9

15

LA-UR- 01-4698

Approved for public release; distribution is unlimited.

c./

Title:

ROUTING IN TIME-DEPENDENT AND LABELED NETWORKS

Author(s):

Christopher L. Barrett, D-2
Keith Bisset, D-2
Riko Jacob, University of Aarhus, Denmark
Goran Konjevod, Arizona State University
Madhav V. Marathe, D-2

Submitted to:

ACM-SIAM Symposium on Discrete Algorithms (SODA 02)
San Francisco, CA
January 2002

Routing in time-dependent and labeled networks

CHRIS BARRETT[2] KEITH BISSET[2] RIKO JACOB[1] GORAN KONJEVOD[3] MADHAV MARATHE[2]

August 1, 2001

Abstract

We study routing problems in time-dependent and edge- and/or vertex-labeled transportation networks. Labels allow one to express a number of discrete properties of the edges and nodes. The main focus is a unified algorithm that efficiently solves a number of seemingly unrelated problems in transportation science. Experimental data gained from modeling practical situations suggest that the formalism allows interesting compromises between the conflicting goals of generality and efficiency.

1. We use edge/vertex labels in the framework of *Formal Language Constrained Path Problems* to handle discrete choice constraints. The label set is usually small and does not depend on the graph. Edge labels induce path labels, which allows us to impose feasibility constraints on the set of paths considered as shortest path candidates.

2. Second, we propose monotonic piecewise-linear traversal functions to represent the time-dependent aspect of link delays. The applications that can be modeled include scheduled transit and time-windows.

3. Third, we combine the above models and capture a variety of natural problems in transportation science such as time-window constrained trip-chaining. The results demonstrate the robustness of the proposed formalisms.

As evidence for our claims of practical efficiency in a realistic setting, we report preliminary computational experience from TRANSIMS case studies of Portland, Oregon.

1 Introduction

We study route-planning models in the context of multi-modal urban transportation systems. The research reported in this paper should be viewed as applied research backed by experimental analysis in realistic settings. Specifically, much of the basic theoretical background for these results is not new and can be found in [BJM98]. Our initial motivation for this study was the TRANSIMS project for transportation analysis and simulation [TR+95a]. Nevertheless, we argue that the solutions are not TRANSIMS-specific, but applicable to a number of other realistic transportation problems.

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 purpose of TRANSIMS is to develop new models and 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. TRANSIMS conceptually decomposes the transportation planning task into three time scales. First, a large time-scale associated with land use and demographic distribution as a characterization of travelers. In this phase, demographic information is used to create *activities* for travelers. Activity information typically consists of requests that travelers be at a certain location at a specified time. and they include information on travel modes available to the traveler. Second, an intermediate time-scale consists of planning routes and trip-chains to satisfy the activity requests. This is the focus of our paper and the TRANSIMS module responsible for this computation is called the *route planner*. Finally, a very short time-scale is associated with the actual execution of trip plans in the network. This is done by a simulation that moves cellular automata corresponding to the travelers through a very detailed representation of the urban transportation network.

The basic purpose of the route planner is to use the activity information (generated earlier from demographic data) about a traveler to determine specific 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

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

[2]Los Alamos National Laboratory, P.O. Box 1663, MS M997, Los Alamos, NM 87545. Email: `barrett,bisset,marathe@lanl.gov`.

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

study 5–10 million trips are planned). 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 has a feedback mechanism: the link delays observed in the simulation are used by the route planner to re-plan a fraction of the travelers. Clearly, this mechanism requires a high computational throughput from the planner. 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 cope with these design requirements simultaneously.

2 Theoretical Results

Our main contribution is a unified modeling framework and an associated efficient algorithm for constrained shortest paths in multi-modal and time-dependent networks. The advantages of our framework are: (1) translation of "real world" questions into mathematically well-defined optimization problems, (2) guidance in the development of algorithms for these problems, and (3) a *single efficient algorithm* for a host of seemingly different optimization problems in transportation science.

An additional goal is to show how to use this framework and the associated algorithm to solve extremely large realistic transportation problems. From a pragmatic point of view, a generic algorithm simplifies the implementation of and experimentation with alternative models. To illustrate the third point above, we give examples where alternative, more direct algorithms are known. The unified framework consists of the three parts described below.

First, we consider models and algorithms for shortest paths with discrete choice constraints. These include travel modes, destination choice, roadway type, etc. In general many of these choices cannot be modeled adequately by edge-weights, but edge- or vertex-labels are more appropriate. Motivated by this, we represent the transportation network by a (possibly time-dependent) *weighted*, (vertex- and/or edge)-*labeled* graph. The labels denote modal or other discrete attributes of the edge (vertex) and are drawn from a finite set. We use regular expressions over the label set to describe feasible paths, explain how to solve these problems efficiently and show how this model encompasses a wide variety of discrete-choice transportation problems (Section 4.2). Regular languages as models for constrained shortest-path problems were suggested earlier by Romeuf [Rom88] and applications to database queries were described by Yannakakis [Ya90] and by Mendelzon and Wood [MW95]. For more details, we refer to Barrett, Jacob and Marathe [BJM98].

Second, we discuss finding (optimal) paths in time-dependent networks. This is an important problem in transportation science [Ch97a, Ch97b, ZM95, ZM92, ZM93]. We propose monotonic piecewise-linear link traversal functions to model time-dependence. We argue that this class is (1) adequate for modeling time-dependent edge lengths in 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. For example, a prototypical question consists in finding the shortest route that takes into account the bus and train schedules. We solve this problem efficiently in our framework (Section 5.3). The ideas we present here are built on a well-established literature on time-dependent shortest-path problems (for a survey see Orda and Rom [OR91]).

Finally, we show how to combine the two models (labels and time-dependence) and the proposed algorithms to capture a variety of important problems including time-windows, trip chaining, etc. These results further demonstrate the robustness of our models and algorithms. To the best of our knowledge, only heuristic methods have been used so far to solve such problems.

3 Experimental results

As mentioned earlier, the algorithms described here have been implemented as part of the TRANSIMS project. This allows testing or our methods on real transportation networks. In order to anchor research in realistic problems, TRANSIMS uses example cases called *Case studies* (see [CS97] for details). Two case studies have been designed—the first one, concluded in May 1997, focused on the Dallas/Fort-Worth (DFW) metropolitan area. It was done in conjunction with a municipal planning organization (MPO) (the North Central Texas Council of Governments, NCTCOG). The second case study is currently underway and focuses on Portland, Oregon. While the goal of the DFW case study was mainly validating uni-modal traffic simulation, the Portland case study will attempt to validate our models and algorithms for multi-modal time-dependent networks. Due to the focus of this paper, we will mainly focus on illustrative experiments done in the context of Portland network. A more detailed experimental study will be found in [BB+01].

Section 7 discusses illustrative experiments that allow us to infer (i) the scalability of our methods, (ii) the power of the modeling framework in capturing realistic problems, (iii) and empirical improvements obtained by augmenting the basic algorithm with heuristic methods.

4 Language-constrained paths

Consider a small pedestrian bridge across a river (or a highway). A traveler can only use the bridge on foot. Since we do not wish to update the network for every single routing question, we annotate the network with such information. More precisely (and abstractly), to each edge and/or vertex of the network, we assign a class (label) $\ell \in \Sigma$. (Σ is a finite set we refer to 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 usually allow our paths to repeat edges and/or vertices and instead use the term *simple path* to denote paths.

More precisely, we specify a *language* $L \subseteq \Sigma^*$ over the alphabet Σ such that any path whose label belongs to L is acceptable. Now we can raise a shortest-path question: given a source node s and a destination node d, find a shortest path p from s to d whose label belongs to L.

Example 4.1. *Multimodal planning.* In a simple multimodal network the edge-labels denote modes of travel allowed on the link. For example, streets will be labeled "c" for car travel, sidewalks and pedestrian bridges "w" for walk, segments of transit lines (buses, rail) "b" and "r", respectively (or, in a simpler model, lumped together under "t" for transit).

Consider routing a traveler who doesn't own a car and takes a bus to her destination. Suppose transfers are undesirable. The traveler will use some *walk* links, then one or more *bus* links and finally again some *walk* links.

In order to find a shortest path for this traveler, the following network suffices. Let there be a vertex for every intersection and every transit stop. For every street block passable to pedestrians (that is, with a sidewalk) between two intersections, add a bidirectional link labeled "w". For every bus line, add a unidirectional link between every consecutive pair of stops and label it "b". Make sure that in order to transfer between buses, a walk link must be used. Now the goal is to find a shortest path between the traveler's origin and destination whose label is of the form $w \dots wb \dots bw \dots w$. □

Note that finding a shortest path with the restriction imposed in the example above does not become any more difficult if the network includes additional arcs with different labels (such as streets ("c"), railway links ("r"), etc.). This shows an important feature of our framework: *it is possible to treat different modal constraints by changing only the constraining language.* Thus we can plan all trips on the same underlying network and avoid the expensive network modification for each different modal constraint.

The following definition formalizes the language constraints. If p is a path in G, by $l(p)$ we denote the label of p, that is, the concatenation of labels of consecutive edges in p.

Definition 4.2. *(Language-constrained shortest-paths.) Given a directed, labeled, weighted graph G, a source $s \in V(G)$, a destination $d \in V(G)$ and a formal language (regular, context free, context sensitive, etc.) L, find a shortest (not necessarily simple) path p from s to d in G such that $l(p) \in L$.*

A complexity analysis of the formal-language-constrained shortest and shortest simple path problems was given by Barrett, Jacob and Marathe [BJM98]. We summarize their results here, using n to denote the number of vertices in the graph G:

(1) If the path is required to be simple, almost all problems are NP-hard. Thus, we only consider shortest paths without the simplicity constraint.

(2) The problem of finding a context-free-language-constrained shortest path is polynomial-time solvable, but the high complexity $O(n^3sr)$ (where s is the number of nonterminals and r the number of rules in the Chomsky normal form of the grammar) of the fastest known algorithm restricts its practicality.

(3) If the language is specified by a nondeterministic finite automaton (NFA), the problem reduces to an ordinary shortest-path problem on a graph with $n \cdot k$ vertices, where k is the number of vertices in the NFA. The solution is in fact a shortest path in the direct product of the graph G and the directed graph representing the NFA.

The last model is the one we consider the most practical for transportation science applications. Hereafter we assume that the constraining language L is specified as a regular expression.

4.1 Algorithm for linear expressions

We now describe the algorithm actually implemented in TRANSIMS. 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. They are of the form $x_1^+ x_2^+ \cdots x_k^+$, where $x_i \in \Sigma \cup (\Sigma + \Sigma)$ for all i.

Note that if $R_1, R_2, \ldots R_k$ are linear regular expressions, then the expression $R_1 + \cdots + R_k$ can also be easily handled by finding the best path for each R_i and then choosing the best one. We call such expressions *rooted paths regular*, since the automata graphs form a set of simple paths joined at the root.

Algorithm 4.3. *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 $(d, |R| - 1)$. □

Theorem 4.4. *Algorithm 4.3 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.*

The running time of the algorithm is equal to $O(|G| + |R| + Heap \log(Heap))$, where $|G|, |R|$ and $Heap$ respectively denote the encodings of graph and regular expression and the maximum size the heap grows to. The algorithm yields significant savings of time in practice. First, we do not need to construct the product explicitly saving us at least $O(|G| \cdot |R|)$ time. Second, typically the heap size never grows to much. In fact, it appears that the run time of the algorithm is more a function of the path length rather than the entire graph. Our results in the experimental section discuss this further.

4.2 Examples of regular constraints

We give some illustrative examples of regular expressions that might be useful in the context of transportation planning. Rather than being exhaustive, this is a list of problems solvable by a single algorithm and implementation.

1. Trip chaining. Consider the following problem: given a sequence of activities that can be performed at different locations, find the shortest path that allows the traveler to perform the activities in the *given order*. To solve the problem, we create new "virtual" loop links at every possible activity location. We label these links according to the activity that can be performed there. For an activity sequence $ABC \ldots$ we would consider the regular expression $TATBTCT \ldots$ where T denotes a regular expression that allows (arbitrary or restricted) travel in the network. Note that this does not solve the traveling salesman problem (TSP) problem in polynomial time—there we would have to consider all possible $n!$ orderings of n activities to find an optimal solution. On the other hand, if the number of activities n is small, enumerating the $n!$ sequences might be feasible. Figure 1 shows an exmaple trip chained route produced for a traveler in Portland.

2. Label subsets and consistent paths. In the course of generating valid activities for planning and microsimulation, it is necessary to ensure that the paths generated are consistent in their use of modes. For instance, if you parked a car at a subway station, the path found by these methods should make sure that you do not drive until you return to the particular parking lot where you left the car. Additional examples include **(1)** finding a shortest path avoiding trains or highways; **(2)** making sure that if we drop our car at a parking lot and then use transit to go to work, we do not use the car for errands during the day; and **(3)** finding a shortest path that may use the freeway but not the interchange. In all of these examples we restrict the path to use only a particular set of labels. This can be achieved by a single-state automaton. (Alternatively we could remove unwanted links from the network, but doing this explicitly is time-consuming.)

3. Intermediate location. Finding paths that

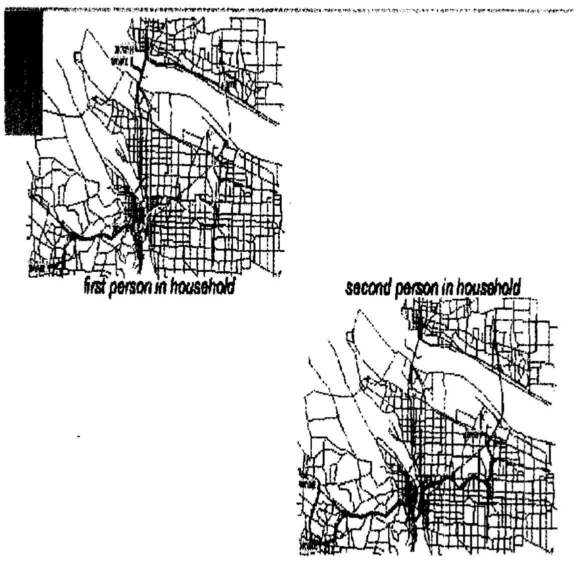

Figure 1: Example of two trip chained routes for two traverles in Portland generated using our router. The first person goes as follows: Home-work-lunch-work-Doctor-shop-home.

use trains so that the train is boarded at a particular subway station.

Here we have to mark the subway station(s) we consider using an appropriate label and then enforce the use of a vertex with this label within the path. This can be done with a two-state automaton. (Alternatively we could split the question into two shortest path computations.)

4. Multimodal plans. See Example 1.

5. Selecting road types. As mentioned earlier, we can use regular expression to express the choice of various road types a traveler might wish to take. Figure 2 illustrates this type of a query on a realistic traffic network—the Dallas/Ft. Worth road network. As explained in the Figure, a traveler can specify the type of roads (e.g. freeways, ramps, arterials) that (s)he wishes to use to complete the trip.

6. Counting constraints. An automaton with $k+1$ states can count up to k occurrences of special links. Thus we can only examine paths with more than k, exactly k or less than k special links. In the latter case we get the simplicity for free. (This follows, for example, from a more general result [MW95].)

The following problems have been studied in the literature and can be immediately solved using the appropriate expression.

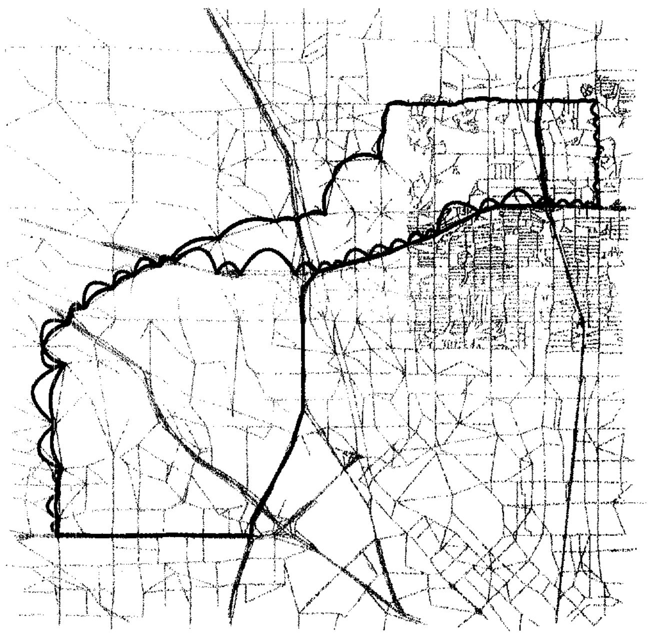

Figure 2: Example in Dallas: unrestricted fastest route is depicted by a straight dashed line, (this is also the fastest route entering the highway system at most once), the fastest route that does not switch between different highways is dashed and curved, the fastest route that stays off the highways is solid.

6a. k-similar paths. There has been considerable interest in algorithms for alternatives to shortest paths [AMO93]. For example, in a recent paper [SJB97], the authors consider the following problem: given a graph G, a (shortest) path P in G and an integer parameter, find the shortest path Q in G that has at most k links in common with P. Call this path the *best k-similar path.* The authors use a Lagrangian relaxation. Although the algorithm appears efficient in practice, it is exponential in the worst case.

Using the above terminology, we make all the links of the (shortest) path P "special." Then the described automaton ensures that the path found does not have more than k links in common with P.

6b. Turn complexity. Assume we have an extended network, where every movement across an intersection (going straight or turning) is explicitly represented as a link. Then we can use 2 and 6a above to search for the shortest path using, for example, less than k left turns.

6c. Transfers. If the network representation uses links of different type to represent bus transfers, we can easily encode the maximum number of transfers allowed.

5 Time-dependent delays

Experience with realistic transportation systems and simulations like TRANSIMS makes it clear that the dynamics of the link delays are an important component of urban traffic. The problem of finding the "best" path in a time-dependent network has been studied extensively (see for example [OR90, ZM95, ZM93]). Since the literature on this subject contains different notions of time-dependence, we first define our terminology.

The fundamental assumption is that the delay incurred by traversing a link cannot be represented by a single value. Instead, every link (a, b) in the network has an associated *link traversal* function $f_{(a,b)}$, defined so that *a traveler starting across the link at endpoint a and time t arrives at endpoint b at time* $f_{(a,b)}(t)$.

Definition 5.1. *A path from a* source s *to a destination t is a sequence of $n+1$ vertices $v_0, v_1, \dots v_n$ and times $t_0, t_1, \dots, t_n$ that satisfies (1) $v_0 = s$; (2) there exists a link from v_i to v_{i+1}; (3) if f is the link traversal function for the link from v_i to v_{i+1}, then $t_{i+1} = f(t_i)$; and (4) $v_n = t$. The numbers t_0 and t_n are also referred to as the departure time and* arrival *time, respectively.*

Earliest arrivals and delay assumptions. As a motivation, consider a traveler starting from an origin node at a certain point in time and looking for a route through the network with the earliest possible arrival time at the destination. Orda and Rom [OR90] present a theoretical study of the complexity of such shortest-path problems. Their basic model consists of a function which describes the delay incurred in traversing the link by a traveler who reaches the link at a given time. Such a function is called the *link delay function.* Note that we prefer link traversal functions, however, it is easily verified that these two formalisms are equivalent. Let $f : \mathbb{Q} \to \mathbb{Q}$ be a link-delay function. Then the equivalent link traversal function $t : \mathbb{Q} \to \mathbb{Q}$ is defined by $t(x) = f(x) + x$. We assume two conditions on delays/traversal times.

The first is the *delay nonnegativity constraint* $f(x) \geq 0$ which becomes $t(x) \geq x$ for a traversal function t (we also write this as $t \geq \text{id}$). A link traversal function t whose equivalent link delay function is nonnegative is said to have a *positive delay*.

The second is the *first-in-first-out condition*, that is a traveler entering the link first leaves the link first as well. This is ensured by the fact that the link traversal function t is *monotone nondecreasing*, that is $f(t') \geq f(t)$ if $t' \geq t$. Strictly speaking, this condition is not realistic because it ignores, for example, passing by cars on the roads. However, without it the problems become too difficult. Note that we do not restrict our functions to $\mathbb{Q}^+$; this is necessary in order to define an incoming shortest-path tree, which corresponds to a shortest-path request with a fixed arrival time.

5.1 MPL functions

The two assumptions of the previous section were motivated by common sense (nonnegativity) and computational feasibility (monotonicity). To get a class of functions that is flexible enough to model various applications, but also allows efficient modeling within a computer program, we use **monotonic piecewise-linear** (MPL) functions. Among other properties, this class allows fast lookup of values and this is important, being a part of the innermost loop of the algorithm. But first, the definitions.

Using $\pm\infty$ is a convenient shortcut for expressing (temporary) unavailability of a link. Thus we define the traversal functions on the extended set of rational numbers $\overline{\mathbb{Q}} = \mathbb{Q} \cup \{\infty, -\infty\}$.

Definition 5.2. *A function $f : \overline{\mathbb{Q}} \to \overline{\mathbb{Q}}$ is called piecewise linear if there exist values $x_1, x_2, \dots, x_n \in \overline{\mathbb{Q}}$ such that $x_i \leq x_{i+1}$ and for $t \in \langle x_i, x_{i+1} \rangle$ the value of f is given by $f(t) = a_i + t \cdot b_i$ for some $a_i, b_i \in \mathbb{Q}$.*

Monotonic functions also have the advantage of being (almost) invertible.

Definition 5.3. *Let $f : \overline{\mathbb{Q}} \to \overline{\mathbb{Q}}$ be an MPL function. Then $g : \overline{\mathbb{Q}} \to \overline{\mathbb{Q}}$ is a weak inverse of f if g is MPL, $g(t) \geq \sup\{x : f(x) < t\}$ and $g(t) \leq \inf\{x : t < f(x)\}$.*

This definition leaves one the choice of an "optimistic" or a "pessimistic" weak inverse for constant f. Other than this, g is uniquely determined. Let g be any weak inverse of f. It is easy to verify that f is also a weak inverse of g. Furthermore g is piecewise linear if f is piecewise-linear. (Proofs can be found in the appendix.)

A data structure for MPL functions is for example 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, an implementation using arrays and binary search performs well.

5.2 Properties of MPL functions

After having introduced the basic concepts of MPL-functions, we now summarize several useful properties and alternative characterizations.

1. *First-in-first-out property*: If traveler A enters the link before traveler B, the monotonicity implies that B cannot leave the link before A. (No passing.)

2. *Traversal-delay equivalence*: A piecewise-linear traversal function corresponds to a piecewise-linear delay function and vice versa.

3. (Temporary) unavailability of a link can be modeled using $\pm\infty$ as the time values.

4. (Weak) invertibility allows simultaneous answers to earliest arrival and latest departure questions. Simple data structures evaluate a function and (a specific) one of its weak inverses.

5. Rounding errors are well understood. If appropriate we can work with integers and (implicitly) approximate by piecewise-constant functions.

6. Continuity is not necessary for the shortest-path algorithm. Even piecewise-constant link traversal functions are suitable. (Not true for link delay functions.)

7. The functions can be "split":

   Proposition 5.4. *Let c be a monotonic, piecewise linear function with n interpolation points and a constant $f \in \langle 0, 1 \rangle$. Then there exist two piecewise linear functions a and b with n interpolation points satisfying the following properties: (i) $a = id + f \cdot (c - id)$ and (ii) $c = b \circ a$.*

5.3 Time-dependence applications

We illustrate the applicability of MPL functions to modeling link delays by giving a few examples. The main goal is to convince the reader that such functions are adequate for many real situations.

1. Routing on a street network. Assume that we have statistics for every link in a road network used by cars. For example, consider the value of the average speed of the cars arriving at the end of the link within a 15-minute time bin. This data may come from real observations or (more likely) from a simulation, as is the case in TRANSIMS. In TRANSIMS link traversal functions are created using interpolation points: we aggregate the information from every time bin into an imaginary car that arrives at the end of the link precisely in the middle of the time bin. Assuming the car has been moving at the reported average speed, we calculate the corresponding departure time into the link. These interpolation points get are then joined by line segments. In similar ways we can use statistics that report average speed of cars starting into a link during a time bin or just speed averages during a time bin. The difficulty (if any) in interpreting this kind of statistic does not arise from the fact that we want to construct a piecewise linear function. However, we must verify that the resulting link traversal functions are in fact (weakly) monotonic, as expected.

2. Transit with schedules. Consider the earliest arrival problem with scheduled transit. A fixed and accurate schedule for the public transportation system is given and used to create a network with link traversal functions. For simplicity we only look at transit, ignoring the walking part of using transit, and assume that the source and destination are transit stops.

The graph has a node for each transit stop, using the same node for several lines if a transfer is possible at the stop. This allows transfers even if the gains are minimal. Some important details like the volatility of real schedules (buses more than trains) and the inconvenience of transferring are ignored.

A bus going from one stop to another defines a link whose traversal function is defined so that *if we arrive at the bus stop before the bus leaves, we will be at the next bus stop at the arrival time of the bus. Otherwise we will not get there at all.*

Formally, assume a bus departs from node A at time t_1 and arrives at node B at time $t_1 + \Delta$. Then the link traversal function for the link (A, B) is given by

$$f(t) = \begin{cases} t_1 + \Delta, & \text{if } t \le t_1; \\ \infty, & \text{if } t_1 < t. \end{cases} \tag{5.1}$$

To model several buses on the same line during the day, we simply combine the functions: the arrival time at the other end of the link coincides with the arrival time of the next bus on the same line. Such a combination is illustrated in Figure 3. In the figure all the buses but one are assumed to take the same time in traversing the link. It is straightforward to adjust the functions to the fact that at different times of the day the busses travel at different speeds; the buses may in fact be simulated themselves by forcing them to take the scheduled route and the actual simulation delays and congestion may be incorporated in the schedule.

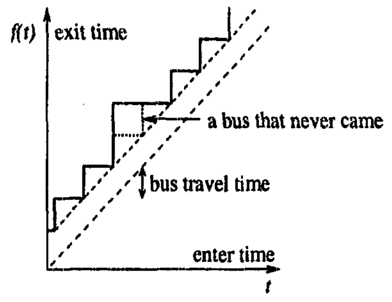

Figure 3: Link traversal function for transit.

3. Shortest path with departure time window. Another problem expressible in our formalism is the following: given a time-dependent network, a source-destination pair and a departure time window, find the departure time within the time window and a path that minimizes overall travel time. The algorithms presented so far cannot solve this question directly. Moreover, although the problem is exactly solvable in polynomial time for unrestricted waiting or monotonic link traversal functions [OR90], the running time might be unacceptable. Therefore we propose an approximation algorithm (in fact an approximation scheme with bounded absolute error) with significantly improved running time. We believe that for real world queries the performance is actually even better than our theoretical bounds imply.

Choose a granularity parameter T. Find a sequence $s_1, s_2, \ldots, s_n$ such that $s_{i+1} - s_i < T$. The time window considered is then $[s_1, s_n]$. Evaluate the fastest path question for all the s_i and take the shortest of those. The worst case occurs when you might have stayed at origin for nearly time T and still arrive at the same time. Therefore the approximate path takes at most T times the optimal time.

6 Combinations

In this final technical section, we consider combinations of the two extensions discussed so far. A desirable feature of the approach is that the algorithm for solving routing problems in time-dependent and the algorithm for solving routing problems in labeled networks can be combined to yield a unified algorithm for routing in time-dependent networks with labeling constraints.

6.1 The Algorithm

Algorithm 6.1. *Input:* A linear regular expression R (as the string $R[0\ldots|R|-1]$, a directed edge-labeled graph G, vertices s and $d \in V(G)$. An MPL traversal function associated with each edge. *Output:* A path from s to d that requires minimum travel time among all those whose labels satisfy R
Similarly to the fixed-delay case, we run Dijkstra's algorithm on $G \times M(R)$. The algorithm remains the same except that link-traversal delays are computed using the MPL functions. To see why this algorithm is correct, think of it as the standard Dijkstra's algorithm run on a graph consisting of multiple copies of $G \times M(R)$, one for each possible time-step. For each time-step t, a copy of $e = uv \in E(G \times M(R)$ with delay $f(e,t)$ joins the tth copy of u to the $t + f(e,t)$-th copy of v The algorithm stops when it reaches *any* copy of the destination vertex d. No vertex is seen twice (even with time-dependence) because the MPL functions are FIFO.

For efficiency, we still do not explicitly construct $G \times M(R)$. In fact, the *only* change in the algorithm is contained in the function that returns the time at which the traveler would arrive at the end of a link as a function of the link identifier and the current time. □

6.2 Examples

1. Trip chaining with time windows. A real-world variant of the trip chaining problem includes a time-dependent transportation network and business hours of the locations where we could perform our activity. To accomplish this, we first create a virtual link (a self-loop) at each activity location. We then use an appropriate link traversal function on the virtual link representing the activity at a certain location. Note that this allows us to adapt to different business hours at different locations of the same type, for example grocery stores. The link traversal function is created to capture the following semantics: if we "enter" before the shop opens, we "arrive" at the opening time of the shop, during working hours we arrive immediately (delay 0), after the shop closes we arrive at time $+\infty$, i.e. never. Formally let a shop s open at time t_o and closes at time t_c. The link traversal function on the virtual link l_s is given by

$$f_{l_s}(t) = \begin{cases} t_o, & \text{if } t \le t_o; \\ t, & \text{if } t_o < t \le t_c; \\ \infty, & \text{if } t_c < t; \end{cases} \qquad (6.1)$$

A function corresponding to a business closed over lunch break is illustrated in Figure 4. To make sure the business is visited during office hours, we label its virtual link and require that the label be used by the path we find. Now solving the shortest-path problem on the modified network with the regular expression constraint yields the required solution.

Note that we can encode the time needed to perform the activity into the link traversal function, modeling for example a situation where we expect the length of queues at the counter to vary with the time of day.

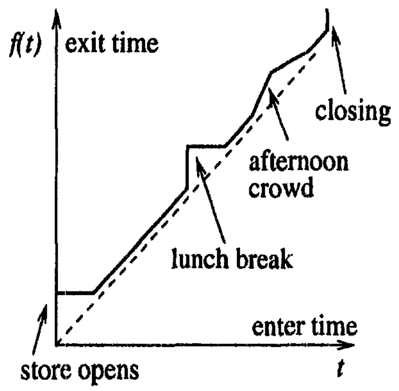

Figure 4: Link traversal function for a store that closes for lunch break and does a lot of business in the afternoon .

2. Vehicle availability. The problem can be stated as follows: find a pair of trips from home to work and back, such that the car is picked up by the second trip exactly where it was left in the first trip. Minimize the overall travel time, i.e the time between departure and arrival at home. We assume fixed arrival and departure times at work. (Otherwise one can do the same type of interval search as for the time window departure.) We solve the problem in our framework as follows:

1. Compute the time-dependent arrival and departure shortest-path tree with transit (not using the car) from the workplace for the appropriate times. This gives a pair of times at every possible parking location p: $t_a(p)$ is the latest time we must arrive there in the morning to reach the office in time and $t_d(p)$ is the earliest time at which we can arrive at the parking location in the evening after starting at a fixed time from the office.

2. Construct a virtual parking link (a self loop at each parking location p) and label it l_p.

3. Each virtual link is assigned a link traversal function as follows: as long as we come to the parking lot earlier than $t_a(p)$ we arrive at the end of the link at time $t_d(p)$. Otherwise (we are late), we arrive at time $+\infty$, i.e. never. Formally, for each virtual link p with times $t_a(p)$ and $t_d(p)$, define $f_p(t) = t_d$ for all $t \leq t_a$, and $f_p(t) = \infty$ for all $t > t_a$. Note that this defines an MPL link traversal function.

4. Then we choose a reasonable time window and run the fastest path with time windowed departure query from home to home subject to the constraint $c^+ l_p^+ c^+$. Here c denotes label of a car link. The solution and the performance guarantee obtained by solving this last problem translate directly into a solution and guarantee for the original problem. □

Note that for this algorithm, the street network to be used with the car and the network a transit passenger uses are completely independent. The process of parking the car and walking from the parking lot to transit stop is modeled "in between" the two explicit networks. Reading out the "virtual parking" link from the two shortest-path trees allows us to model even additional details, like a crowded parking lot.

This can be extended to the trip-chaining problem with one point of the trip fixed in time.

7 Preliminary Experiments

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, data also specifies the lanes, grade, pocket/turn lanes, etc. Much of this was not required in the route planner module.

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

Figure 5: Break down of links and edges by different types.

Runtime	100 %	25 %	20 %	15 %
	14.74	3.64	3.19	2.45

Figure 6: Running time in hours as a function of the fraction of total trips (8.9 million). This is the total wall clock running time on the 124 CPU system.

The networks details are as follows: There were 475 264 external nodes and 650 994 external links. Most of these links were bidirectional. Moreover, no connectivity to parking sites, houses, bus stops was provided. The composite network on which we could route thus became quite large, spanning half a million nodes and over three million edges. (The number of links and edges by mode types is given in Figure 5.) For instance edges under parking column tell us how many edges go to and fro from parking locations. The Portland population located on this network is about 650 000 households with approximately 1.8 million travelers who participate in 8.9 million activities during the course of a 24 hour period.

Results. 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
- maximum heap size
- number of links and length of the path

Figure 6 shows the running time of our algorithm. Roughly, it took 14 hours to run 8.9 million trips on a 124 processor system. The scaling can be seen to be roughly linear: 25 % trips required approximately 4.5 hours.

Figures 7 and 8 shows the traffic density in Portland downtown in the morning using the routing information. Figure 9 is a table describing the average path lengths for set of approximately 1500 travelers using a given modal string. Three different modal strings were used: `w`, `wcw` and `wtw`. The Figure shows the path length in terms the number of time taken with free flow speed, the distance, and the number of hops. All of them report numbers for free flow speed. Numbers when we iterate are available from the authors. The last two columns denote respectivelt the number of nodes touched and the running time taken to find such a route.

Figures 10 and 11 show the tradeoff between the running time and quality of solution as we increase the *overdo* parameter (as suggested by our earlier work in [JMN99] and the work of Sedgewick and Vitter [SV86]). See Section 9 for details on this method. For the discussion here it suffices to remember that overdo parameter is a multiplicative factor used to weigh the Euclidean distance estimate of the current node to the destination.

Figures 12–15 plot the dependence of the running time (in seconds) on (Fig. 12) the distance (in meters) between the origin and the destination, (Fig. 13) the length of the trip (in nodes) and the time (in seconds) the trip takes, for (Fig. 14) car and (Fig. 15) walk trips. Superficially, the dependence does not seem quite linear (as we expected), but we postpone a more detailed statistical analysis for a later paper [BB+01].

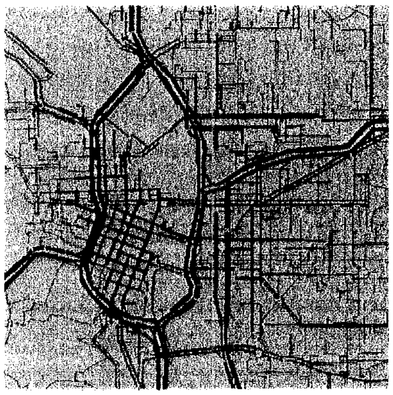

Figure 7: Plot showing the expected number of people in the down town Portland area at 7 a.m. Color coding is green=normal, blue=dense, red=very dense.

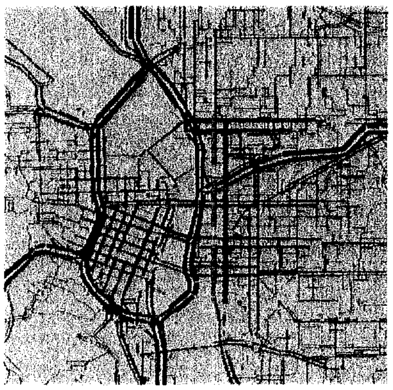

Figure 8: Plot showing the expected number of people in the down town Portland area at 8 a.m. Note the increase in traffic.

mode	time	dist	Hops	Heapsize	Runtime
`w`	4447	4688	41	28391	0.795275
`cw`	402	11445	58	25287	2.713795
`wtw`		21149	154	3827461	62.905374

Figure 9: Table showing the runtime and quality of paths for three different modes. The last two columns also show the number of nodes touched and the time taken to calculate the paths.

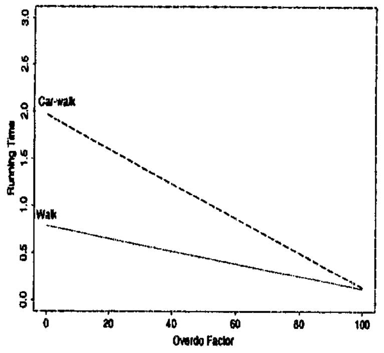

Figure 10: Graph showing the running time as a function of overdo parameter.

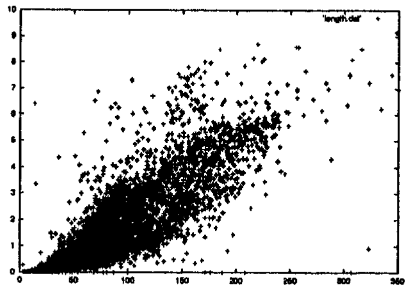

Figure 13: Runtime in terms of the number of nodes.

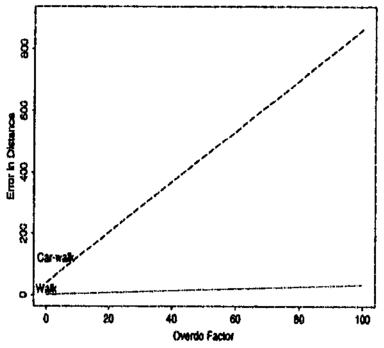

Figure 11: Graph showing the quality as a function of overdo parameter.

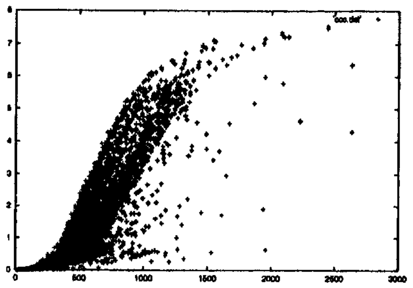

Figure 14: Runtime in terms of trip time (car trips).

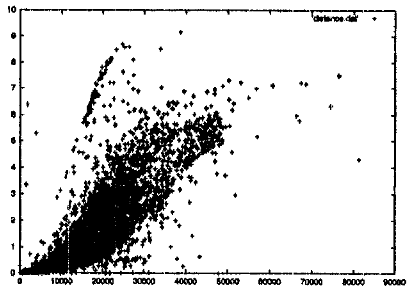

Figure 12: Runtime in terms of the distance

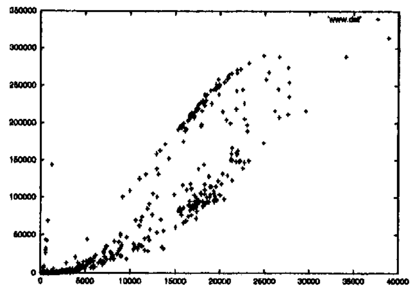

Figure 15: Runtime in terms of trip time (walk trips).

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, 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

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

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

[BB+01] C. Barrett, R. Beckman, K. Bisset, B. Bush, S. Eubank, G. Konjevod and M. Marathe *Experimental Analysis of Routing Algorithms in in Time Dependent and Labeled Networks*, manuscript, 2001.

[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.

[BJM98] C. Barrett, R. Jacob, M. Marathe, *Formal Language Constrained Path Problems* in *SIAM J. Computing*, 30(3), pp. 809-837, June 2001. Preliminary version in *Proc. 6th Scandanavian Workshop on Algorithmic Theory (SWAT)*, Stockholm, Sweden, LNCS 1432, pp. 234–245, Springer Verlag, July 1998.

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

[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.

[Ch97b] I. Chabini, *A New Algorithm for Shortest Paths in Discrete Dynamic Networks*, 8th IFAC/IFIP/IFORS Symposium, Chania, Greece, pp. 551-557.

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

[CH66] K L. Cooke and E. Hasley, *The shortest path through a network with time dependent internodal transit times*, J. Math. Anal. Appl. No. 14, (1966), pp. 493–498.

[De69] S. E. Dreyfus, *An appraisal of some shortest path algorithms*, Operations Research, Vol. 17, 1969, pp. 395–412.

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

[Ha77] J. Halpern, *The shortest route with time dependent length of edges and limited delay possibilities on nodes*, Z. Operations Research, No. 21, 1977, pp. 117–124.

[Ha92] R. Hassin, "Approximation schemes for the restricted shortest path problem", *Mathematics of Operations Research 17*, 1 (1992), 36–42.

[HU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, Reading MA., 1979.

[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*, Volume 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.

[KS93] D. E. Kaufman and R. L. Smith, *The Fastest Path in Time Dependent Networks for Intelligent Vehicle/Highway Systems*, IVHS Journal, Vol. 1 (1993), pp. 91–95.

[Kl72] E. Klafszky, Determination of Shortest Path in a Networks with time Dependent Edge Lengths, Math. Operations for Sch. and Statistics No. 3 (1972), pp. 255–257.

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

[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.

[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.

[OR91] A. Orda and R. Rom, *Minimum Weight Paths in Time Dependent Networks*, Networks, Vol. 21, (1991), pp. 295–319.

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

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

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

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

[SJB97] K. Scott, G. Pabon-Jimenez and D. Bernstein, "Finding Alternatives to the Best Path," *Proc. 76th Annual Meeting of The Transportation Research Board*, Washington, D.C. Paper No. 970682, Jan. '97. Also available as Draft Report *Intelligent Transport Systems Program*, Princeton University, '97.

[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.

[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.

[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.

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

[ZM92] A. Ziliaskopoulos and H. Mahmassani, *Design and Implementation of a Shortest Path Algorithm with Time Dependent Arc Costs,* Proc. 5th Advanced Technology Conference, Washington D.C., (1992), pp. 221–242.

[ZM93] A. Ziliaskopoulos and H. Mahmassani, *A Time Dependent Shortest Path Algorithm for Real Time Intelligent Vehicle/Highway Systems,* Proc. Transportation Research Record, Washington D.C., (1993), pp. 94–104.

Appendix

8 Experimental Setup

Preparing the Network. The data about parking locations, houses, bus stops, etc came as separate file in form of geo-locations. Additionally the light rail data was given separately and thus a substantial amount of time was spent in creating a unified network with all the features. It also naturally increased the size of the network.

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

Hardware and Software Support. Most of the experiments were performed on a MPP Linux cluster with either 46 or 62 nodes with Dual 500 Mhz Pentium II processors in each node. Each node had 1 Gb of main memory. Much 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 1Gb of memory and thus did not cause problems. As mentioned earlier, the code can parallelized using threads but the above method turned out to be more suitable for the experiments reported here. In particular it avoided much of the network contention. We used the GCC compiler. We note that due to design requirements our code is also portable to any SUN machine.

Apart from the scalability tests, most of the experiments reported here were carried out with 1000 randomly selected travelers for each modal choice. For experiments we considered three basic modal choices: walk, walk-car-walk, walk-transit-walk, where transit was either a bus or light rail.

9 Shortcuts and heuristics

A number of data structure and algorithmic heuristics were employed to improve the execution time of the algorithms. We list some of them below.

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.

Implicit vs. explicit network modification. We argued that a unified algorithm allows planning of multiple travelers on a single network, thus circumventing the time-consuming task of generating a new network for each traveler. However, in our implementation, some restrictions on the regular language are in fact implemented using a trick that allows transparent network modification on a traveler-by-traveler basis. Within the route planner, the network is constructed in *layers* consisting of *car*, *walk* and *transit* links, with walk links also crossing between car and transit layers. It is possible to order the addition of edges to 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.

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 NFA and may be planned more efficiently if only a part of the network is examined. In addition to the network modification trick just described, for such cases we use a separately optimized and compiled procedure implemented using the C++ template mechanism.

Sedgewick-Vitter. 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 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 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.

Because street networks are not always dense and regular due to natural and man-made obstacles and also because our delays are not constant, the paths produced using SV are not strictly optimal. However, varying the amount of the bias allows a useful tradeoff between the speed and quality of paths found.

Running time vs. graph size. The running time of Dijkstra's algorithm directly depends on the size of the graph. With larger NFAs (and especially with time-dependence), this size may become a very large number, however with the fairly regular street networks, a more accurate predictor of the running time is in fact the length (the number of edges) of the path found by the algorithm.

Traversal functions. The link-traversal functions are represented using an array of segments. To calculate the delay of an edge at a certain time, the correct segment is first determined using binary search and then the rest is easy.

Turn costs. Turning left may take more time than going straight on through the intersection or turning right. This is easy to implement by expanding each vertex into an in- and out-vertex for each in- and out-going link. However, this expansion need not be explicit: if the current node index also contains the index of the link used to enter the node, we can calculate the turn costs on the fly.

Multicost experiments. In order to model certain aspects of traveler behavior for which NFAs are insufficient, another component may be added to the time delays. For example, certain cost may be associated with some links and the objective might be to minimize a weighted sum of the costs and time-delays along a path.

Delay noise. One of the attempts to speed up the convergence of the system in successive planner-microsimulation iterations included adding a small random noise value to each link delay. Strange behavior resulted, but this should have been expected. For example, cars would pull out of a parking lot, drive to the first intersection and promptly make a U-turn to go back in the direction they could have taken the first time. The problem of course is that random noise violates the FIFO condition.

Other jokes the planner told us. Parking locations in the network caused other problems as well. For example, unless care is taken (by implementing turn costs or by separating parking lots associated with the two directions along a link), drivers will use parking lots to make illegal U-turns. Buses were a whole other story. Early after transit modes were implemented, a number of travelers were noticed to transfer between buses multiple times on a trip to save only a few seconds. One of the fixes used has been to increase the time required to transfer between buses (by forcing the traveler to walk off the bus to the node in the walk layer of the network associated with the transit stop).

10 More on time-dependence

Definition 10.1. *Let $f\colon \overline{\mathbb{Q}} \to \overline{\mathbb{Q}}$ be a (weakly) monotonic (piecewise-linear) function. Then $g\colon \overline{\mathbb{Q}} \to \overline{\mathbb{Q}}$ is a weak inverse of f if for all x and t in $\overline{\mathbb{Q}}$ $f(x) < t \Rightarrow x \leq g(t)$ and $t < f(x) \Rightarrow g(t) \leq x$.*

Let g be any weak inverse of f. This definition leaves the freedom of choosing an "optimistic" or "pessimistic" weak inverse for the local situations where f is constant. Otherwise g is uniquely determined. This stems from the fact that $\{x|f(x) = t\}$

is for all t an interval. For values of x, where f is not locally constant, i.e. when $\{x|f(x) = t\}$ is a single value. Then for all $\varepsilon > 0$ we have (by weak monotonicity of f that $f(x-\varepsilon) < f(x) < f(x+\varepsilon)$. This implies then $x - \varepsilon \leq g(f(x)) \leq x + \varepsilon$, hence $g(f(x)) = x$.

g is non-decreasing (weak monotone increasing). Let $t < s$. If there exists an x such that $t < f(x) < s$, we immediately get $g(t) \leq x \leq g(s)$. Otherwise take some u such that $t < u < s$. By the assumption of this case, there can't be an x such that $f(x) = u$. Therefore are $I = \{x|f(x) < u\}$ and $J = \{x|f(x) > u\}$ are two intervals such that $I \cup J = \mathbb{Q}$. If both of them are open, there exists an z such that for all $\varepsilon > 0$ we have $z - \varepsilon \in I$ and $z + \varepsilon \in J$. So we have $f(z+\varepsilon) > u > t$ and hence $z+\varepsilon \geq g(t)$. Analogously we get $f(z-\varepsilon) < u < s$ and hence $z - \varepsilon \leq g(s)$. Together we get $g(t) \leq z \leq g(s)$.

If J is the empty set or then singleton $\{+\infty\}$, we get for all $x \in \mathbb{Q}$ that $f(x) < s$ and therefore $x < g(s)$, hence $g(s) = \infty$. Analogously, if I is not open we get $g(t) = -\infty$. So we trivially get weak monotonicity in these cases. It is easy to verify that f is also weak inverse of g.

Proof. (Proposition 5.4.) To verify the above claim, assume c is given by the value pairs $c_x(i)$ and $c_y(i)$.Define $a_x(i) = c_x(i)$ and $a_y(i) = a_x(i) + f \cdot (c_y(i) - c_x(i))$ Define $b_x(i) = a_y(i)$ and $b_y(i) = c_y(i)$. The condition that a splits c at fraction f is obvious by the definition of a. For the second condition, first note that $a \circ b$ matches with c for the values $c_x(i)$. As the images of $c_x(i)$ under a are the only non-linearity points in the domain of b, it follows $a \circ b = c$. Note that in the above proof the definition of a can more or less arbitrarily depend on c and there still exists a proper extension function b. □

10.1 Waiting models and theory

Three different waiting policies have been studied in the literature [OR90].

Forbidden waiting: compatible with Definition 5.1.
Unrestricted waiting: waiting allowed everywhere. Relaxes Definition 5.1 to $f(t_i) \leq t_{i+1}$.
Source Waiting: waiting only allowed at the source.

Especially with unrestricted waiting, it is useful to consider a modification of the link traversal function. Given the original link traversal function f, we define the *optimal-waiting* link traversal function g by $g(t) = \inf_{t' \geq t} f(t)$. This captures the optimal waiting strategy for a traveler by combining the optimal waiting at a node with the time needed to traverse the link. If the infimum is not a minimum, there may be no optimal path. Then the best we can hope for is to find a path such that adjusting the waiting times achieves an arrival time arbitrarily close to the optimal arrival time. In the sequel we assume for simplicity that this infimum is always achieved. As mentioned earlier, the basic results on this topic can be found in [Ch97a, Ch97b, OR90, ZM95].

We summarize the computational complexity of shortest-path problems in time-dependent networks with various forms of waiting. As stated earlier some of these results are not new [Ch97a, Ch97b, OR90, ZM95].

Theorem 10.2. **(1)** *In the forbidden waiting model with arbitrary link-traversal functions and a given departure time, finding a path with earliest arrival time is NP-hard. For monotonic, nonnegative link-traversal functions, the problem is polynomial time solvable.*
(2) *In the unrestricted waiting model with positive delay link traversal functions, with polynomial time computable optimal-waiting link-traversal functions, the problem of computing a path with the earliest arrival time is solvable in polynomial time.*
(3) *For the source waiting model with arbitrary functions, the earliest possible arrival time is as hard as the problem for the forbidden waiting model.* *

include the source waiting easiness (??)

Sketch of the Proof of Theorem 10.2:

Part 1. NP-hardness follows by a polynomial time reduction from the partition problem. For a monotonic, positive delay function, the problem can be solved using a straightforward extension of Dijkstra's label setting algorithm: instead of setting labels for the shortest path distance we set labels for the earliest possible arrival time. As the functions represent positive delays, we can set the labels starting from the smallest without having to change them later on. Because the functions are monotonic, it is sufficient to consider the earliest possible time to start into a link (it cannot pay off to start into it later).

Part 2. Note that the Optimal-Waiting link traversal functions are monotonic and inherit the positive delay property. Therefore the algorithm described in the main body of the paper can be used. As for every topology of a path one fastest path is represented, the computed path is optimal.

Part 3. By adding links that can only be traversed at a certain point of time, we get a reduction from the Forbidden Waiting earliest arrival problem. □