LA-UR-98-2249
Approved for public release; distribution is unlimited

A Computational Study of Routing Algorithms for Realistic Transportation Networks

Riko Jacob, Madhav Marathe and Kai Nagel

A Computational Study of Routing Algorithms for Realistic Transportation Networks

RIKO JACOB [1] MADHAV V. MARATHE [1] KAI NAGEL [1]

July 1, 1999

Abstract

We carry out an experimental analysis of a number of shortest path (routing) algorithms investigated in the context of the TRANSIMS (TRansportation ANalysis and SIMulation System) project. The main focus of the paper is to study how various heuristic as well as exact solutions and associated data structures affect the computational performance of the software developed for realistic transportation networks. For this purpose we have used a road network representing with high degree of resolution the Dallas Ft-Worth urban area.

We discuss and experimentally analyze various one-to-one shortest path algorithms. These include classical exact algorithms studied in the literature as well as heuristic solutions that are designed to take into account the geometric structure of the input instances.

Computational results are provided to empirically compare the efficiency of various algorithms. Our studies indicate that a modified Dijkstra's algorithm is computationally fast and an excellent candidate for use in various transportation planning applications as well as ITS related technologies.

Keywords: Experimental Analysis, Transportation Planning, Design and Analysis of Algorithms, Network Design, Shortest Paths Algorithms.

AMS 1991 Subject Classification: 68Q25, 68Q45, 90B06, 68R10

[1]Los Alamos National Laboratory, P.O. Box 1663, MS M997, Los Alamos, NM 87545. Email: {marathe, kai}@lanl.gov. Research supported by the Department of Energy under Contract W-7405-ENG-36.

[2]Part of the work done while at Los Alamos National Laboratory, Los Alamos, NM 87545. and supported by the Department of Energy under Contract W-7405-ENG-36. Current Address: BRICS, Department of Computer Science, University of Aarhus Ny Munkegade Bldg. 540, DK-8000 Århus C, Denmark. Email: rjacob@brics.dk.

[3]Preliminary version of the paper was presented at the 2nd Workshop on Algorithmic Engineering (WAE), Saarbrücken, Germany.

1 Introduction

TRANSIMS is a multi-year project at the Los Alamos National Laboratory and is funded by the Department of Transportation and by the Environmental Protection Agency. The main purpose of TRANSIMS is to develop new methods for studying transportation planning questions. A prototypical question considered in this context would be to study the economic and social impact of building a new freeway in a large metropolitan area. We refer the reader to [TR+95a] and the web-site `http://transims.tsasa.lanl.gov` to obtain extensive details about the TRANSIMS project.

The main goal of the paper is to describe the computational experiences in engineering various path finding algorithms in the context of TRANSIMS. Most of the algorithms discussed here are not new; they have been discussed in the Operations Research and Computer Science community. Although extensive research has been done on theoretical and experimental evaluation of shortest path algorithms, most of the empirical research has focused on randomly generated networks and special classes of networks such as grids. In contrast, not much work has been done to study the computational behavior of shortest path and related routing algorithms on realistic traffic networks. The realistic networks differ from random networks as well as from homogeneous (structured networks) in the following significant ways:

(i) Realistic networks typically have a very low average degree. In fact in our case the average degree of the network was around 2.6. Similar numbers have been reported in [ZN98]. In contrast random networks used in [Pa84] have in some cases average degree of up to 10.

(ii) Realistic networks are not very uniform. In fact, one typically sees one or two large clusters (downtown and neighboring areas) and then small clusters spread out throughout the entire area of interest.

(iii) For most empirical studies with random networks, the edge weights are chosen independently and uniformly at random from a given interval. In contrast, realistic networks typically have short links.

With the above reasons and specific application in mind, the main focus of this paper is to carry out experimental analysis of a number of shortest path algorithms on real transportation network and subject to practical constraints imposed by the overall system. See also Section 6, the point "Peculiarities of the network and its Effect" for some intuition what features of the network we consider crucial for our observations.

The rest of the report is organized as follows. Section 2 contains problem statement and related discussion. In Section 3, we discuss the various algorithms evaluated in this paper. Section 4 summarizes the results obtained. Section 5 describes our experimental setup. Section 6 describes the experimental results obtained. Section 7 contains a detailed discussion of our results. Finally, in Section 8 we give concluding remarks and directions for future research. We have also included an Appendix (Section 8.1) that describes the relevant algorithms for finding shortest paths in detail.

2 Problem specification and justification

The problems discussed above can be formally described as follows: let $G(V, E)$ be a (un)directed graph. Each edge $e \in E$ has one attribute $w(e)$ denoting the weight (or cost) of the edge e. Here, we assume that the weights are non-negative floating point numbers.

Definition 2.1 One-to-One Shortest Path:
Given a directed weighted, graph G, a source destination pair (s, d) find a shortest (with respect to w) path p in G from s to d.

Note that our experiments are carried out for shortest path between a pair of nodes, as against finding shortest path trees. Much of the literature on experimental analysis uses the second measure to gauge the efficiency. Our choice to consider the running time of the one-to-one shortest path computation is motivated by the following observations:

1. In our setting we need to compute shortest paths for roughly a million travelers. In highly detailed networks, most of these travelers have different starting points (for example, in Portland we have 1.5 million travelers and 200 000 possible starting locations). Thus, for any given starting location, we could re-use the tree computation only for about ten other travelers.

2. We wanted our algorithms to be extensible to take into account additional features/constraints imposed by the system. For example, each traveler typically has a different starting time for his/her trip. Since we use our algorithms for time dependent networks (networks in which edge weights vary with time), the shortest path tree will be different for each traveler. As a second example we need to find paths for travelers with individual mode choices in a multi-modal network. Formally, we are given a directed labeled, weighted, graph G representing a transportation network with the labels on edges representing the various modal attributes (e.g. a label t might represent a rail line). There the goal is to find shortest (simple) paths subject to certain labeling constraints on the set of feasible paths. In general, the criteria for path selection varies so much from traveler to traveler that the additional overhead for the "re-use" of information is unlikely to pay off.

3. The TRANSIMS framework allows us to use paths that are not necessarily optimal. This motivates investigation of very fast heuristic algorithms that obtain only near optimal paths (e.g. the modified A^* algorithm discussed here). For most of these heuristics, the idea is to bias a more focused search towards the destination – thus naturally motivating the study of one-one shortest path algorithms.

4. Finally, the networks we anticipate to deal with contain more than 80 000 nodes and around 120 000 edges. For such networks storing all shortest path trees amounts to huge memory overheads.

3 Choice of algorithms

Important objectives used to evaluate the performance of the algorithms include (i) time taken for computation on real networks, (ii) quality of solution obtained, (iii) ease of implementation and (iv) extensibility of the algorithm for solving other variants of the shortest path problem. A number of interesting engineering questions were encountered in the process. We experimentally evaluated a number of variants of Dijkstra's algorithm. The basic algorithm was chosen due to the recommendations made in Cherkassky, Goldberg and Radzik [CGR96] and Zhan and Noon [ZN98]. The algorithms studied were:

- Dijkstra's algorithm with Binary Heaps [CGR96],
- A^* algorithm proposed in AI literature and analyzed by Sedgewick and Vitter [SV86],
- a modification of the A^* algorithm that we will describe below, and alluded to in [SV86].

A bidirectional version of Dijkstra's algorithm described in [Ma, LR89] and analyzed by [LR89] was also considered. We briefly recall the A^* algorithm and the modification proposed. Details of these algorithms can be found in the Appendix.

When the underlying network is (near) Euclidean it is possible to improve the average case performance of Dijkstra's algorithm by exploiting the inherent geometric information that is ignored by the classical path finding algorithms. The basic idea behind improving the performance of Dijkstra's algorithm is from [SV86, HNR68] and can be described as follows. In order to build a shortest path from s to t, we use the original distance estimate for the fringe vertex such as x, i.e. from s to x (as before) *plus* the Euclidean distance from x to t. Thus we use global information about the graph to guide our search for shortest path from s to t. The resulting algorithm runs much faster than Dijkstra's algorithm on typical graphs for the following intuitive reasons: (i) The shortest path tree grows in the direction of t and (ii) The search of the shortest path can be terminated as soon as t is added to the shortest path tree.

We note that the above algorithms, only require that the Euclidean distance between any two nodes is a valid lower bound on the actual shortest distance between these nodes. This is typically the case for road networks; the link distance between two nodes in a road network typically accounts for curves, bridges, etc. and is at least the Euclidean distance between the two nodes. Moreover in the context of TRANSIMS, we need to find fastest paths, i.e. the cost function used to calculate shortest paths is the time taken to traverse the link. Such calculations need an upper bound on the maximum allowable speed. To adequately account for all these inaccuracies, we determine an appropriate lower bound factor between Euclidean distance and assumed delay on a link in a preprocessing step.

We can now modify this algorithm by giving an appropriate weight to the distance from x to t. By choosing an appropriate multiplicative factor, we can increase the contribution of the second component in calculating the label of a vertex. From a intuitive standpoint this corresponds to giving the destination a high potential, in effect biasing the search towards the destination. This modification will in general **not** yield shortest paths, nevertheless our experimental results suggest that the errors produced can be kept reasonably small.

4 Summary of Results

We are now ready to summarize the main results and conclusions of this paper. As already stated the *main focus* of the paper is the engineering and tuning of well known shortest path algorithms in a practical setting. Another goal of this paper to provide reasons for and against certain implementations from a practical standpoint. We believe that our conclusions along with the earlier results in [ZN98, CGR96] provide practitioners a useful basis to select appropriate algorithms/implementations in the context of transportation networks. The general results/conclusions of this paper are summarized below.

1. We conclude that the simple Binary heap implementation of Dijkstra's algorithm is a good choice for finding optimal routes in real road transportation networks. Specifically, we found that certain types of data structure fine tuning did not significantly improve the performance of our implementation.

2. Our results suggest that heuristic solutions using the underlying geometric structure of the graphs are attractive candidates for future research. Our experimental results motivated the formulation and implementation of an extremely fast heuristic extension of the basic A^* algorithm. The parameterized time/quality trade-off the algorithm achieves in our setting appears to be quite promising.

3. Our study suggests that bidirectional variation of Dijkstra's algorithm is not suitable for transportation planning. Our conclusions are based on two factors: (i) the algorithm is not extensible to more general path problems and (ii) the running time does not outperform the other exact algorithms considered.

5 Experimental Setup and Methodology

In this section we describe the computational results of our implementations. In order to anchor research in realistic problems, TRANSIMS uses example cases called *Case studies* (See [CS97] for complete details). This allows us to test the effectiveness of our algorithms on real life data. The case study just concluded focused on Dallas Fort-Worth (DFW) Metropolitan area and was done in conjunction with Municipal Planning Organization (MPO) (known as North Central Texas Council of Governments (NCTCOG)). We generated trips for the whole DFW area for a 24 hour period. The input for each traveler has the following format: (starting time, starting location, ending location).[4] There are 10.3 million trips over 24 hours. The number of nodes

[4] This is roughly correct, the reality is more complicated, [NB97, CS97].

and links in the Dallas network is roughly 9863, 14750 respectively. The average degree of a node in the network was 2.6. We route all these trips through the so-called focused network. It has all freeway links, most major arterials, etc. Inside this network, there is an area where *all* streets, including local streets, are contained in the data base. This is the study area. We initially routed all trips between 5am and 10am, but only the trips which went through the study area were retained, resulting in approx. 300 000 trips. These 300 000 trips were re-planned over and over again in iteration with the micro-simulation(s). For more details, see, e.g., [NB97, CS97]. A 3% random sample of these trips were used for our computational experiments.

Preparing the network. The data received from DFW metro had a number of inadequacies from the point of view of performing the experimental analysis. These had to be corrected before carrying out the analysis. We mention a few important ones here. First, the network was found to have a number of disconnected components (small islands). We did not consider (o, d) pairs in different components. Second, a more serious problem from an algorithmic standpoint was the fact that for a number of links, the length was *less* than the actual Euclidean distance between the the two end points. In most cases, this was due to an artificial convention used by the DFW transportation planners (so-called centroid connectors always have length 10 m, whatever the Euclidean distance), but in some cases it pointed to data errors. In any case, this discrepancy disallows effective implementation of A^* type algorithms. For this reason we introduce the notion of the "normalized" network: For all links with length less than the Euclidean distance, we set the reported length to be equal to the Euclidean distance. Note here, that we take the Euclidean distance only as a lower bound on shortest path in the network. Recall that if we want to compute fastest path (in terms of time taken) instead of shortest, we also have to make assumptions regarding the maximum allowable speed in the network to determine a conservative lower bound on the minimal travel time between points in the network.

Preliminary experimental analysis was carried out for the following network modifications that could be helpful in improving the efficiency of our algorithms. These include: (i) Removing nodes with degrees less than 3: (Includes collapsing paths and also leaf nodes) (ii) Modifying nodes of degree 3: (Replace it by a triangle)

Hardware and Software Support. The experiments were performed on a Sun UltraSparc CPU with 250 MHz, running under Solaris 2.5. 2 gigabyte main memory were shared with 13 other CPUs; our own memory usage was always 150 MB or less. In general, we used the SUN Workshop CC compiler with optimization flag -fast. (We also performed an experiment on the influence of different optimization options without seeing significant differences.) The advantage of the multiprocessor machine was reproducibility of the results. This was due to the fact that the operating system does not typically need to interrupt a live process; requests by other processes were assigned to other CPUs.

Experimental Method 10,000 arbitrary plans were picked from the case study. We used the timing mechanism provided by the operating system with granularity .01 seconds (1 tick). Experiments were performed only if the system load did not exceed the number of available processors, i.e. processors were not shared. As long as this condition was not violated during the experiment, the running times were fairly consistent, usually within relative errors of 3%.

We used (a subset) of the following values measurable for a single or a specific number of computations to conclude the reported results

- (average) running time excluding i/o
- number of fringe/expanded nodes
- pictures of fringe/expanded nodes
- maximum heap size
- number of links and length of the path

Software Design We used the object oriented features as well as the templating mechanism of C++ to easily combine different implementations. We also used preprocessor directives and macros. As we do not want to introduce any unnecessary run time overhead, we avoid for example the concept of virtual inheritance. The software system has classes that encapsulate the following elements of the computation:

- network (extensibility and different levels of detail lead to a small, linear hierarchy)
- plans: (o, d) pairs and complete paths with time stamps
- priority queue (heap)
- labeling of the graph and using the priority queue
- storing the shortest path tree
- Dijkstra's algorithm

As expected, this approach leads to an apparent overhead of function calls. Nevertheless, the compiler optimization detects most such overheads. Specifically, an earlier non templated implementation achieved roughly the same performance as the corresponding instance of the templated version. The results were consistent with similar observations when working on mini-examples. The above explanation was also confirmed by the outcome of our experiments: We observed, that reducing the instruction count does not reduce the observed running time as might be expected. Assuming we would have a major overhead from high level constructs, we would expect to see a strong influence of the number of instructions executed on the running time observed.

6 Experimental Results

Design Issues about Data Structures We begin with the design decisions regarding the data structures used.

A number of alternative data structures were considered to investigate if they results in substantial improvement in the running time of the algorithm. The alternatives tested included the following. (i) Arrays versus Heaps , (ii) Deferred Update, (iii) Hash Tables for Storing Graphs, (iv) Smart Label Reset (v) Heap variations, and (vi) struct of arrays vs. array of structs. Appendix contains a more detailed discussion of these issues. We found, that indeed good programming practice, using common sense to avoid unnecessary computation and textbook knowledge on reasonable data structures are useful to get good running times. For the alternatives mentioned above, we did not find substantial improvement in the running time. More precisely, the differences we found were bigger than the unavoidable noise on a multi-user computing environment. Nevertheless, they were all below 10% relative difference. A brief discussion of various data structures tried can be found in the Appendix.

Analysis of results. The plain Dijkstra, using static delays calculated from reported free flow speeds, produced roughly 100 plans per second. Figure 1 illustrates the improvement obtained by the A^* modification. The numbers shown in the corner of the network snapshots tell an average (of 100 repetitions to destroy cache effects between subsequent runs) running time for this particular O-D-pair, in system ticks. It also gives the number of expanded and fringe nodes. Note that we have used different scales in order to clearly depict the set of expanded nodes. Overall we found that A^* on the normalized network (having removed network anomalies as explained above) is faster than basic Dijkstra's algorithm by roughly a factor of 2.

Modified A^* (Overdo Heuristic) Next consider the modified A^* algorithm – the heuristic is parameterized by the multiplicative factor used to weigh the Euclidean distance to the destination against the distance from the source in the already computed tree. We call this the *overdo* parameter. This approach, can be seen as changing the conservative lower bound used for in the A^* algorithm into an "expected" or "approximated" lower bound. Experimental evidence suggests that even large overdo factors usually yield reasonable paths. Note that this nice behavior might fail as soon as the link delays are not at all directly related to the link length (Euclidean distance between the endpoints), as might be expected in a network with link lengths proportional to travel times in a partially congested city. As a result it is natural to discuss the time/quality trade-off of the heuristic as a function of the *overdo* parameter. Figure 2 summarizes the performance. In the figure the X-axis represents the overdo factor, being varied from 0 to 100 in steps of 1. The Y-axis is used for multiple attributes which we explain below. First, the Y axis is used to represent the average running time per plan. For this attribute, we use the $\log$ scale with the unit denoting 10 milliseconds. As depicted by the solid line, the average time taken without any overdo at all is 12.9 milliseconds per plan. This represents the base measurement (without taking the geometric information into account, but including time taken for computing of Euclidean distances). Next, for overdo value of 10 and 99 the running times are respectively 2.53 and .308 milliseconds. On the other hand, the quality of the solution produced by the heuristic worsens as the overdo factor is increased. We used two quantities to measure the error — (i) the maximum relative error incurred over 10000 plans and (ii) the

ticks 2.40, #exp 6179, #fr 233

ticks 0.64, #exp 1446, #fr 316

Figure 1: Figure illustrating the number of expanded nodes while running (i) Dijkstra (ii) A^* algorithms. The figures clearly show the A^* heuristic is much more efficient in terms of the nodes it visits. In both the graphs, the path is outlined as a dark line. The fringe nodes and the expanded nodes are marked as dark spots. The underlying network is shown in light grey. The source node is marked with a big circle, the destination with a small one. Notice the different scales of the figures.

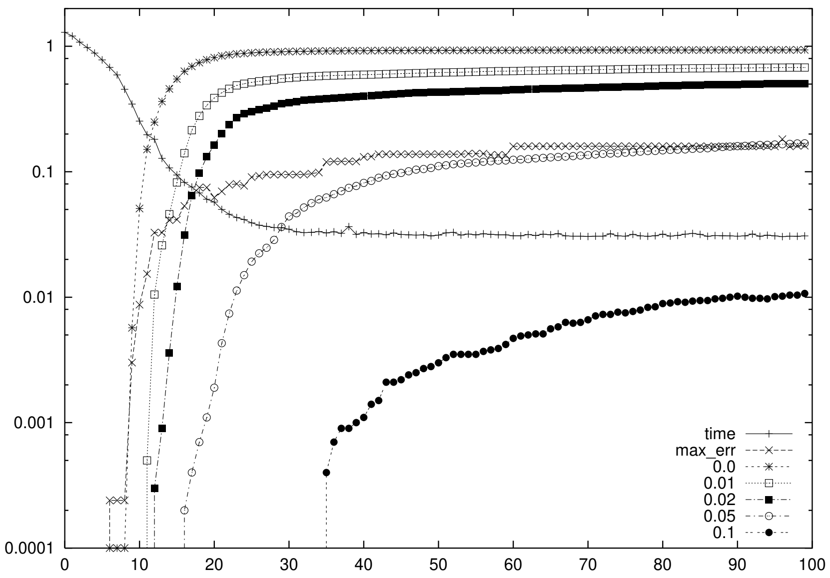

Figure 2: Figure illustrating the trade-off between the running time and quality of paths as a function of the overdo-paramameter. The X axis represents the overdo factor from 0 to 100. The Y axis is used to represent three quantities plotted on a log scale — (i) running time, (ii) Maximum relative error and (iii) fraction of plans with relative error greater than a threshold value. The threshold values chosen are 0%, 1%, 2%, 5%, 10%.

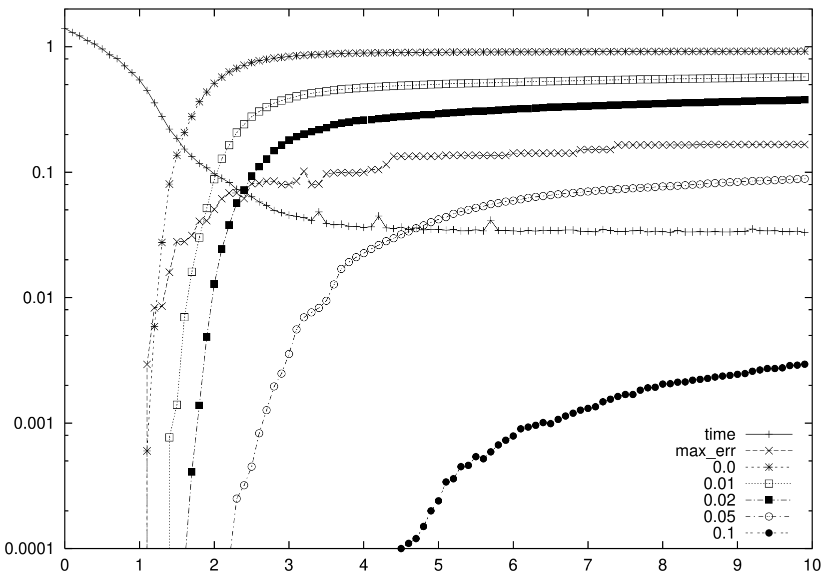

Figure 3: Figure illustrating the trade-off between the running time and quality of paths as a function of the overdo-parameter on the *normalized network*. The meaning of the axis and depicted things is the same as in the previous figure.

fraction of plans with errors more than a given threshold error. Both types of errors are shown on the Y axis. The *maximum relative error* (plot marked with *) ranges from 0 for overdo factor 0 to 16% for overdo value 99. For the other error measure, we plot one curve for each threshold error of 0%, 1%, 2%, 5%, 10%. The following conclusions can be drawn from our results.

1. The running times improve significantly as the overdo factor is increased. Specifically the improvements are a factor 5 for overdo parameter 10 and almost a factor 40 for overdo parameter 99.

2. In contrast, the quality of solution worsens much more slowly. Specifically, the maximum error is no worse than 16% for the maximum overdo factor. Moreover, although the number of erroneous plans is quite high (almost all plans are erroneous for overdo factor of 99), most of them have small relative errors. To illustrate this, note that only around 15% of them have relative error of 5% or more.

3. The experiments and the graphs suggest an "optimal" value of overdo factor for which the running time is significantly improved while the solution quality is not too bad. These experiments are a step in trying to find an empirical time/performance trade-off as a function of the overdo parameter.

4. As seen in Figure 3 the overall quality of the results shows a similar tradeoff if we switch to the normalized network. The only difference is that the errors are reduced for a given value of the overdo parameter.

5. As depicted in Figure 4, the number of plans worse than a certain relative error decreases (roughly) exponentially with this relative error. This characteristic does not depend on the overdo factor.

6. We also found that the near-optimal paths produced were visually acceptable and represented a feasible alternative route guiding mechanism. This method finds alternative paths that are quite different than ones found by the k-shortest path algorithms and seem more natural. Intuitively, the k-shortest path algorithms, find paths very similar to the overall shortest path, except for a few local changes.

7. The counterintuitive local maximum for overdo value 3.2 in Figure 3 can be explained by the example depicted in Figure 5.

   - the optimal length is 21,
   - for overdo parameter 2 we get a solution of length 22 Here node B gets inserted with a value of 24 opposed to values 29 and 25 of A and C. As these values for A and B are bigger than the resulting path using B, this path stays final.
   - for overdo parameter 4 we get again a solution of length 21. This stems from the fact that now C gets inserted into the heap with the value 33 where as B with a value of 40.

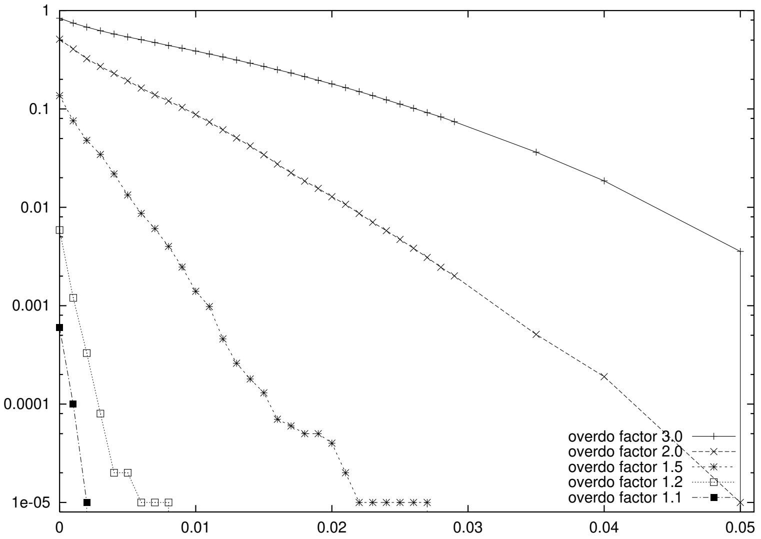

Figure 4: The distribution of wrong plans for different overdo-parameters in the normalized network for Dallas Ft-Worth. In X direction we change the "notion of a bad plan" in terms of relative error, in Y direction we show the fraction of plans that is classified to be "bad" with the current notion of "wrong".

> It is easy to see that such examples can be scaled and embedded into larger graphs. Since the maximum error stems from one particular shortest path question, it is not too surprising to encounter such a situation.

Peculiarities of the network and its Effect In the context of TRANSIMS, where we needed to find one-to-one shortest paths, we observed possibly interesting influence of the underlying network and its geometric structure on the performance of the algorithms. We expect similar characteristics to be visible in other road networks as well, possibly modified by the existence of rivers or other similar obstacles. Note that the network is almost Euclidean and (near) homogenous to justify the following intuition: Dijkstra's algorithm explores the nodes of a network in a circular fashion. During the run we see roughly a disc of expanded nodes and a small ring of fringe nodes (nodes in the heap) around them. For planar and near panar graphs it has been observed that the heap size is $O(\sqrt{n})$ with high probability. This provides one possible explanation of why the maximum heap sizes in our experiments was close to 500. In particular, even if the area of the circular (in number of nodes) reaches the size of the network (10 000), the ring of fringe nodes is roughly proportional to the circumference of the circular region (and thus roughly proportional to $\sqrt{10\,000}$). We believe that this homogenous and almost Euclidean structure is also the reason for our observations about the modified A^* algorithm. The above discussion provides at least an intuitive explanation of why special algorithms such as A^* might perform better on Euclidean and close to Euclidean networks.

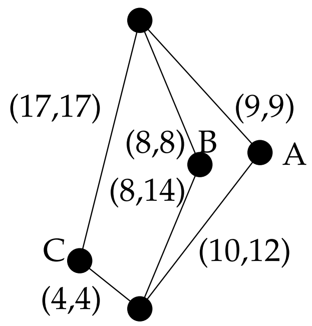

Figure 5: Example network having a local maximum for the computed path length for increasing overdo parameter; the edges are marked with (Euclidean distance, reported length).

Effect of Memory access times In our experiments we observed, that changes in the implementation of the priority queue have minimal influence on the overall running time. In contrast, the instruction count profiling (done with a program called "quantify") pinpoints the priority queue to be the main contributer to the overall number of instructions. Combining these two facts, we conclude that the running time we observe is heavily dependent on the time it takes to access the graph representation that do do not fit the cache. Thus the processor spends a significant amount of time waiting.

We expect further improvement of the running time by concentrating on the memory accesses, for example by making the graph representation more compact, or optimize accesses by choosing memory location of a node according to the topology of the graph. In general the conclusions of our paper motivate the need for design and analysis of algorithms that take the memory access latency into account.

7 Discussion of Results

First, we note that the running times for the plain Dijkstra are reasonable as well as sufficient in the context of the TRANSIMS project. Quantitatively, this means the following: TRANSIMS is run in iterations between the micro-simulation, and the planner modules, of which the shortest path finding routine is one part. We have recently begun research for the next case study project for TRANSIMS. This case study is going to be done in Portland, Oregon and was chosen to demonstrate the validate our ideas for multi-modal time dependent networks with public transportation following a scheduled movement. Our initial study suggests that we now take .5 sec/trip as opposed to .01 sec/trip in the Dallas Ft-Worth case [Ko98]. All these extensions are important from the standpoint of finding algorithms for realistic transportation routing problems. We comment on this in some detail below. Multi-modal networks are an integral part of most MPO's. Finding optimal (or near-optimal) routes in this environment therefore constitutes a real problem. In the past, solutions for routing in such networks was handled

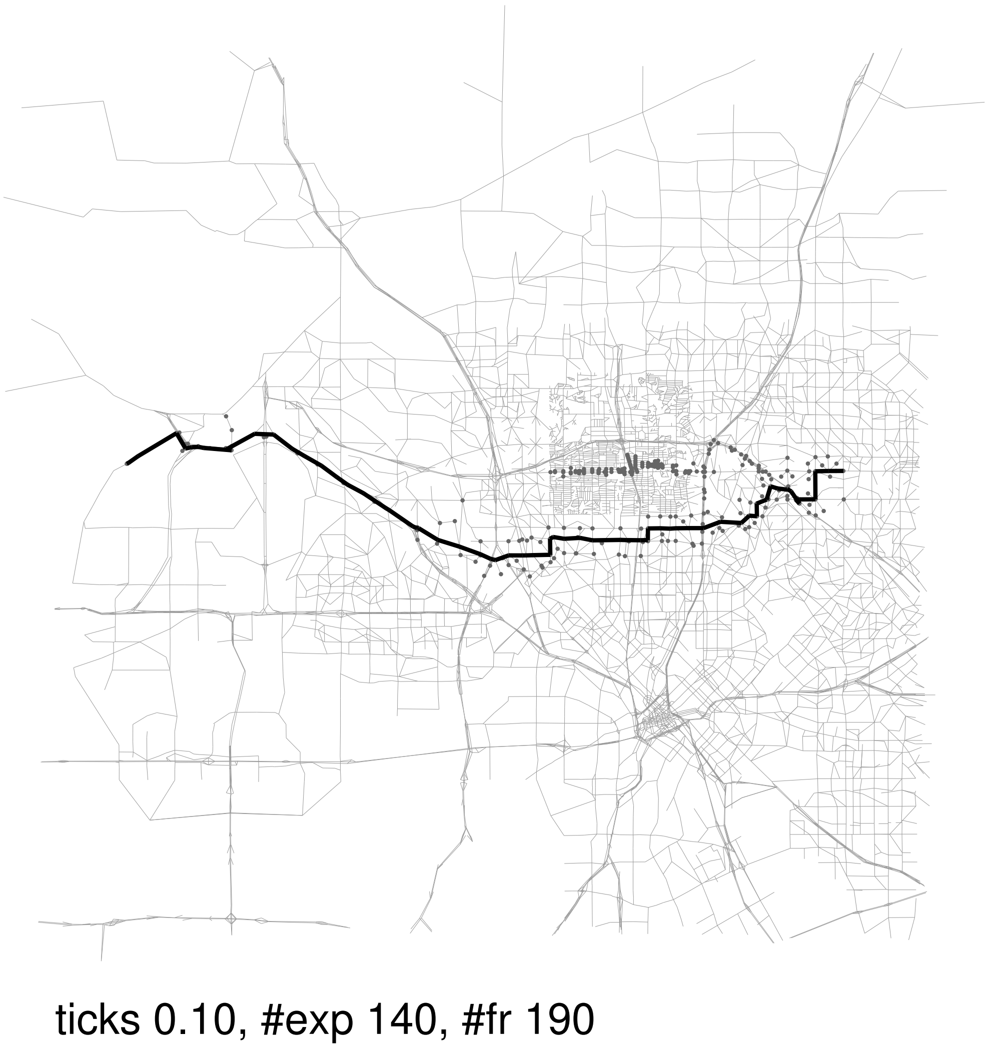

Figure 6: Figure illustrating two instances of Dijkstra's algorithms with a very high overdo parameter start at origin and destination respectively. One of them really creates the shown path, the beginning of the other path is visible as a "cloud" of expanded nodes

in an ad hoc fashion. The basic idea (discussed in detail in [BJM98]) here is to use regular expressions to specify modal constraints. In [BJM98, JBM98], we have proposed models and polynomial time algorithms to solve this and related problems. Next consider another important extension — namely to time dependent networks. We assume that the edge lengths are modeled by monotonic non-decreasing, piecewise linear functions. These are called the link traversal functions. For a function f associated with a link $e = (a, b)$, $f(x)$ denotes the time of arrival at b when starting at time x at a. By using an appropriate extension of the basic Dijkstra's algorithm, one can calculate optimal paths in such networks. Our preliminary results on these topics in the context of TRANSIMS can be found in [Ko98, JBM98]. The Portland network we are intending to use has about 120 000 links and about 80 000 nodes. Simulating 24 hours of traffic on this network will take about 24 hours computing time on our 14 CPU machine. There will be about 1.5 million trips on this network. Routing all these trips should take

$1.5 \cdot 10^6$ trips $\cdot\, 0.5$ sec/trip ≈ 9 days on a single CPU and thus less than 1 day on our 14 CPU machine. Since re-routing typically concerns only 10% of the population, we would need less than 3 hours of computing time for the re-routing part of one iteration, still significantly less than the micro-simulation needs.

Our results and the constraints placed by the functionality requirement of the overall system imply that bidirectional version of Dijkstra's algorithm is not a viable alternative. Two reasons for this are: (i) The algorithm can not be extended in a direct way to path problems in a multi-modal and time dependent networks, and (ii) the running times of A^* is better than the bidirectional variant; the modified A^* is much more faster.

8 Conclusions

The computational results presented in the previous sections demonstrate that Dijkstra's algorithm for finding shortest paths is a viable candidate for compute route plans in a route planning stage of a TRANSIMS like system. Thus such an algorithm should be considered even for ITS type projects in which we need to find routes by an on-board vehicle navigation systems.

The design of TRANSIMS lead us to consider one-to-one shortest path algorithms, as opposed to algorithms that construct the complete shortest-path tree from a given starting (or destination) point. As is well known, the worst-case complexity of one-to-one shortest path algorithms is the same as of one-to-all shortest path algorithms. Yet, in terms of our practical problem, this is not applicable. First, a one-to-one algorithm can stop as soon as the destination is reached, saving computer time especially when trips are short (which often is the case in our setting). Second, since our networks are roughly Euclidean, one can use this fact for heuristics that reduce computation time even more. The A* with an appropriate overdo parameter apperas to be an attractive candidate in this regard.

Making the algorithms time-dependent in all cases slowed down the computation by a factor of at most two. Since we are using a one-to-one approach, adding extensions that for example include personal preferences (e.g. mode choice) are straightforward; preliminary tests let us expect slow-downs by a factor of 30 to 50. This significant slowdown was caused by a number of factors including the following:

(i) The network size increased by a factor of 4 and was caused by addition and splitting of nodes and/or edges and adding public transportation. This was done to account for activity locations, parking locations, adding virtual links joining these locations, etc.

(ii) The time dependency functions used to represent transit schedules and varying speed of street traffic, implied increased memory and computational requirement. Initial estimates are that the memory requirement increases by a factor of 10 and the computational time increases by factor of 5. Moreover, different type of delay functions were used for inducing a qualitatively different exploration of the network by the algorithm. This seems to prohibit keeping a small number of representative time dependency functions.

(iii) The algorithm for handling modal constraints works by making multiple copies of the original network. The algorithm is discussed in [JBM98] and preliminary computational results are discussed in [Ko98]. This increased the memory requirement by a factor of 5 and computation time by an additional factor of 5.

Extrapolations of the results for the Portland case study show that, even with this slowdown the route planning part of TRANSIMS still uses significantly less computing time than the micro-simulation.

Finally, we note that under certain circumstances the one-to-one approach chosen in this paper may also be useful for ITS applications. This would be the case when customers would require customized route suggestions, so that re-using a shortest path tree from another calculation may no longer be possible.

Acknowledgments: Research supported by the Department of Energy under Contract W-7405-ENG-36. We thank the members of the TRANSIMS team in particular, Doug Anson, Chris Barrett, Richard Beckman, Roger Frye, Terence Kelly, Marcus Rickert, Myron Stein and Patrice Simon for providing the software infrastructure, pointers to related literature and numerous discussions on topics related to the subject. The second author wishes to thank Myron Stein for long discussion on related topics and for his earlier work that motivated this paper. We also thank Joseph Cheriyan, S.S. Ravi, Prabhakar Ragde, R. Ravi and Aravind Srinivasan for constructive comments and pointers to related literature. Finally, we thank the referees for helpful comments and suggestions.

References

[AMO93] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading MA., 1974.

[BJM98] C. Barrett, R. Jacob, M. Marathe, *Formal Language Constrained Path Problems* to be presented at the Scandinavian Workshop on Algorithmic Theory, (SWAT '98), Stockholm, Sweden, July 1998. Technical Report, Los Alamos National Laboratory, LA-UR 98-1739.

[TR+95a] C. Barrett, K. Birkbigler, L. Smith, V. Loose, R. Beckman, J. Davis, D. Roberts and M. Williams, *An Operational Description of TRANSIMS*, Technical Report, LA-UR-95-2393, Los Alamos National Laboratory, 1995.

[CS97] R. Beckman et. al. *TRANSIMS-Release 1.0 – The Dallas Fort Worth Case Study*, LA-UR-97-4502

[CGR96] B. Cherkassky, A. Goldberg and T. Radzik, *Shortest Path algorithms: Theory and Experimental Evaluation*, Mathematical Programming, Vol. 73, 1996, pp. 129–174.

[GGK84] F. Glover, R. Glover and D. Klingman, *Computational Study of am Improved Shortest Path Algorithm*, Networks, Vol. 14, 1985, pp. 65–73.

[EL82] R. Elliott and M. Lesk, "Route Finding in Street Maps by Computers and People," *Proceedings of the AAAI-82 National Conference on Artificial Intelligence,* Pittsburg, PA, August 1982, pp. 258-261.

[HNR68] P. Hart, N. Nilsson, and B. Raphel, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on System Science and Cybernetics,* (4), 2, July 1968, pp. 100-107.

[HM95] Highway Research Board, *Highway Capacity Manual*, Special Report 209, National Research Council, Washington, D.C. 1994.

[JBM98] R. Jacob, C. Barrett and M. Marathe *Models and Efficient Algorithms for Routing Problems in Time Dependent and Labeled Networks,* Technical Report, LA-UR-98-xxxx, Los Alamos National Laboratory, 1998.

[Ko98] G. Konjevod, et al. *Experimental Analysis of Routing Algorithms in in Time Dependent and Labeled Networks*, in preparation, 1998.

[LR89] M. Luby and P Ragde, "A Bidirectional Shortest Path Algorithm with Good Average Case Behavior," *Algorithmica*, 1989, Vol. 4, pp. 551-567.

[Ha92] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36-42 (1992).

[Ma] Y. Ma, "A Shortest Path Algorithm with Expected Running time $O(\sqrt{V}\log V)$," Master's Thesis, University of California, Berkeley.

[MCN91] J.F. Mondou, T.G. Crainic and S. Nguyen, *Shortest Path Algorithms: A Computational Study with C Programming Language*, Computers and Operations Research, Vol. 18, 1991, pp. 767–786.

[NB97] K. Nagel and C. Barrett, *Using Microsimulation Feedback for trip Adaptation for Realistic Traffic in Dallas,* International Journal of Modern Physics C, Vol. 8, No. 3, 1997, pp. 505-525.

[NB98] K. Nagel, *Experiences with Iterated Traffic Microsimulations in Dallas,* in D.E. Wolf and M. Schreckenberg, eds. Traffic and Granular flow II Springer Verlag 1998. Technical Report, Los Alamos National Laboratory, LA-UR 97-4776.

[Pa74] U. Pape, *Implementation and Efficiency of Moore Algorithm for the Shortest Root Problem*, Mathematical Programming, Vol. 7, 1974, pp. 212–222.

[Pa84] S. Pallottino, *Shortest Path Algorithms: Complexity, Interrelations and New Propositions*, Networks, Vol. 14, 1984, pp. 257–267.

[Po71] I. Pohl, "Bidirectional Searching," *Machine Intelligence*, No. 6, 1971, pp. 127-140.

[SV86] R. Sedgewick and J. Vitter "Shortest Paths in Euclidean Graphs," *Algorithmica*, 1986, Vol. 1, No. 1, pp. 31-48.

[SI+97] T. Shibuya, T. Ikeda, H. Imai, S. Nishimura, H. Shimoura and K. Tenmoku, "Finding Realistic Detour by AI Search Techniques," Transportation Research Board Meeting, Washington D.C. 1997.

[ZN98] F. B. Zhan and C. Noon, *Shortest Path Algorithms: An Evaluation using Real Road Networks* Transportation Science, Vol. 32, No. 1, (1998), pp. 65–73.

Appendix: Description of Basic Algorithms

In this section, we describe the basic algorithms considered in this paper. Most of the results in this section are not new; we recall them here for completeness and for description of experimental results.

8.1 Dijkstra's Algorithm

Dijkstra's algorithm solves the single source shortest path problem on a weighted (un)directed graph $G(V, E)$, when all the edge weights are nonnegative. Let $w(u, v)$ denote the weight of an edge in the network.

Suppose we wish to find a shortest path from s to t. Dijkstra's algorithm maintains a set S of vertices whose final shortest paths from the source s have been already computed. The algorithm repeatedly finds a vertex in the set $u \in V - S$ which has the *minimum* shortest path estimate, adds u to S and updates the shortest path estimates of all the neighbors of u that are not in S. The algorithm continues until the terminal vertex is added to S. In general, it is convenient to think of the vertices in the graph being divided into three classes during the execution of the algorithm: (i) *shortest path tree vertices* – (those which have been added to S and hence their shortest path has already been determined, (ii) *unseen vertices* – those for which the distance estimate is ∞ and (iii) *fringe vertices* – those that are adjacent to the vertices in S but have themselves not been added to S. Now each iteration of the algorithm consists of adding a fringe vertex with minimum distance to the shortest path tree and updating its neighbors to be fringe vertices. Using this terminology, initially, only s is a shortest path tree vertex, neighbors of s are fringe vertices, and others are unseen vertices.

DIJKSTRA'S ALGORITHM outlines the steps of the algorithm. In the remainder of the section, we will use $\delta(u)$ to denote the cost of a shortest path from s to u. We will also assume that $|V| = n$ and $|E| = m$. Also, for a given vertex v let $N(v)$ denote the set of neighbors of v i.e. $N(v) = \{w \mid (v, w) \in E\}$. Finally, by the phrase *extract a vertex* from V we mean choose a vertex and delete it from V.

8.2 Bidirectional Dijkstra's Algorithm

The bidirectional algorithm has been used in the operations research community and analyzed by theoretical computer scientists providing quantitative reasons for its improved performance. (See [LR89, Ma] for more details.) The bidirectional search algorithm consists of two phases. In the first phase we alternate between two unidirectional searches: one forward from s, growing a tree spanning a set of nodes S for which the minimum distance from s is known, and the second that consists of growing a tree spanning a set of nodes D for which the minimum distance from d is known. We alternately add one node to S and one to D until an edge crossing from S to D is drawn. At this point, the shortest path is known to lie within the search trees associate with S and D except for one additional edge from S to D.

DIJKSTRA'S ALGORITHM:

- *Input:* $G(V, E)$ - a network, a source s and a destination vertex d and a non-negative weight function $l : E \rightarrow Z^+$.
- 1. *Initialization:* Set $S = \phi$, $d(s) = 0$ and $\forall v \in V - \{s\}, d(v) = \infty$. Found = 0.
  2. *Iterative Step:* **while** Found = 0 **do**
     (a) *Extract Minimum Step:* Among all vertices $v \in V - S$ extract a vertex v with minimum value of $d(v)$. Set $S = S \cup \{v\}$. If $v = d$ then set Found = 1.
     (b) *Decrease (Update) Key:* For each edge (v, w), such that $w \in N(v)$, set $d(w) = \min\{d(w), d(v) + w(v, w)\}$.
- *Output:* A shortest path from s to d, i.e. a path $p =< v_0, \ldots v_k >$ where $v_0 = s$ and $v_k = d$ and the weight $w(p) \sum_{i=1}^{i=k} w(v_{i-1}, v_i)$ is the minimum over all paths from s to d

8.3 A Modification For Euclidean Graphs: A^*-Algorithm

When the underlying network is Euclidean, it is possible to improve the average case performance of Dijkstra's algorithm. Euclidean graphs are defined as follows. The vertices of the graph correspond to points in R^d and the weight of each edge is proportional to the Euclidean distance between the two points. Typically, while solving problems on such graphs, the inherent geometric information is ignored by the classical path finding algorithms. The basic idea behind improving the performance of Dijkstra's algorithm is from Sedgewick and Vitter [SV86] and is originally attributed to Hart Nilsson and Raphel [HNR68] is simple and can be described as follows. To build a shortest path from s to t, we use the original distance estimate for the fringe vertex such as x, i.e. from s to x (as before) *plus* the Euclidean distance from x to t. Thus we use global information about the graph to guide our search for shortest path from s to t. To formalize this, define $D(x, y)$ to be the Euclidean distance between x and y and define $l(x, y)$ to be the shortest path from x to y in the graph. The length of the path as usual is equal to the sum of the edge lengths that constitute the path: the weight of an edge (x, y) is defined to be $D(x, y)$. Now each fringe vertex x is assigned the following value: $\min_w\{l(s, w) + D(w, x)\} + D(x, t)$ The resulting algorithm runs much faster than Dijkstra's algorithm on typical graphs for the following reasons: (i) The shortest path tree grows in the direction of t and (ii) The search of the shortest path can be terminated as soon as t is added to the to the shortest path tree. The correctness of the algorithm follows from the fact that $D(x, t)$ is a lower bound on $l(x, t)$. Another way to interpret the algorithm and its correctness is by using the concept of vertex potentials – an idea first used by Gabow.

Concept of Vertex Potentials. Each vertex is assigned a non-negative value $D(x)$ – called its *potential*. The intuition is, that we put the vertices in a mountainous region – i.e. we assign a altitude to them. Furthermore we take the edge-weights as the energy needed to get from one end of the edge to the other. Then the resulting energy (cost) needed to traverse an edge is the original cost modified by the potential (energy) difference of the endpoints. This leads to the following formal definition:

$$\forall (u,v) \in E, \;\; \tilde{l}(u,v) = l(u,v) + D(u) - D(v)$$

The potentials are called *admissible* or *feasible* if the new lengths are all positive. The following theorem shows that the the shortest paths in the graph with modified weights remains the same.

Theorem 8.1 *Let D be a set of admissible vertex potential. Then the weight of a path $p =< s = v_1, \ldots v_r = t >$ from s to t is given by*

$$\tilde{w}(p) = \sum_{i=1}^{i=n} l(v_i v_{i+1}) + D(s) - D(t)$$

In other words the length of each path from s to t is changed by the same constant additive amount. Thus if p is a shortest $s - t$ path in the original graph then it is still the shortest path in the graph with modified edge weights.

8.4 Modified A^*

We briefly discuss some of the heuristic improvements to the basic A^* algorithm that can be used in practice. Again, recall that in many practical situations (including TRANSIMS), it is not necessary to find exact shortest paths – approximately shortest paths suffice. We tried two heuristic solutions in this context.

(1) The modified A^* algorithm. Recall that the A^* algorithm (implicitly) takes the estimate (here the Euclidean distance) as an potential. As long as this estimate is a lower bound on the shortest paths for all pairs of nodes, the graph modified by the potential still has only non-negative weights and Dijkstra's algorithm works correctly. If we now increase the influence of the potential by multiplying it, this correctness doesn't have to be maintained. Nevertheless we get an fast heuristic algorithm that produces simple path, that turn out to be still of interestingly high quality. An important advantage of this method is, that we can choose this "overdo parameter", which reflects the strength of the potential all the way from 1 (correct) to ∞ in which case the algorithm always expands the node next (Euclidean) to destination, which tends to be very fast. Note that the usefulness of this heuristic heavily depends on the graph it is used on. As our results in Section 6 point out, it appears that an appropriate constant results in a very good trade-off between quality of solution and the time required.

(2) Combining A^* with Bidirectional Search The discussion in the above sections suggests combining the bidirectional search heuristic with the A^* search. One possible way to do it is to use two potentials $D_s(u)$ and $D_t(u)$ for each vertex, the potentials reflecting the lower bounds (usually geometric distances) of u from s and t. A naive implementation of this idea is unfortunately incorrect, since the two potentials imply building shortest path trees from s and t. As shown in [SI+97], a modified potential suffices to ensure the correctness of the algorithm.

9 Discussion on Data Structures

(1) Arrays versus Heaps. In a naive implementation of the algorithm using an array $\mathcal{A}$, in which for each vertex, v_i we store the value of $d(v_i)$ in location $\mathcal{A}(i)$. In each iteration *Extract Minimum Key* takes $O(n)$ time (finding a minimum value in an unsorted array takes $O(n)$ time) and *Decrease Key* takes time $O(deg(v))$. Here $deg(v)$ denotes the degree of v. The total running time is therefore $\sum_v O(n + deg(v)) = O(n^2 + m)$. Using *Binary Heaps* (as has been done in the current implementation of the algorithm), we can improve the running time. First consider *Extract Minimum Key* operation. The time to do this is $O(\log n)$ since we simply pick the top of the heap and then process the data structure to maintain the heap property (using HEAPIFY). Next consider the *Decrease Key* operation. This operation takes time $O(deg(v) \log n)$ for the following reason. We need to update the distance estimate for each of the $deg(v)$ neighbors, each operation taking time $O(\log n)$. The time to build the heap for the first time is $O(n)$. Thus the total running time of the algorithm is $\sum_v O((\log n) + deg(v) \log n)) = O(n \log n + m \log n) = O(n + m) \log n$. We also considered using Fibonacci Heaps. Our experimental analysis revealed that typically the number of nodes that are kept in a heap is around 500; thus using a more sophisticated data structure with higher constants was not likely to yield better results in practice. Using Fibonacci heaps could potentially improve the theoretical running time of the algorithm by of $\log(\mathcal{H})$, where $\mathcal{H}$ denotes the maximum heap size at any stage of the execution. This implies an improvement of at most a factor of 9. But the constants with the heap operations and the complicated code for implementing this data structure weigh more heavily against it.

(2) Deferred Update. Recall that we need to update the values of the distance estimates in Step 2b of DIJKSTRA'S ALGORITHM. Assume that the heap is $\mathcal{H}$, and degree of a node v being d_v, it would take roughly $2d_v \log \mathcal{H}$ operations to update the distance estimates. The reason for this is as follows: We can maintain an auxiliary array that keeps pointers to the nodes in the heap. Every time a nodes distance estimate is updated, the node moves through the heap (as a part of HEAPIFY operation) to settle in the final position. During the course of this other nodes on its path also change positions. This implies that the pointed values for each of the nodes need be updated. (We are assuming an array implementation of the Heap.) Another possible way to do this is to insert multiple copies of a node in the heap. In this way, the time taken is roughly proportional to adding these nodes plus the additional factor depending on the size of the heap for future operations. Again, let d_v denote the degree of a node and d_{max} be the maximum degree. Then the heap size grows at most by a multiplicative factor of d_{max}. Since the Heap operations take time roughly $\log \mathcal{H}$ this implies that the total time for executing Step 2b is no more than $d_{max} \log(d_{max}\mathcal{H})$ which is $d_{max}(\log \mathcal{H} + \log d_{max})$. Typically, the average degree of a node in the Case study network is $2.6 \sim 3$ and you expect that it only gets inserted roughly only by half its neighbors resulting in an average increase of no more than 4 on the size of the heap. This implies that we spend only an additional additive factor of $2d_v$ for each run of Step 2b.

(3) Hash Tables for Storing Graphs. The graph we received from Dallas MPO is given using long Link and Node Id's. Although the naming convention is useful in other contexts, such a naming convention yields a inefficient use of the domain space. To illustrate the point, the link and the node It's given were typically made of 32 bits long. Thus the name space of for the nodes is roughly 2^{32}. In contrast the number of nodes is roughly $10^4 \sim 2^{12}$. Such a discrepancy immediately motivated a use of hash tables to improve the name space utilization. We used a Hash table of size roughly 2^{14} This is achieving efficiency for two possible reasons. The first and more important reason is that the array used to store the structure (information) associated with each node is small enough to typically fit the first level cache. In contrast arrays of size 2^{32} will never fit in a fast cache and thus will imply a significant increase in the processing time. It is well known that memory access is significant bottleneck in the design of fast algorithms. Another reason is that small words might be useful in minimizing the amount of memory accessed in the inner loops of the algorithm. Also, note that the Hash table needs to be accessed only during input and output of the plans and thus the process, even if it were expensive, does not contribute significantly to the time the planner uses.

(4) Smart Label Reset We now discuss the improvement performed in the context of finding paths for a number of travelers. Note that in Step 1 we need to set the distance estimates of all the nodes to be initialized to infinity. This takes $O(n)$ time per run of the algorithm. We instead relabel only those nodes whose labels have changed during the course of the algorithm. This simply consists of the nodes that were at anytime inserted in the heap. Since on an average the total number of nodes visited is a small fraction of the total number of nodes (in fact is $O(\sqrt{n})$ for bidirectional implementation) this yields significant improvements in the running time of the algorithm.

(5) Heap tricks At the innermost loop of our heap implementation are two small details: one is the test on a special case at the end of the heap. This test can be replaced by setting unused elements of the array to the value infinity, by this replacing an operation in the loop by (possibly) one more iteration in the loop. The other possibility is to "streamline" the comparison at this loop from possibly four down to three.

(6) struct of arrays vs. array of structs Following object oriented design goals one ends up having different, independent arrays for storing data for the network, label-setting and the shortest-path-tree module. Considering caching behavior of the processor it seems advantageous to combine these to one big array of structs having entries for the different modules.