Implementations of routing algorithms for transportation networks

Chris Barrett* Keith Bisset† Martin Holzer‡ Goran Konjevod§
Madhav Marathe¶ Dorothea Wagner‖

November 8, 2006

Abstract

We discuss the generalization of the point-to-point (and single-source) shortest path problem to instances where the shortest path must satisfy a formal language constraint. We describe theoretical and experimental results on a generalization of Dijkstra's algorithm to finding regular-language-constrained shortest paths. This algorithm forms a model for single-source shortest paths and point-to-point problems that generalizes several previously published algorithms for multimodal shortest path problems.

The experiments include work with two different implementations. One is the original implementation of the restriction of the problem to linear regular expressions, which formed a part of the TRANSIMS project at the Los Alamos National Laboratory. The other is a new implementation of the general regular-language-constrained shortest path algorithm that uses an implicit representation of the product graph. The second implementation also provides several speed-up techniques which have previously only been used for standard point-to-point shortest path problems.

Through our experiments, we study the scalability of the algorithm with respect to the network size as well as with respect to the constraining language complexity. Further, we study the effectiveness of speed-up techniques such as bidirectional search, shortest path containers, bit-vectors and the multilevel technique when applied to the multimodal shortest path problems formalized by the regular language constraints (Some of these are in progress and will be fully described in a more complete version of the paper.)

1 Introduction

The shortest path problem is one of the most basic and best studied problems in combinatorial optimization. We describe and discuss some generalizations of the single-source shortest path problem that allow efficient algorithms and have applications that range from transportation science to databases.

In this extended abstract, we describe results obtained so far and sketch further work planned. Our specific goals include:

1. We describe a generalization of Dijkstra's algorithm, intended to solve regular-language constrained shortest path problems. We discuss two separate implementations, one restricted to a special class of languages most useful for transportation planning, and built into a module of the TRANSIMS transportation simulation and analysis system, and the other fully general, and intended primarily to study the problem, the scalability of the algorithm and its practical applications. Our implementations can efficiently solve many shortest path problems that arise in transportation applications and that

*Virginia Tech
†Virginia Tech
‡Universität Karlsruhe
§Arizona State University
¶Virginia Tech
‖Universität Karlsruhe

can easily be generalized to a time-dependent setting. However, in order to focus on regular-language constraints, we only very briefly discuss the background of the time-dependent problem, and leave the details for another paper.

2. We study the applicability of standard speed-up techniques for shortest path algorithms to the formal-language-constrained setting. For example, we study the benefits of using a goal-directed search in instances where the vertex-set of the graph is a set of points in a geometric space. We also examine the benefits of using a bi-directional search in the context of the regular language constraints. We envision a more comprehensive study, including several other speed-up techniques, and discuss their applicability.

1.1 Shortest paths

The formulation of a *shortest path problem* consists of a network, a pair (source, destination) of points in the network and possibly a starting time. The solution is an algorithm that finds the most efficient route from the source to the destination leaving the source at the given time. Additional constraints may be imposed, restricting the set of feasible routes. Once a network is specified, the source-destination pair and the starting time together form a *shortest-path query*. We focus on *single-source* shortest path problems,

In most cases of interest to us, the network is adequately represented by a *directed graph* $G = (V, E)$ with a *time-dependent cost (delay) function* $c : E \times \mathbb{R}_+ \to \mathbb{R}_+$ that associates with each *edge* e and each *moment in time* t a nonnegative real value $c(e, t)$ called *delay* denoting the time needed to traverse the edge e if the traversal is begun at time t. (We use these definitions for generality and to emphasize that all of our algorithms and implementations can be generalized to the time-dependent case, however, we do not describe the details of the algorithms, nor discuss any experimental results in this paper.) In the typical shortest path problem instance, a *source* $s \in V$ and a *destination* $d \in V$ are given as vertices of G, and the *start time* t_0 as a real number. The problem then consists of finding a *route* represented by a *path*) P in G from the source to the destination in the graph. The delay function provides a natural optimization objective: find the *shortest* path from s to d, as measured by the delay function. In other words, given the start time t_0, find the path from s to d whose total delay (sum of delays on the individual edges of the path) is the smallest possible among all paths that start at the same time t_0.

In order to conform with most of the existing literature on shortest path problems, we allow our paths to repeat vertices, thus effectively using the term to describe what is referred to as a *trail* in the graph-theoretic literature. In fact, we allow our "paths" to be *walks*, that is, they may repeat even edges. However, other features of the model usually imply that the most efficient paths never repeat edges, making this distinction artificial. In rare cases where we need to ensure that a certain path does not repeat vertices, we use the term *simple path*.

The choice of the cost function provides the formulation above with generality, but even more flexibility arises from allowing additional restrictions on the feasible set of paths. The focus of most research has traditionally been the cost function and besides the basic nonnegative cost model [D59], the case where edge costs are allowed negative values is well solved and there is a rich literature on various formulations of shortest-path problems with time-dependent costs [OR90]. There has been comparatively much less work on the other approach to generalization, that is on constraints that restrict the set of feasible paths. There are reports on studies of *multimodal* or *intermodal* shortest paths in transportation science literature, but these are generally of limited applicability and are usually very simple from an algorithmic standpoint. The comprehensive paper on the theoretical complexity of such problems is [BJM98].

1.2 Impetus: TRANSIMS

Our initial motivation for studying shortest paths came from the TRANSIMS project at Los Alamos National Laboratory [TR+95a], and our first implementation formed a module of the larger system for transportation analysis and simulation.

TRANSIMS was a multi-year project at the Los Alamos National Laboratory funded by the Department

of Transportation and by the Environmental Protection Agency, with the purpose of developing new models and methods for studying transportation planning questions, such as the economic and social impact of building new roads in a large metropolitan area. Besides the documentation cited above, we refer the reader to the web-site `http://ndssl.vbi.vt.edu/transims.html` for more extensive descriptions of the TRANSIMS project and its successors. 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 work 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 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 module that places the travelers (in vehicles and on foot) in the network and simulates the actual behavior of drivers and pedestrians. We refer to this module as the *microsimulation*. 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 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.

Besides the the simple single-source shortest path problem, our implementations address the generalization to regular-language-constrained shortest paths [BJM98] and the generalization to graphs with time-dependent edge-delays with a first-in-first-out assumption. While several authors have studied special cases of the problem that we can solve (such as traffic-light networks or special cases of the regular language constraint), these studies are somewhat isolated and largely independent of a larger real-life system or application. As far as we are aware, ours was the first implementation scalable to problems with hundreds of thousands of vertices that could solve shortest-path problems with **both** formal language constraints and time-dependence.

1.3 Formal language constraints

We now motivate and introduce the model that allows solving shortest path problems with regular language constraints. Consider an urban road network with streets differentiated by their purpose and capacity into highway segments, primary and secondary arterials and local (residential) streets. Suppose now that we are asked to find a shortest path from point s to point d in the network that uses at most one highway segment, that is, a path that avoids on- and off-ramps. A similar case is represented for example by asking for a travel route for a pedestrian who is prepared to take a bus but not to transfer between multiple bus routes.

A somewhat simpler example is that of mode restrictions. A pedestrian bridge cannot be used by cars 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.

All the problems described above can be solved by a single general algorithm. In order to see how, assign to each edge and/or vertex of the network a label $\ell \in \Sigma$. We refer to the finite set Σ as the *label alphabet*. We call such labels *modes* and say that the labeled network is *multimodal*. By concatenation, the labeling

extends to paths in the network. The label of a path determines whether or not the path is feasible.

The problem of finding a shortest path subject to a formal-language constraint is the following: given an edge-weighted directed graph, a *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. (Here, of course, the cost or length of the path is measured as the sum of costs of edges belonging to the path.) This problem is solvable in polynomial time.

Regular languages as models for constrained shortest-path problems were first suggested by Romeuf [Rom88] and applications to database queries were described by Yannakakis [Ya90] and by Mendelzon and Wood [MW95]. We give a concise description of this algorithm in Section 2.1, but for more details on algorithmic and complexity-theoretical results we refer to Barrett, Jacob and Marathe [BJM98].

1.4 Time dependence

In some applications, the cost or delay of traversing an edge may vary as a function of time. This is one of the important algorithmic problems in transportation science and has been studied extensively [Ch97a, Ch97b, ZM95, ZM92, ZM93]. However, unless some restrictive assumptions are made, time-dependence of delays causes the shortest path problem to become NP-hard [OR90]. A natural restriction that avoids NP-hardness is that the traffic on each link obey the *first-in-first-out rule*. Our model uses *piecewise-linear traversal functions* and is a natural implementation of this assumption. This model 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 has in our opinion not been previously realized. We argue that this class of functions 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 is that of finding the shortest route for a traveler in a public transportation system. To find this route we must consider the bus and train schedules. Several general versions of this problem can be solved efficiently in our framework, but we leave the detailed study of the time-dependent problem for another occasion.

2 Algorithms

Here we describe the algorithms and our implementations in some detail. We skim over material that has been published earlier and focus on the new contributions.

2.1 Regular language constraints

Barrett, Jacob and Marathe [BJM98] give polynomial-time algorithms for shortest path problems with regular and context-free language constraints. For completeness, we summarize those of their results that are relevant to this paper.

2.1.1 Problem definition

The *regular language constrained shortest path problem*, or REG-ShP, is defined as follows [BJM98].

Given a labeled weighted graph $G = (V, E, w)$, a source s, destination d, and a regular language L, find a shortest (not necessarily simple) s-d path $p = e_1 e_2 \cdots e_k$ in G such that $\ell(p) \in L$. Here $\ell(p)$ is the string defined as the concatenation $\ell(e_1)\ell(e_2)\cdots\ell(e_k)$ of the labels of the edges of p.

This problem definition is not complete until we specify how L is encoded as input to the algorithm. By Kleene's theorem, we may represent a regular language by a deterministic or nondeterministic finite automaton. To allow a concise representation of a larger class of languages, we specify the regular language L by providing to the algorithm a nondeterministic finite automaton (NFA) that accepts L.

In other words, the input to an algorithm for REG-ShP consists of

ALGORITHM RE-CONSTRAINED SHORTEST PATHS:

- *Input*: NFA $A = (Q, \Sigma, \delta, q_0, F)$, a directed labeled weighted graph G, source s and destination d.
-
  1. Construct $G \times A$. The length of each edge in the product graph is chosen to be equal to the corresponding edge in G.
  2. Find a shortest path in $G \times A$ that joins (s, s_0), to some vertex of the form (d, f), where $f \in F$.
  3. Let p^* be the path found in the previous step.
- *Output*: The path in G corresponding to p^*.

Figure 1: The general regular expression case.

1. a representation of the labeled weighted graph G,
2. indices of or pointers to the two distinguished vertices s and d of G, and
3. a labeled directed graph A that represents the state diagram of an NFA that accepts L.

2.1.2 Product graphs and language constraints

Consider the directed graph defined as the direct product of the underlying (labeled, weighted) graph G and the state diagram A of the NFA M. To specify A more precisely, we write $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the vertex set of A (the state set of M), Σ the alphabet over which the labels are defined, δ the transition function of M that defined the edge set of A, q_0 the start vertex of A, and F the set of final vertices of A. The product $G \times A$ is defined to have as vertex set the set of ordered pairs $\{(v, q) \mid v \in V(G),\ q \in V(A)\}$ and the edge set $\{(e, t) \mid e \in E(G),\ t \in E(A),\ \ell(e) = \ell(t)\}$. Given an edge (e, t) in the product graph, its weight is defined to be exactly the weight $w(e)$.

The following theorem unifies all the algorithms we present for the general and special cases of REG-ShP, as well as many of the algorithms from the literature.

Theorem 2.1. *The problem* REG-ShP *is equivalent to finding a shortest path in the product $G \times A$ of the underlying graph G and the state diagram A of an NFA M that accepts L, among all those paths in $G \times A$ that*

1. *start at (s, q_0), where q_0 is the start state of A,*
2. *end at a vertex of the form (d, f), where $f \in F$ is a final state of A.*

In order to prove the theorem, one need only observe that there is a one-to-one correspondence between paths in $G \times A$ that start at (s, q_0) and end at some vertex of the form (d, f) with $f \in F$, and paths in G whose labels belong to L. For a full proof, please refer to Barrett et al. [BJM98].

The discussion of this section now implies that an algorithm for REG-ShP may be specified as follows.

2.1.3 Implicit product graph representation

In this section we explain how to implement the algorithm of Figure 1 without computing an explicit representation of the product graph $G \times A$. The main benefit of such an implementation is the reduction in the required storage space from $\Theta(|G||A|)$ to $\Theta(|G|+|A|)$, where $|G|$ and $|A|$ denote the amount of space required to store a representation of the graph G and A, respectively. (It is true Dijkstra's algorithm may in the worst case examine all of the vertices of the product graph, and in such a case the heap storage required during the execution of the algorithm may be comparable to the storage required for the explicit representation of the product graph. However, such instances are rare.) At the same time, the time complexity of the algorithm does not increase by more than a constant factor.

The basic step in Dijkstra's algorithm is always the same: select the vertex closest to the set of already explored vertices, and add it to the set. This is made efficient by maintaining a priority queue of vertices on the "fringe" of the explored set, keyed by their distances from the origin. Then in each step we can extract the minimum-key element from this set. Once the minimum-key element is added to the set of explored vertices, we iterate through its neighbors and add them all to the priority queue. If a neighbor is already in the queue, its key may be decreased at this point. To do all this, it is sufficient that we be able to efficiently list the neighbors of any given vertex. Thus we do not need to maintain an explicit description of the product graph, but only enough information to iterate through the neighbors of any given vertex. We keep both basic graphs (the network and the NFA state diagram) stored as adjacency lists. The adjacency lists are arranged so that the neighbors of each vertex are sorted according to the labels of the edges joining the vertex to them. The priority queue still contains vertices of the product graph (they may be represented using a pair data structure, or by integer identifiers created by hashing the identifier pairs that represent the graph vertex and the NFA state). This is the only place where storage comparable to that needed for the full product graph may be required in the unlikely situation that most of the vertices of the product graph are examined. When the algorithm needs to examine all the neighbors of a vertex (v, q) (where v is a network vertex and q a state of the NFA), we do as follows:

for all $\alpha \in \Sigma$ **do**
  for all q' reachable from q by a transition labeled α **do**
    for all v' reachable from v by an edge labeled α **do**
      Insert the pair (v', q') into the priority queue.
    end for
  end for
end for

This innermost loop adds exactly the neighbors of q in the product graph $G \times A$ to the priority queue. All of these would be considered in the explicit case as well. We claim that all the flow control operations can be implemented in constant time, and thus the overhead is asymptotically negligible. (We present a detailed experimental analysis in the full version of the paper.)

2.1.4 Linear regular expressions

In this section, we describe a specialization of the algorithm for language-constrained shortest paths to a smaller class of regular languages. This special class includes many languages that arise naturally in transportation planning problems. In addition to representing the graph implicitly, as in the previous section, this specialization includes several other optimizations, resulting in a very efficient algorithm for general shortest path problems in transportation planning applications. This is the algorithm that was first implemented in TRANSIMS in 1998, and the contents of this section are based on a previous paper [BB+02].

First, let us review 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. A *linear* (or *simple-path* regular expression has the form $x_1^+ x_2^+ \cdots x_k^+$, where $x_i \in \Sigma$ 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.

2.1.5 The algorithm (linear regular expressions)

We represent the linear expression $x_1^+ \cdots x_k^+$ by the string $x_1 \cdots x_k$.

The special structure of linear regular expressions allows further optimization. The important observation is the following. Consider a valid source-destination path p and a vertex $u \in p$. Suppose the initial part of p from s to u is labeled by the word $w = x_1 \cdots x_i$ and denote the next vertex on p after u by v. Then there are only two possible labels for the edge $uv \in p$: either x_i or x_{i+1}. Thus we can run Dijkstra's algorithm in the following way: for each vertex that we examined keep the last label observed on the shortest path used to reach this vertex (and possibly more than one). When iterating through the neighbors of the current vertex,

consider only those reachable by edges of the current label, or by edges of the next label in the expression. As long as we keep adjacency lists arranged by label, it will be easy to skip irrelevant ranges and the overhead incurred as opposed to ordinary Dijkstra's algorithm will be very small.

More precisely, let $R = x_1^+ \cdots x_k^+$ be the given regular expression. We run Dijkstra's shortest-path algorithm on G, with the following changes: each vertex is referred to by the pair consisting of its name in G and an integer $0 \leq a \leq |R| - 1$ denoting the "location" of the product vertex within R.

When the algorithm begins, $a = 0$ and the only "explored" vertex is $(s, 0)$. In each subsequent exploration step of Dijkstra's algorithm, we have a "current" product vertex (v, a), where v is a vertex of G and a is the index of a label in R. As for neighbors, we consider an edge $e = vw$ of G that leaves the current vertex v, if and only if the label of e is the current state $R[a]$ or the "next" state $R[a+1]$: $l(e) = R[a]$ or $l(e) = R[a+1]$. When an edge $e = vw$ with $l(e) = R[a+1]$ is explored, then the vertex reached will be stored in the priority queue as $(w, a+1)$. Otherwise the vertex reached is (w, a). The algorithm halts when it reaches the vertex $(d, |R| - 1)$.

Theorem 2.2. *The algorithm described above 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 $O(|G| + |R| + H \log(H))$, where $|G|$, $|R|$ and H denote the encoding sizes of the graph and the regular expression, and the maximum size of the heap, respectively. The algorithm yields significant savings in time. First, we do not need to construct the product explicitly (which takes $O(|G| \cdot |R|)$ time). Second, the heap rarely grows very large. In fact, it appears that the running time of the algorithm is more a function of the path length rather than the size of the graph. We discuss this further in the section on experimental results.

2.2 Regular language speed-up techniques

In the following few subsections, we indicate some approaches to further improvement of the algorithm for either the general or the linear expression case. Most speed-up techniques that work for the general Dijkstra-type algorithms are directly applicable here: for a correctness argument it usually suffices to consider the product graph. Indeed, on the product graph, any source-destination shortest path results in a regular language-constrained shortest path for the original graph. Thus we only need to correctly "project" the improvement applied to Dijkstra's algorithm to get an improvement for the language-constrained version.

Our discussion here is brief because this work is still very much in progress and the full paper will contain more detail, including experimental results.

2.2.1 Bidirectional search

Dijkstra's algorithm in a point-to-point application typically runs faster when implemented so that the search is done simultaneously forward from the source and backward from the destination. In reasonably regular distributions of vertices in two dimensions, the improvement is clear: the forward search will explore roughly k^2 vertices to find a k-link shortest path, while the forward and backward searches are likely to meet when each has explored roughly $(k/2)^2$ vertices, thus indicating an expected halving of the number of explored vertices.

As long as the priority queue is used to store both the graph vertex and the NFA state, and the termination rule is defined carefully, we can use this to improve the performance. See also [JMN99].

2.2.2 Shortest-path containers

With the shortest-path containers approach [WW03], we determine in a preprocessing step for each edge $e = (u, v)$ the set of those nodes S that are reachable from u via e on a shortest path. Provided that a node embedding is given, each edge is assigned a geometric container (e.g., a bounding rectangle) containing S.

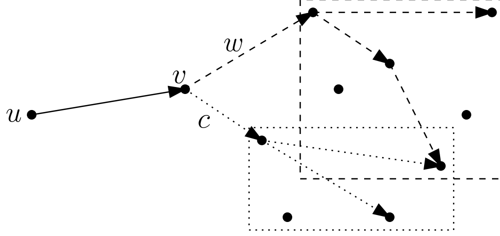

Figure 2: Refined shortest-paths containers: edge (u, v) has two associated containers, one for each of the labels (c and w) of outgoing edges of v.

During the search phase, an edge e being scanned can be discarded if the destination node d does not lie within the container associated with e.

We can apply the very same procedure also in the linear-language constraint case. For correctness, it suffices to verify that no edge forming part of a constrained shortest path is pruned: if we prune an edge e, then we know that there is no shortest path to d via e (with whatever labeling).

In the unimodal case, performance of the search phase is determined by the tradeoff between the complexity of the shortest-path containers (and hence running time for processing them) and their quality (depending on their shape, shortest-path containers may contain more or less false positives, i.e., nodes included in one container that do not lie on a shortest path with respect to the edge in question). In an experimental study [WW03], bounding rectangles turned out to yield best performance in general.

In the multimodal case, it may additionally occur that starting from a given edge, there is actually a shortest path to d but such that the current constraint is not fulfilled, which may damp the performance factor of the unimodal shortest-path containers.

We therefore suggest the following refinement. For each edge $e = (u, v)$, instead of one, several shortest-path containers may be computed, classified by the labels of the outgoing edges of v. For an illustration see Figure 2. During the search phase, one need respect only containers whose belonging *subsequent* label matches the constraint. This process could also be iterated in the sense that different containers are used depending on the labeling of *any number* of subsequent edges.

2.2.3 Bit-vectors

The general idea behind bit-vectors [MS+05] is similar to that of shortest-path containers: the input graph is divided up into multiple regions such that each node is attributed to exactly one of them. Each edge e is annotated with bit-vectors indicating for each region if any node of that region can be reached on a shortest path via e. Moreover, a recursive version of this approach is described in the same paper. Implementation of multimodal bit-vectors can hence be done similarly to shortest-path containers.

2.2.4 Multi-level technique

The multi-level technique relies on a preprocessing step where a decomposition of the input graph into several levels, induced by subsets—or selections—of the graph's node set, is computed and the graph is augmented with additional edges (cf. [HSW06]). There are two variants: basic multi-level graphs, where between nodes of the *same* selection new edges, so-called level edges, are introduced, and extended multi-level graphs, which additionally have so-called upward and downward edges, passing between certain nodes selected at *different* levels. New edges are assigned the length of a shortest path between the two nodes in the original graph. To find a shortest path between given nodes s and d, it suffices to examine a subgraph of the multi-level graph.

This technique can be extended to the linear-language-constraint case as follows. We assume that in practical settings, only a small number of constraints occur. Instead of one shortest-path length, assign each new edge the length of a constrained shortest path (or ∞ if there is no such path), for each subword—in the following also called *chunk*—of any constraint respected in our scenario. The subgraphs on which an actual

s-d-search takes place are the same in both the uni- and the multimodal cases.

During the search phase, proceed as follows: let $w = x_1^+ \cdots x_k^+$ be the constraint, and for each touched node keep the chunk of w that "has led" to that node. Let further be u the node being scanned by Dijkstra's algorithm, and assume that the portion $x_1^+ \cdots x_i^+$ has been processed for u, so $x_i^* \cdots x_k^+$ is the remaining subword to be considered. For each edge being scanned, pick as its length the shortest one from amongst all those associated with the chunks x_i^+, $x_i^+ x_{i+1}^+$, $\ldots$, $x_i^+ \cdots x_k^+$ and x_{i+1}^+, $x_{i+1}^+ x_{i+2}^+$, $\ldots$, $x_{i+1}^+ \cdots x_k^+$.

To summarize, with the multi-level model, new-added edges correspond to paths in the input graph; thus, the graph is just enriched with consistent information. For the multimodal case, this does still hold. However, for an actual gain in performance one has to ensure that enough additional information is provided: for the algorithm to be able to exploit precomputed long-distance edges independently of a given constraint, keep, for two specific nodes, not only one single shortest path but many shortest paths, subject to any constraint arising in our setting.

We now want to briefly outline a correctness proof. What we have to show is that a constrained shortest path in the subgraph used for computation is a constrained shortest path in the whole graph: (i) By [SWZ02], the subgraph contains all edges (genuine edges and new edges, where the latter correspond to subpaths in the original graph) such that a shortest s-d-path can be found. (ii) Each new edge $e = (u, v)$ being scanned is labeled correctly: from amongst all lengths of paths between u and v subject to any portion of constraint to be processed yet we pick the minimal one.

2.2.5 A* search (Sedgewick-Vitter heuristic)

If we do not insist on exact shortest paths, a very simple trick to speed up the algorithm by a great deal, is Sedgewick-Vitter [SV86] heuristic originally proposed for Euclidean shortest paths. The speedup results from a bias introduced in the search, which expands the set of examined vertices in the direction of the source-destination vector.

To ensure optimal shortest paths are found, one need 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 often 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.

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. In experiments we show how varying the amount of the bias affects the running time and quality of paths, and find a useful tradeoff.

3 Applications of our framework

We give just a couple of examples to illustrate types of problems solvable in our framework. There are many more problems that can be easily solved, including variants of turn complexity, counting constraints, k-similar path problems etc.

3.1 Applications of regular language constraints

3.1.1 Multimodal plans

In a simple multimodal network the edge-labels denote modes of travel allowed on the link. For example, streets may 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 the 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 $w \ldots wb \ldots bw \ldots w$.

3.1.2 Trip chaining

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.

4 Experimental study sketch

This section is still in a preliminary state. Some of the results described were reported in a previous paper [BB+02]. However, we expect the main strength of our full paper to be the comparison between that early special-case implementation (from TRANSIMS), and the new fully general code described above, together with several speed-ups. Our experiments test the following theses:

1. For all but largest instances of the shortest-path problem, Dijkstra's algorithm, even with relatively simple data structures such as the standard binary heap, can be quite efficient as the basis for an implementation when modern speed-up techniques are used.

2. As the intuition behind most speed-up techniques is geometric, they tend to work well in graphs structured like typical road networks, which are reasonably "regular" and almost hierarchical. Even with regular-language constraints, we get product graphs of almost-geometric road networks and simple NFAs may have a "layered" structure that retains enough geometry to allow goal-oriented speed-up techniques to work well. However, the usefulness of geometric techniques is reduced for more restrictive expressions, which may require the shortest path to "wiggle around" and not appear very efficient geometrically.

4.1 Early experiments with the TRANSIMS router

These experiments were run on a multi-modal transportation network representing the full road network and a full copmlement of scheduled transit in the city of Portland, OR.

To generate test instances (shortest-path queries), the Portland traffic network was divided into approximately 1200 traffic analysis zones (TAZs). From these a distance matrix was created by using the Euclidean distance between centers of TAZ pairs. Source and destination TAZs 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 the given TAZ, producing 5000 trips.

The TRANSIMS shortest path implementation allowed testing of our algorithm on real transportation networks in multimodal situations. In order to anchor theoretical research in realistic problems, TRANSIMS

uses *Case studies* (see [CS97] for details). The other case studies run for the TRANSIMS project included Dallas/Fort Worth (1997), with an early incomplete implementation of the code) and Chicago (2003).

In the basic Portland network, there are a total of 475 264 external nodes and 650 994 external links. The internal network adds nodes to many links to represent parking locations and includes copies of many links with different labels— for street segments that have sidewalks and allow pedestrian traffic, as well as a scheduled transit network laid on top of the existing road network. The network thus grows to over three million edges

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).

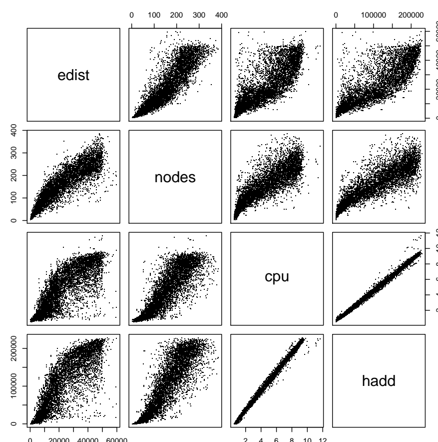

Figure 3: Plots of Euclidean origin-destination distance, trip duration, running time and total number of nodes added to the heap. The linear relation between the running time and the number of nodes added to the heap during the execution is obvious from the plot and also very clear from the algorithm 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.

We defer the analysis of distinctions between various modes of transportation, a more detailed multimodal study, and the time-dependent delay case to the full version of the paper.

4.2 Varying the Sedgewick-Vitter bias

We next 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, the speed-up between 0 and 0.15 is quite impressive.

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

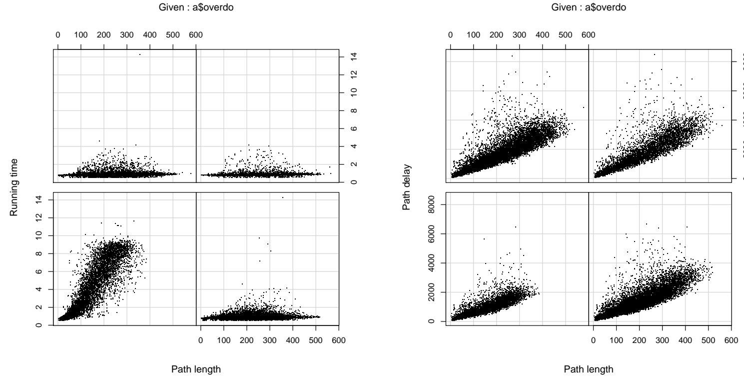

Figure 4: Running time and path delay plotted against the number of nodes on the path produces by the algorithm for four different values of `overdo` parameter (0,0.15,0.25,0.5). The "strange" horn-shape with `overdo`= 0 appears because the boundary of the network restricts the number of available vertices to be explored. One important thing to notice is that the running time appears linear in the other three plots. In fact, at first it appears *constant*, but a linear fit shows it just has a very low slope, as is to be expected. However, the quality of the paths output continues to decay with increased overdo.

a 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 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 geometrically direct 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 bias set to 0.15, 0.25 and 0.5. The average running times are practically equal, and a formal analysis of variance shows that we should not reject the hypothesis of equality of the running times with the bias at 0.15 and 0.25.

4.3 Experiments with more general constraints

4.4 Dijkstra's algorithm and simple regular expressions

We ran Dijkstra's algorithm (in both the standard and bi-directional version) to find shortest paths between pairs of points selected randomly from opposite quarters of the Phoenix metropolitan region street network (Figure 5). We randomly selected 30 source-destination pairs, each source from a region in the northwestern corner, and each destination from a region in the southeastern corner of the map. For each pair, we ran our algorithm, while constraining the set of feasible paths by using several different regular expressions. In what follows, we use symbols a_1, a_2, a_3 and a_4 to represent four different road classes specified in the TIGER/LINE data from which we derived the graph. These are:

1. limited-access highways (e.g. interstates—blue),
2. unlimited-access primary roads (e.g. state or county highways—red),
3. secondary roads and connectors—green, and
4. local, neighborhood and rural roads—not drawn to avoid clutter.

The expressions we used in this set of tests were:

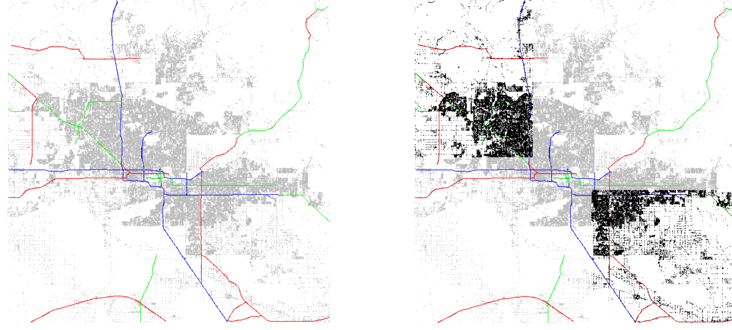

Figure 5: The Phoenix metropolitan area. Roughly 140000 vertices and 360000 edges. Blue, red and green represent limited-access highways, primary roads without limited access, and secondary and connecting roads, respectively. The local streets (a majority of all edges in the graph) are not drawn, but only edges, in order to reduce clutter. On the right, the northwest and southeast regions are marked in black. The source-destination pairs for some of our experiments are selected from these two regions.

1. $(a_1 \cup a_2 \cup a_3 \cup a_4)^*$, that is, any sequence of links.
2. $a_4^*(a_1 \cup a_2 \cup a_3)^* a_4^*$, that is any sequence of links that contains at most one contiguous subsequence of highway links.
3. a_4^*, that is, only local streets.

The running time in seconds as a function of the shortest path length in the number of edges shown in Figure 6.

We compared the performance of the straightforward generalization of Dijkstra's algorithm to the language-constrained case to the performance of a bidirectional version. Intuitively, the bidirectional version will examine only about half of the vertices before finding a source-destination path. In our experiments, this improvement can be observed, although the bidirectional algorithm usually examines somewhat more than half of the vertices that the standard Dijkstra does. The information on the number of vertices touched by the two algorithms is given in Table 7. Unfortunately, currently the running time of our bidirectional variant is not consistently much better than that of the standard algorithm. We believe that this can be improved by fine-tuning our implementation.

4.4.1 More complex expressions

Here, we describe experiments on a smaller network, but with more complex constraints. For this purpose, we selected a subset of the Phoenix metropolitan area street network corresponding roughly to the City of Tempe, and consisting of about 8000 vertices and about 20000 edges 8.

We use r, b and g, respectively, to represent the links drawn in red, blue and grey in the figure above. We sampled uniformly 100 sources and, independently, 100 destinations from the vertex set, and ran shortest path queries for these inputs.

The constraints we used in this set of tests were:

1. $(r \cup b \cup g)^*$, that is, no restriction on the path.
2. $(g \cup r(b \cup g) \cup b(r \cup g))^*$, that is, no consecutive red or blue links (and no paths end in red or blue).
3. $(g \cup b)^* r(g \cup b)^*$, that is, any path containing at most one red link.
4. $(g \cup r)^* b(g \cup r)^*$, that is, any path containing at most one blue link.

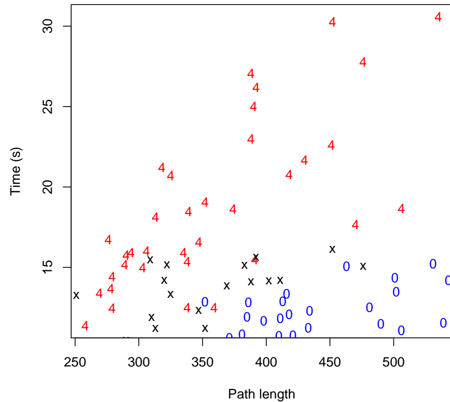

Figure 6: Running time as a function of shortest path length for three different expressions (expression 1 from above in black "X"s, expression 2 in red "4"s and expression 3 in blue "0"s). It appears that more complex an expression, the steeper is the dependence the of the running time on the length of the path found. Intuitively, this may follow from the fact that more complex expressions lead to a larger product graph. Since the edges that link different states of the NFA correspond to edges crossing between different copies of the network in the product graph, more complicated expressions also increase the geometric dimension of the graph and require larger neighborhoods to be explored during the search for the destination.

5. The constraints imposed by the NFAs of Figure 10.

The summary of the results for our experiments on the Tempe network is shown in Table 9. The numbers reported are the averages for the number of vertices touched by the algorithm (normally a good indicator of the running time, as seen in the data from TRANSIMS runs above), the length of the found shortest path in the number of edges, and the total running time itself in seconds. For the fifth expression, many of the queries resulted in no feasible path found. Therefore the average path length reported in the table is computed only over the queries that returned a feasible path. For the running time and the number of vertices touched, it seemed reasonable to report the average over the whole set, including the infeasible queries.

5 Conclusion

We have run preliminary experiments to study the performance of the formal-language constrained shortest path algorithm for various regular languages. We have also done basic comparisons between the running time of the basic Dijkstra's algorithm and the bidirectional version. The next steps we will undertake are to implement further speed-up techniques, such as the goal-directed search, shortest-path containers, bit-vectors and the multilevel technique. In a more complete version of the paper we will discuss further experiments that focus on the following issues.

- Comparisons in accuracy between different speed-up techniques when applied to multi-modal routing problems (some techniques will not yield exact shortest paths in certain settings).
- Comparisons between performance on explicit and implicit product graph representations.
- Scalability study for larger networks (the full US road network as provided by the Challenge participants contains road class information, which allows multimodal queries).

Instance	all streets		local streets		one highway	
	Dijkstra	bidirectional	Dijkstra	bidirectional	Dijkstra	bidirectional
1	136542	88714	132497	114040	231800	179049
2	124012	68450	136625	136649	272250	145058
3	101660	69425	114243	105807	158154	129738
4	102266	78502	114349	111342	165826	155419
5	86823	69375	106894	106802	140397	130830
6	90178	71383	105828	106804	137504	129580
7	136032	56725	133684	103748	237232	100175
8	120407	81738	118217	117659	190558	154325
9	78184	71924	102264	109694	112581	113749
10	88717	73510	102017	101554	139772	136730
11	104179	71052	104787	107110	163954	142742
12	71180	65680	98381	110828	108897	109133
13	91827	54181	104689	87851	141629	103872
14	126184	84515	126270	117553	167442	117507
15	64313	44266	89006	89618	102315	77890
16	80820	62166	109395	103963	125248	107446
17	126851	69106	127217	111788	206305	136686
18	138942	91644	134041	120924	243756	179138
19	77016	49781	94264	90900	119121	90978
20	115461	65597	119664	105111	181444	124820
21	107692	71941	114613	110735	145747	116941
22	115198	52736	121503	87446	184754	96989
23	133335	75296	132813	120352	200493	120718
24	128420	73994	127705	96358	156178	119894
25	74246	50398	93751	95987	116180	93601
26	68716	61306	96064	105179	109666	101849
27	139936	84494	134980	110492	267995	177760
28	132100	68875	132215	112295	216030	125327
29	86037	71879	112114	110810	132009	127194
30	85847	58420	106437	100830	133691	110736
Mean	104437.4	68569.1	114884.2	107007.6	166964.3	125195.8

Figure 7: The number of vertices touched by the two algorithms for three different NFAs in Phoenix.

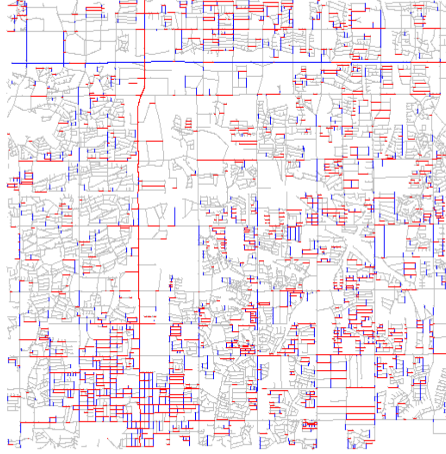

Figure 8: A part of the Phoenix Metro street network containing the city of Tempe. Roughly 8000 vertices and 20000 edges. Blue and red labels were assigned by an arbitrary decision to links directed almost due east-west (red) or almost due north-south (blue). A few exceptions are left over from the original labeling by road class. The visible squares form the basic development unit, and are almost exactly one mile by one mile in real life.

Expression and algorithm	Vertices touched	Path length	Time	NFA vertices	NFA edges
$(r \cup b\ cupg)^*$				1	3
Dijkstra	3935.65	50.34	0.4244626		
bidirectional	2612.98	50.34	0.3311927		
$(g \cup r)^* b (g \cup r)^*$				2	5
Dijkstra	6658.56	59.38	0.7077589		
bidirectional	4446.63	59.38	0.5598001		
$(g \cup b)^* r (g \cup b)^*$				2	5
Dijkstra	6226.12	58.63	0.6654075		
bidirectional	5307.41	58.63	0.6724628		
$(g \cup r(b \cup g) \cup b(r \cup g))^*$				3	7
Dijkstra	5687.46	60.51	0.577886		
bidirectional	5833.75	60.51	0.7233237		
$(g \cup rbg)^*(\lambda \cup r \cup rb)$				3	4
Dijkstra	1734.69	57.54^*	0.1726844		
bidirectional	3928.78	57.54^*	0.4745679		

Figure 9: Summary of results for five different expressions on a smaller graph (Tempe). The last constraint resulted in many infeasible source-destination pairs, and so we report the average path length over the subset of feasible queries.

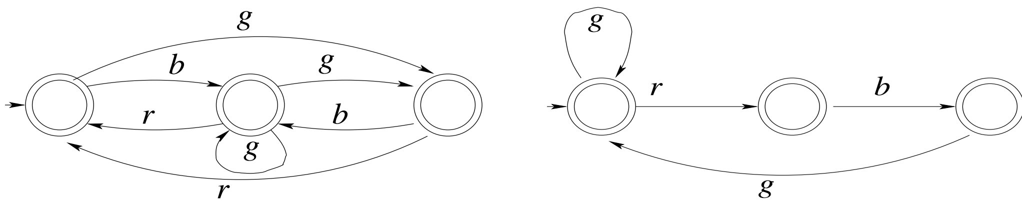

Figure 10: Two NFAs used to restrict the set of feasible paths. For the NFA on the left, only for 3 of the 100 random source-destination pairs does there exist a path in its language. The NFA on the right is much less restrictive.

References

[BB+02] C. Barrett, K. Bisset, R. Jacob, G. Konjevod and M. Marathe *Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router*, Proc. ESA 2002, LNCS 2461, pp. 126–138, 2002.

[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.

[D59] E. W. Dijkstra, *A note on two problems in connexion with graphs,* Numerische Mathematik, Vol. 1 (1959), S. 269–271

[HSW06] M. Holzer, F. Schulz, and D. Wagner, "Engineering multi-level overlay graphs for shortest-path queries", *Proc. 8th Workshop on Algorithm Engineering and Experiments*, Vol. 129, pp. 156–170, SIAM, 2006.

[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.

[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.

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

[MS+05] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm, "Partitioning graphs to speed up Dijkstra's algorithm", *Proc. 4th International Workshop on Efficient and Experimental Algorithms*, Vol. 3503, pp. 189–202, Springer, 2005.

[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.

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

[SWZ02] F. Schulz, D. Wagner, and C. Zaroliagis, "Using multi-level graphs for timetable information in railway systems", *Proc. 4th Workshop on Algorithm Engineering and Experiments*, Vol. 2409, pp. 43–59, Springer, 2002.

[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.

[WW03] D. Wagner and T. Willhalm, "Geometric speed-up techniques for finding shortest paths in large sparse graphs", *Proc. 11th European Symp. Algorithms*, Vol. 2832, pp. 776–787, Springer, 2003.

[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.

[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.