
15	25.346	4.029	6.3	49702.7	8135.5	6.1	3.1
16	48.763	3.629	13.4	98637.0	6946.1	14.2	5.5
17	99.405	2.920	34.0	197561.3	5316.2	37.2	9.7
18	199.850	1.830	109.2	401114.0	3018.8	132.9	26.3
19	392.120	1.610	243.6	775489.8	2608.2	297.3	49.0
20	833.373	1.880	443.3	1542992.7	2771.9	556.6	77.1

Table 4: Average path calculation times [msec] and number of touched nodes with (fast) / without (slow) preprocessing for the Great Lakes roadmap with travel times tabulated by locality.

9 Application Scenarios

There are two main applications for our algorithm in the context of route planning:

Firstly, in autonomous car navigation systems, the preprocessing can be done once and results are stored on the system's CDROM. This way we can immensely speed up route calculations (including slow seeks on the CDROM). However, when we want to be able to react to traffic jams, we have to fall back to heuristic solutions.

Secondly, in a navigation system with client server architecture (a central route planning server with the cars as clients), we can do the preprocessing at the server in a cyclic fashion (say, every 15 minutes) based on the current traffic situation.

10 Availability

The concept of edge flags and associated algorithms are patent protected. Source code licenses are available.

11 Conclusions

Our preprocessing algorithm using edge flags has been evaluated using road networks from the 9th DIMACS challenge. We achieve high speed-ups with a memory overhead of about 6 Bytes per edge and a break-even point for amortization of preprocessing around 2000 path calculations.

In comparison to other methods, the modification needed in the target application is very small and the additional operation is just one bit-lookup per edge to be traversed, whereas container based algorithms need to answer a containment query between a geometric container and a node's coordinates and hierarchical methods need to use the hierarchy in the target application.

References

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

[2] R. Dial, F. Glover, D.K., Klingman, D.: A computational analysis of alternative algorithms for finding shortest path trees. Networks **9** (1979) 215–248

[3] Tarjan, R.E.: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics (1983)

[4] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall (1993)

[5] Lauther, U., Enders, R.: United states patent no. us 6,636,800 b1: Method and device for computer assisted graph preprocessing. (2003)

[6] Lauther, U., Enders, R.: Europaeische patentschrift ep 1 027 578 b1: Verfahren und anordnung zur rechnergestuetzten bearbeitung eines graphen. (1998)

[7] Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In M. Raubal, A. Sliwinski, W.K., ed.: GI-Tage 2004. Volume 22. (2004) 219 – 230

[8] Ertl, G.: Shortest path calculation in large road networks. OR Spectrum **20** (1998) 15–20

[9] Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* search meets graph theory. In: SODA. (2005) 156–165

[10] Wagner, D., Willhalm, T., Zaroliagis, C.D.: Geometric containers for efficient shortest-path computation. ACM Journal of Experimental Algorithms **10** (2005)

[11] Flinsenberg, I.: Route planning algorithms for car navigation. PhD thesis, Technische Universiteit Eindhoven (2004)

[12] Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: ESA. (2005) 568–579

[13] Lauther, U.: The C++ class library TURBO - a toolbox for discrete optimization. Software@Siemens (2000) 34–36