Classical and contemporary shortest path problems in road networks: implementation and experimental analysis of the TRANSIMS router

Chris Barrett[1], Keith Bisset[1], Riko Jacob[2], Goran Konjevod[3], and Madhav Marathe[1]

[1] Los Alamos National Laboratory, P.O. Box 1663, MS M997, Los Alamos, NM
`barrett,bisset,marathe@lanl.gov`

[2] BRICS, Department of Computer Science, University of Aarhus, Denmark.
`rjacob@brics.dk`

[3] Department of Computer Science, Arizona State University, Tempe, AZ.
`goran@asu.edu`

Abstract. We describe and analyze empirically an implementation of some generalizations of Dijkstra's algorithm for shortest paths in graphs. The implementation formed a part of the TRANSIMS project at the Los Alamos National Laboratory. Besides offering the first implementation of the shortest path algorithm with regular language constraints, our code also solves problems with time-dependent edge delays in a quite general first-in-first-out model.

We describe some details of our implementation and then analyze the behavior of the algorithm on real but extremely large transportation networks. Even though the questions we consider in our experiments are fundamental and natural, it appears that they have not been carefully examined before. A methodological contribution of the present work is the use of formal statistical methods to analyze the behaviour of our algorithms. Although the statistical methods employed are simple, they provide a possibly novel approach to the experimental analysis of algorithms.

Our results provide evidence for our claims of efficiency of the algorithms described in a very practical setting.

1 Introduction

TRANSIMS is a multi-year project at the Los Alamos National Laboratory funded by the Department of Transportation and by the Environmental Protection Agency. Its purpose is to develop models and methods to answer planning questions, such as the economic and social impact of building new roads in a large metropolitan area. We refer the reader to [TR+95a] and the website `http://transims.tsasa.lanl.gov` for more extensive descriptions of the TRANSIMS project.

The basic purpose of the TRANSIMS module we describe (the *route planner*) is to use activity information generated earlier from demographic data to determine the optimal mode choices and travel routes for each individual traveler. The routes need to be computed for a large number of travelers (in the Portland case study 5–10 million trips). After planning, the routes are executed by a microsimulation that places the travelers (in vehicles and on foot) in the network and simulates their behavior. In order to remove the forward causality artificially introduced by this design, and with the goal of bringing the system to a "relaxed" state, TRANSIMS uses a feedback mechanism: the link delays observed in the microsimulation are used by the route planner to repeatedly re-plan a fraction of the travelers.

Clearly, this requires high computational throughput. The high level of detail in planning and the efficiency demand are both important design goals; methods to achieve reasonable performance are well known if only one of the goals needs to be satisfied. Here, we propose a framework that uses two independent extensions of the basic shortest path problem to simultaneously cope with both.

1.1 Shortest paths

In the first part of this paper, we describe our implementation of a generalized Dijkstra's shortest path algorithm. The general problem we solve is that of finding regular-language-constrained shortest paths [BJM98] in graphs with time-dependent edge delays with a first-in-first-out assumption. Several authors have studied special cases (such as traffic-light networks or special cases of the regular language constraint), but these studies seem to be isolated and largely independent of a larger real-life system or application. As far as we are aware, ours is the first implementation scalable to problems with millions of vertices and edges that can find shortest-path problems in the presence of **both** language constraints and time dependence.

The various models and practical settings, especially in the context of multimodal urban transportation systems, are discussed in a companion paper [BB+02].

1.2 Formal language constraints

Consider a pedestrian bridge across a river or a highway. Suppose we are asked to find a shortest path between some two points in the network. The bridge can only be used by pedestrians, so we must take care not to route cars across it. Similarly, we should not use highways as parts of the routes planned for pedestrians or bicyclists. In order not to have to update the network for every single routing question, we annotate the network with information needed to deal with these problems. More precisely to each edge and/or vertex of the network, we assign a label $\ell \in \Sigma$. We also refer to the finite set Σ as the *alphabet*. We call such labels *modes* and say that a labeled network is *multimodal*.

By concatenation, the edge and/or vertex labeling extends to walks. The resulting string of labels is called the *label* of the walk. This walk-label determines whether or not the walk is acceptable as a particular traveler's itinerary. We usually refer to walks in the network as paths; in other words, we allow our paths to repeat edges and/or vertices and instead use the term *simple path* to denote paths that do not repeat edges or vertices.

The problem of finding a shortest path subject to a formal-language constraint can be stated as follows: given a finite set (*language*) $L \subseteq \Sigma^*$ over the alphabet Σ, a source node s and a destination node d, find a shortest path p from s to d whose label belongs to L. Our results in [BJM98] prove that this problem can be solved in polynomial time.

Regular languages as models for shortest-path problems were also suggested independently by Romeuf [Rom88], Yannakakis [Ya90] and Mendelzon and Wood [MW95]. For more details on the theoretical background, refer to [BJM98].

1.3 Time dependence

Finding optimal paths in time-dependent networks is an important problem [Ch97a,ZM95]. Unless restrictive assumptions are made, time-dependence of delays implies NP-hardness [OR90]. A natural assumption is that the traffic on each link has the first-in-first-out property. Our model, using *piecewise-linear delay functions*, is a natural implementation of this assumption. It has been rediscovered independently at least once more —by Sung et al. [SB+00]—but the full power of this model for various problems arising in transportation science is not evident from their paper.

We argue that this model is (1) adequate for the rapidly changing conditions on roadways and (2) flexible enough to describe more complicated scenarios such as scheduled transit and time-window constraints but also (3) allows computationally efficient algorithms. Several general versions of this problem can be solved efficiently in our framework (more details in the companion paper on models for transportation problems [BB+02]).

1.4 Experimental analysis

In the second part of this paper we describe experimental results. We ran our algorithm on a set of shortest path instances similar to trips that typical urban travelers take each day. Origin-destination pairs were placed in the street network of Portland, Oregon, and a total of 280000 shortest paths found for several categories of travelers and trip types. This allowed us to analyze the behavior of the algorithm on a typical set of problem instances generated in the TRANSIMS framework.

The TRANSIMS network for Portland has been divided into approximately 1200 traffic analysis zones (TAZs). From these a distance matrix was created from the Euclidean distance between each pair of TAZs. Source and destination TAZ pairs were selected from this matrix so that the distance between the source and destination ranged from 1000 to 50000 meters ($\pm 10\%$) in increments of 500 meters, with 50 trips of each size selected. For each TAZ pair, starting and ending points were randomly selected from within the given TAZ, producing 5000 trips.

Through the experiments, we attempt to infer the scalability of our methods and empirical improvements obtained by augmenting the basic algorithm with heuristic methods.

For example, we show the power and limits of the Sedgewick-Vitter heuristic when applied to a not strictly Euclidean graph. We see that the heuristic is only effective up to a certain point, and study its running time in comparison with ordinary Dijkstra's algorithm. We find that the running time of the heuristic is statistically linear in the number of edges of the solution path (thus, a constant fraction of vertices examined ends up in the solution path). The exact algorithm, on the other hand, seems to follow a logistic response function. We believe this to be just an artifact of our experiment design (the correct answer having the form $k \log k$) and we will need to do more experiments to complete this study.

1.5 Discussion

Even though our questions are fundamental, it appears that they have not been examined before. In order to experimentally analyze the behavior of the algorithms in realistic settings we employ simple statistical techniques (e.g. ANOVA), that have to our knowledge not been used earlier in the field of experimental algorithmics. In our opinion, formal statistical techniques such as ANOVA and experimental design may provide an excellent tool for empirical analysis of algorithms. We have recently used similar tools in analyses of certain flow algorithms and interaction of communication protocols [MM+02,BCF+01].

This research should be viewed as experimental analysis of a well-known algorithm and its generalizations and variants in a realistic setting. No references we have been able to locate cover more than a small part of our theoretical framework, and the situation is similar with actual implementations. To the best of our knowledge TRANSIMS contains the first unified implementation of these results and it is both reasonably complete from a theoretician's point of view and useful in practice. Nevertheless, we argue that our algorithm and conclusions on the implementation are not TRANSIMS-specific, but applicable to a number of other realistic transportation problems.

2 Implementation of the TRANSIMS router

2.1 Algorithm for linear regular expressions

First, some (standard) notation: w^+ denotes one or more repetitions of a word (string) w, $x + y$ denotes either x or y, Σ typically denotes the alphabet, that is the set of all available symbols.

TRANSIMS currently supports *linear* (or *simple-path*) regular expressions. These are the expressions of the form $x_1^+ x_2^+ \cdots x_k^+$, where $x_i \in \Sigma \cup (\Sigma + \Sigma)$.

Algorithm 1 *Input:* A linear regular expression R (as the string $R[0 \ldots |R|-1]$), a directed edge-labeled weighted graph G, vertices s and $d \in V(G)$. *Output:* A minimum-weight path p^* in G from s to d such that $l(p^*) \in R$.

Conceptually, the algorithm consists of running Dijkstra's algorithm on the direct product of G and the finite automaton $M(R)$ representing R.

For efficiency, we do not explicitly construct $G \times M(R)$, but concatenate the identifier of each vertex of G with the identifier of the appropriate vertex in $M(R)$. In other words, we run Dijkstra's shortest-path algorithm on G with the following changes: each vertex is referred to by the pair consisting of its index in G and an integer $0 \leq a \leq |R| - 1$ denoting the location within R.

In the first step, $a = 0$ and the only "explored" vertex is $(s, 0)$. In each subsequent exploration step of Dijkstra's algorithm, consider only the edges e leaving the current vertex (v, a) with $l(e) = R[a]$ or $l(e) = R[a+1]$. If an edge $e = vw$ with $l(e) = R[a+1]$ is explored, then the vertex reached will be $(w, a+1)$. Otherwise the vertex reached is (w, a). The algorithm halts when it reaches the vertex $(d, |R| - 1)$. $\square$

Theorem 2. *Algorithm 1 computes the shortest R-constrained path in G (with nonnegative edge-weights) in time $O(T(|R||G|))$, where $T(n)$ denotes the running time of a shortest-path algorithm on a graph with n nodes.*

2.2 Time-dependent delays

To get a class of functions that is flexible enough to model various applications, but computationally feasible, we use **monotonic piecewise-linear** (**MPL**) functions. Among other properties, they allow fast lookup of values and this is important, being a part of the innermost loop of the algorithm.

We represent MPL functions using a sorted set of pairs that can be searched for both x and (linearly interpolated) y values. For functions that do not need to be modified frequently (for example, a real TRANSIMS situation in which the traversal functions are only modified after planning a substantial number of travelers), an implementation using arrays and binary search performs very well.

For details, see our modeling paper [BB+02].

2.3 Data structures and network representation

The network used by the route planner is substantially different from the TRANSIMS network used by the microsimulation and some other modules of the system. The reason for extra work is the improved efficiency achieved by streamlining the description and removing the features ignored by the route planner and reorganizing some others. A schematic drawing of the network used by the planner is given in Figure 1.

Priority queue. Dijkstra's algorithm requires the implementation of a priority queue in order to maintain the fringe vertices amongh which it chooses the next vertex to explore. We used the simple binary heap. Due to a relatively regular structure of graphs involved, the size of the heap never became too large to allow significant improvements by using a more complex data structure.

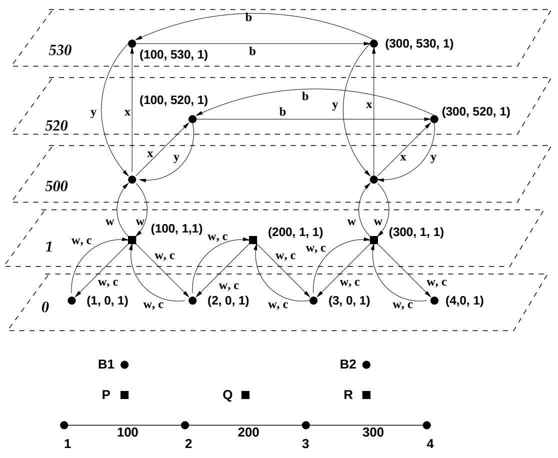

Fig. 1. The network as seen by the route planner. The underlying street network is shown in the bottom of the figure, with nodes 1, 2, 3 and 4 and street segments connecting them. Nodes P, Q and R represent parking locations (present on most links). Associated with the links joining 1 with 2 and 3 with four are also bus stops B1 and B2. The internal network constructed by the planner is shown in the upper portion of the figure, with "layers" separated for clarity. Layers 0 and 1 describe the street network with links that can be traversed by drivers (mode c) and pedestrians (mode w). Nodes in layer 0 are intersections, those in layer 1 parking locations. Bus stops are located in layer 500 and each bus stop is associated with a parking location (and only accessible from it). To actually board a bus, however, a traveler must walk from a bus stop to a node associated with a specific bus line (layer $500 + x$ for bus line x). This forces walking to transfer between buses and helps avoid some modeling artifacts.

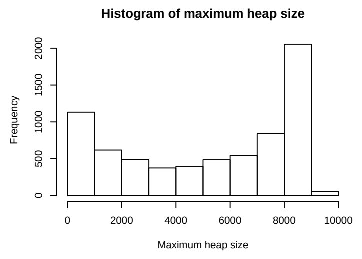

Fig. 2. The maximum heap size over 30000 car trips planned. The heap size never exceeds 10000 vertices, so the standard binary heap is still quite efficient.

2.4 Some low-level implementation details

Software Design. We used the object oriented features as well as the template mechanism of C++ to easily combine different implementations. As we did not want to introduce any unnecessary runtime overhead, we avoided for example the concept of virtual inheritance.

For example, using templates, we implemented different versions of Dijkstra's algorithm for some frequently used mode strings (such as "wcw" for simple car plans and "t" for transit plans) in order to avoid all overhead associated with checking whether certain links may be used by the traveler and most of the overhead of the regular-language Dijkstra. This is somewhat ironic, considering our claim of a unified algorithm.

Compile-time optimization. A simpler algorithm suffices for some types of plans. For example, all-car or all-walk plans should not require the overhead of examining the finite automaton that specifies the constraint language, and may be planned more efficiently if only a part of the network is examined. In addition to the network modification trick to be described, for such cases we use a separately optimized and compiled procedure implemented using the C++ template mechanism.

Thus in fact we are able to implement several variants of the algorithm and transparently vary the underlying network independently so that the appropriate algorithm and network can be chosen and run on a traveler-by-traveler basis with no overhead, we just disguise them as one using C++ templates. In the case where a simple mode is used by a large proportion of travelers, the implementation and program size overhead is small compared to the savings.

Implicit vs. explicit network modification. As described earlier, the network consists of *layers* representing *car*, *walk* and *transit* links, with walk links also crossing between car and transit layers. It is possible to order the creation of edges in the network so that edges with a fixed label form a consecutive interval in the adjacency list of each vertex. Thus for example, edges numbered 0 to i_1 will be car links, those from $i_1 + 1$ to i_2 transit links and those from $i_2 + 1$ to i_3 transit links. Then if the traveler is only allowed to use walk and transit links, we ask Dijkstra's algorithm to only examine the end of each adjacency list and ignore car links completely, at the cost of a single extra table lookup per vertex examined. This trick can be extended to modifying the adjacency list in more general ways as long as the links are added to the network in a sensible order.

Hardware and Software Support. Most of the experiments were performed on an MPP Linux cluster utilizing either 46 or 62 nodes with Dual 500 Mhz Pentium II processors in each node. Each node had 1 Gb of main memory. Many of the experiments were done by executing independent shortest paths runs at each node. For this we had to create copies of the network. Fortunately the network fits in just under 1 Gb of memory and thus did not cause problems. On each node, the route planner was run using 2 routing threads and 1 output thread.
Parallelization. The implementation may use multiple threads running in parallel and it may also be distributed across multiple machines using MPI. Threads enable the parallel execution of several copies of the path-finding algorithm on a shared-memory machine. Each thread uses the same copy of the network. Because separate threads are used for reading, writing and planning, improvements in the running time may be observed even with a single-processor machine.

3 The Sedgewick-Vitter heuristic

One of the additional optimizations we've used for all-car plans is the Sedgewick-Vitter [SV86] heuristic for Euclidean shortest paths that biases the search in the direction of the source-destination vector.

We can modify the basic Dijkstra's 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. This multiplicative factor is called the *overdo* parameter.

4 A first look at the data

As mentioned, the experiments were carried out on a multi-modal transportation network spanning the city of Portland. The network representation is very detailed and contains *all* the streets in Portland. In fact, the data also specifies the lanes, grade, pocket/turn lanes, etc. Much of this was not required in the route planner module, but most of it was used by at least some parts of TRANSIMS (usually the microsimulation or the emissions analyzer).

Types	Street	Parking	Activity	Bus+Rail	Route
Vertices	100511	121503	243423	9771+56	30874
Edges	249222	722745	2285594	55676	30249

In the basic TRANSIMS network, there are a total of 475 264 external nodes and 650 994 external links. The internal network thus grows to over three million edges (see Section 2.3).

Measured quantities. We base our results on measurements and counts of the following quantities: **cpu**: running time used for finding the shortest path (no i/o), **nodes**: number of nodes on the path found by the algorithm, **hadd**: number of nodes added to the heap during the execution, **max**: maximum size of the heap during the execution, **touched**: total number of nodes touched (a node may be counted multiple times here), **unique**: number of unique nodes touched, **edist**: Euclidean (straight line) distance between the origin and destination, **time**: time to traverse the path found by the algorithm,

In addition, each observation can be categorized according to its *mode* (walk, auto, transit, light rail, park-and-ride, bus), *overdo factor* (strength of bias when/if using the Sedgewick-Vitter heuristic—0 (none), 0.15, 0.25, 0.5), *delay* (for car trips— free-speed link delays, or those produced by feedback from the microsimulation after 7 or 24 iterations).

Figure 3 illustrates the relationships between some parameters that are easy to understand. In the later sections, we focus on a few that are not completely obvious. Due to lack of space, our discussion focuses on car trips and on free-speed link delays. We will have more to say (most importantly about the distinctions

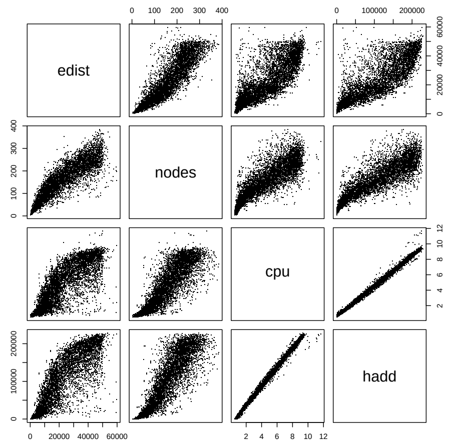

Fig. 3. Plots of Euclidean origin-destination distance, trip duration, running time and total number of nodes added to the heap. The linear relation between the running time and the number of nodes added to the heap during the execution is obvious from the plot and also very clear from the algorithm statement (as long as the time for individual heap operations does not vary too much). We do not consider the Euclidean distance further in this extended abstract.

between various modes and about the adjustments to link delays produced by the microsimulation feedback) in the full version of the paper.

4.1 Varying the Sedgewick-Vitter bias

We now take a look at the results obtained by setting different values of the bias parameter (`overdo`) in the Sedgewick-Vitter heuristic. To summarize briefly, it appears that a value of more than 0.15 is not very useful, as it gives only a marginal improvement in the running time, whereas the path quality continues to decrease. However, as we increase the `overdo` parameter from 0 to 0.15 the running time improves quite substantially.

	Estimate	Std. Error	t value	$P[>\|t\|]$
Intercept	0.8247292	0.0043564	189.32	$< 2 \cdot 10^{-16}$
Slope	0.0002389	0.0000197	12.13	$< 2 \cdot 10^{-16}$

The quality of paths found continues to worsen as the bias parameter is increased, without a corresponding improvement in the running time. Thus we must conclude that it is not useful to push `overdo` beyond 0.15. We still need to investigate the values between 0 and 0.15 to find the best tradeoff.

The reason why the running time can be expected to be linear in the length of the path produced when running the Sedgewick-Vitter heuristic is precisely because of the bias: instead of performing the depth-first-search and expanding equally in all directions (where the number of nodes examined for example in a

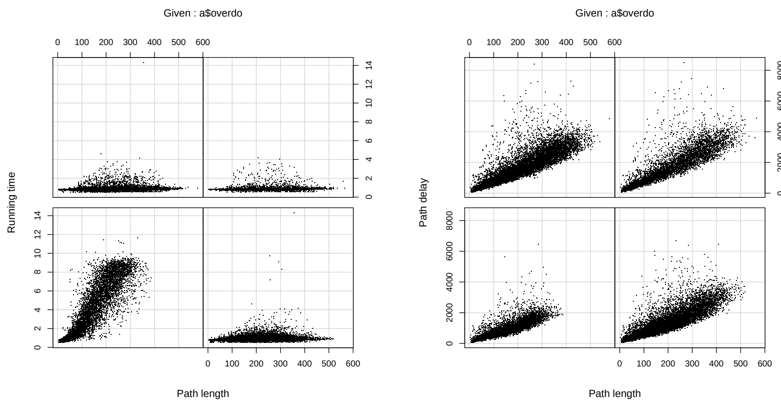

Fig. 4. Running time and path delay against the number of nodes on the path for four different values of `overdo` (0 0.15,0.25,0.5). The figures correspond to increasing overdo parameters and are to be read row wise and bottom up. Note the steep decrease in running time as we go from `overdo` = 0 to `overdo` = .15; the improvement is small after that. We will take a closer look at the horn-shape with `overdo`= 0, but note that the running time appears linear in the other three plots. In fact, it seems *constant*, but a linear fit shows a very low slope, as is to be expected. However, the quality of the paths output continues to decay with increased `overdo`.

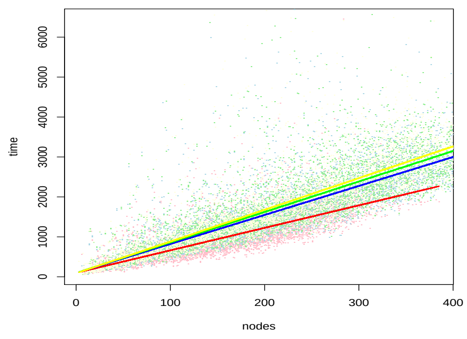

Fig. 5. Delay (quality) of paths produced for various values of the bias parameter. The color scheme for the points and the fitted lines is as follows: red/pink: 0, blue/light blue 0.15, green/light green 0.25, yellow, light yellow 0.5.

grid would be proportional to the square of the path length), the search expands primarily in the single direction towards the destination. Note that this is even a stronger claim than the theoretical result applicable to graphs with Euclidean distance functions, which says that the running time is linear in the size of the graph. However, there are some caveats we should be aware of. By studying the plot for `overdo`= 0.15 (bottom right in Figure 4), we see that only the lower envelope of the data set is a straight line. The upper envelope is not. These points correspond to the cases where the bias led the algorithm astray, for example where the direct geometric route led to the river bank, hoping to get across but not finding a bridge in the vicinity.

An interesting phenomenon is the similarity of running times with the `overdo` set to 0.15, 0.25 and 0.5. The average running times are practically equal, and a formal analysis of variance (ANOVA) test shows that we should not reject the hypothesis of equality of the running times with the bias at 0.15 and 0.25. (Admittedly, it is not at all clear that all the assumptions necessary to validate the test are satisfied.)

4.2 Nonlinear dependence of running time on path length

Finally, let us take a closer look at the dependence of running time on the path length in the case of exact Dijkstra's algorithm (no Sedgewick-Vitter heuristic). The best fit for the data appears to be a logistic response curve of the form $y = \frac{a}{1+e^{\frac{b-x}{c}}}$.

How to explain this behavior? Consider a regular two-dimensional grid (a reasonable approximation to the dense street network of a city). If the path length from the origin to the destination is k, then there are roughly k^2 nodes to be examined before the destination is reached. Thus, we may expect that the running time grows as a square of the path length. However, in a finite grid, if the explored area reaches a boundary, the number of nodes examined must grow slower simply because there are no more nodes to be examined. Thus, after a certain path length is achieved without finding the destination, the growth of the running time should slow down. Indeed, the point of inflection on the logistic curve is just under half the maximum path length among our planned trips, and intuitively, this is where the curve tapers off—when the algorithm runs out of new useless nodes to explore.

Acknowledgements: The research was performed under contract from Department of Transportation and the Environmental Protection Agency and also supported by Department of Energy under Contract W-7405-ENG-36. Riko Jacob and Goran Konjevod worked on this project while visiting Los Alamos National Laboratory. We thank the members of the TRANSIMS team in particular, Richard Beckman, Kathy Birkbigler, Brian Bush, Stephan Eubank, Deborah Kubicek, Kristie Henson, Phil Romero, Jim Smith, Ron Smith, Paula Stretz, for providing the software infrastructure, pointers to related literature and numerous discussions on topics related to the subject. This paper would not have been possible without their support. We also thank David Bernstein, Joseph Cheriyan, Terrence Kelly, Ana Maria Kupresanin, Alberto Mendelzon, Kai Nagel, S.S. Ravi, Daniel Rosenkrantz, Prabhakar Ragde, R. Ravi and Aravind Srinivasan for constructive comments and pointers to related literature.

References

[BB+02] C. Barrett, K. Bisset, R. Jacob, G. Konjevod and M. Marathe *Algorithms and models for routing and transportation problems in in time-dependent and labeled networks*, in preparation, 2002.

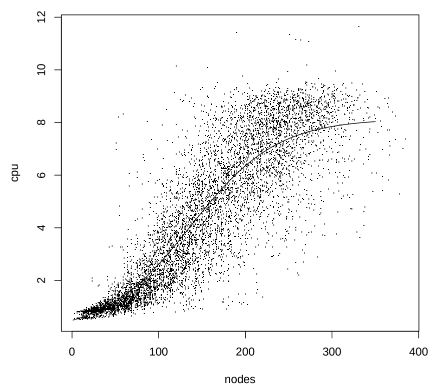

Fig. 6. Running time plotted against the number of nodes on the path produced by the algorithm for `overdo`= 0. The parameters were estimated at $a = 8.14297$, $b = 137.083$ and $c = 49.629$. Notice that the point of inflection is a little under half the maximum path length.

[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.

[BCF+01] C. Barrett, D. Cook, V. Faber, G. Hicks, A. Marathe, M. Marathe, A. Srinivasan, Y. J. Sussmann and H. Thornquist, *Experimental analysis of algorithms for bilateral-contract clearing mechanisms arising in deregulated power industry*, Proc. WAE 2001, pp. 172–184.

[BJM98] C. Barrett, R. Jacob, M. Marathe, *Formal Language Constrained Path Problems* in SIAM J. Computing, 30(3), pp. 809-837, June 2001.

[Ch97a] I. Chabini, *Discrete Dynamic Shortest Path Problems in Transportation Applications: Complexity and Algorithms with Optimal Run Time,* Presented at 1997 Transportation Research Board Meeting.

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

[JMN99] R. Jacob, M. Marathe and K. Nagel, *A Computational Study of Routing Algorithms for Realistic Transportation Networks,* invited paper appears in ACM J. Experimental Algorithmics, 4, Article 6, 1999. `http://www.jea.acm.org/1999/JacobRouting/` Preliminary version appeared in Proc. 2nd Workshop on Algorithmic Engineering, Saarbrucken, Germany, August 1998.

[MM+02] C. L. Barrett, M. Drozda, A. Marathe, and M. Marathe, *Characterizing the interaction between routing and MAC protocols in ad-hoc networks*, to appear in Proc. ACM MobiHoc 2002.

[MW95] A. Mendelzon and P. Wood, *Finding Regular Simple Paths in Graph Databases*, SIAM J. Computing, vol. 24, No. 6, 1995, pp. 1235-1258.

[OR90] A. Orda and R. Rom, *Shortest Path and Minimum Delay Algorithms in Networks with Time Dependent Edge Lengths*, J. ACM, Vol. 37, No. 3, 1990, pp. 607-625.

[Rom88] J. F. Romeuf, *Shortest Path under Rational Constraint* Information Processing Letters 28 (1988), pp. 245-248.

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

[SB+00] K. Sung and M. G. H. Bell and M. Seong and S. Park, *Shortest paths in a network with time-dependent flow speeds*, European Journal of Operational Research 121 (1) (2000), pp. 32–39.

[Ya90] M. Yannakakis "Graph Theoretic Methods in Data Base Theory," invited talk, *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Database Systems (ACM-PODS)*, Nashville TN, 1990, pp. 230-242.

[ZM95] A. Ziliaskopoulos and H. Mahmassani, *Minimum Path Algorithms for Networks with General Time Dependent Arc Costs*, Technical Report, December 1997.