Algorithms for Routing with Multiple Constraints

Anuj Puri and Stavros Tripakis

December 16, 1999

Abstract

In this paper, we study the problem of routing under multiple constraints. We consider a graph where each edge is labelled with a cost and a delay. We then consider the problem of finding a path from a source vertex to a destination vertex such that the sum of the costs on the path satisfy the cost constraint and the sum of the delays satisfy the delay constraint. We study the complexity of this problem and then present three different algorithms for solving the problem. These algorithms have varying levels of complexity and solve the problem with varying degrees of accuracy. We present an implementation of these algorithms and discuss their performance on different graphs.

1 Introduction

Routing data from one node to another is among the most basic problems in computer networking. A model for such a problem is a graph where an edge in the graph represents a physical link in the network. Associated with each edge is its length. This length could represent the delay of the physical link, or its cost, or a summary of various properties of the link. The routing problem is to find a path from the source to the destination of minimum length.

This formulation requires us to summarize all properties of a link with a single number, or to focus only on a single property at the expense of others. For example, we may have a choice between a low cost link which has a high delay (such as a satellite link), or a high cost link with a low delay (such as a fiber optic link). It is not clear how to summarize these links with a single number in the routing problem.

In this paper, we present a more general formulation of the problem. We model a physical link with an edge in the graph that is labelled with two numbers: a delay and a cost. We are given a delay constraint D and a cost constraint C. Our objective is to find a path from the source to the destination such that the sum of all delays on the path is less than D, and the sum of all costs is less than C. We first show that this problem is NP-hard. We then present three different algorithms for solving the problem. The first algorithm is a pseudo-polynomial time algorithm which solves the problem exactly in time $O(|V||E|\mathsf{min}\{C, D\})$ where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. The algorithm either reports back with a path satisfying the constraints or states that no such path exists. The second algorithm solves the problem approximately but with an error of at most ϵ. That is, either it states that no path satisfying the constraints exists, or it finds a path such that the sum of costs on the path is at most $C(1+\epsilon)$, and the sum of delays is at most $D(1+\epsilon)$. The complexity of this algorithm is $O(|V|^2|E|(1+\frac{1}{\epsilon}))$. The third algorithm finds a path with an error of at most $\epsilon = 1$. This algorithm requires a solution of the shortest path problem on the given graph. Although most of the paper is focused on dealing with two constraints, The first two algorithms generalize in a straight forward manner to more than two constraints.

In Section 2, we define our problem more formally and introduce our notation. In Section 3, we show the problem is NP-Complete. In Section 4, we present a pseudo-polynomial time algorithm for the problem. In Section 5, we present a polynomial time approximation scheme. In Section 6, we present an algorithm based on solving the shortest path problem on the graph. In Section 7, we present a linear programming solution to a relaxed problem. In Section 8 we discuss other possible extensions. Secion 9 discusses the performance results for the different algorithms and Section 10 is the conclusion. In the appendix, we describe the details of our software and how it can be used.

Relationship to other work

The routing problem with more than one constraint seems to have been studied by several researchers. It seems to be well known that the problem is NP-Complete [3, 2, 1]. An explicit proof of this is provided in [4]. In [3], a pseudo-polynomial time algorithm is presented for exactly solving the problem. This algorithm is similar to the algorithm presented in Section 4. Although [3] states that the complexity of the algorithm is $O(|V|^5\mathsf{max}\{C, D\}log(|V|\mathsf{max}\{C, D\}))$. By a more careful analysis and

using the data structures in a more clever manner, we can show the complexity of our algorithm is $O(|V||E|\mathsf{min}\{C,D\})$. In [3], an approximation algorithm that solves the problem with approximation error $\epsilon = 1$ using the shortest path algorithm is also presented. Although this is similar to our algorithm in Section 6, our algorithm in general will perform better because we solve a series of shortest path problems, each obtaining a better solution than the last one. In [2], several algorithms are presented for approximating the solution to the problem. Although the author restricts to acyclic graphs, extensions to general graphs should be straight forward. The complexity of the two approximation algorithms are $O(loglogB(\frac{|V||E|}{\epsilon} + loglogB))$ and $O(|E|\frac{|V|^2}{\epsilon}log(\frac{n}{\epsilon}))$ where ϵ is the error of the approximation and $B = \mathsf{max}\{C,D\}$. Our approximation algorithm in Section 5 has complexity $O(|V|^2|E|(1+\frac{1}{\epsilon})$. The algorithms also use somewhat different techniques. Our algorithm is essentially a generalization of the Bellman-Ford algorithm where we keep track of errors during the iteration.

Although, several algorithms have been proposed for solving the routing problem with multiple constraints, there seems to be no results available about the actual implementation or the performance of these algorithms. In our work, we present several new ideas for solving the problem, a complete implementation of these ideas and a comparison of the performance of these algorithms.

2 Problem Formulation

We consider a directed graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. An edge $e \in E$ is $e = (v, w, c, d)$ where the edge goes from v to w, and has delay $\mathsf{delay}(e) = d$ and $\mathsf{cost}(e) = c$. We write this as $u \stackrel{(c,d)}{\longrightarrow} v$. When there is no confusion, we may also write the edge as (v, w) and say the edge is labelled with (c, d).

A path is $p = v_1 \stackrel{(c_1,d_1)}{\longrightarrow} v_2 \stackrel{(c_2,d_2)}{\longrightarrow} v_3 \stackrel{(c_3,d_3)}{\longrightarrow} \cdots \stackrel{(c_n,d_n)}{\longrightarrow} v_{n+1}$. The cost of a path is $\mathsf{cost}(p) = \sum_{i=1}^{n} c_i$ and its delay is $\mathsf{delay}(p) = \sum_{i=1}^{n} d_i$.

Given a path p and cost constraint C and delay constraint D, we say p is *feasible* provided $\mathsf{cost}(p) \leq C$ and $\mathsf{delay}(p) \leq D$. The problem of *routing under two constraints* is, given $G = (V, E)$, cost constraint C and delay constraint D, a *source* node $s \in V$ and a *destination* node $t \in V$, find a feasible path p from s to t, or decide that no such path exists.

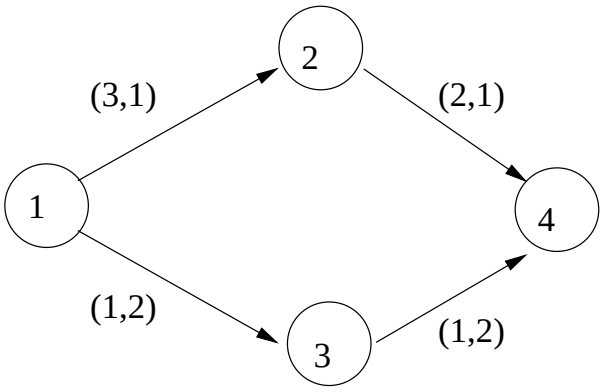

Figure 1: A Simple Network

Example 2.1 *Consider the graph of Figure 1. Each edge is labelled with* (c, d) *where* c *is the cost of the edge and* d *is the delay of the edge. For example, the edge from vertex 1 to vertex 2 has cost 3 and delay 1. Suppose the source vertex is 1, the destination vertex is 4, the cost constraint is* $C = 5$ *and the delay constraint is* $D = 2$*. Then, the path* $1 \xrightarrow{(3,1)} 2 \xrightarrow{(2,1)} 4$ *is feasible, whereas* $1 \xrightarrow{(1,2)} 3 \xrightarrow{(1,2)} 4$ *is not (since it violates the delay constraint).*

The reader can check that if $C = 4$ *and* $D = 3$*, then there is no feasible path.*

Rather than checking to see if a graph has a feasible path, it is sometimes useful to minimize the following objective function

$$M(p) = \mathsf{max}\{\frac{\mathsf{max}\{\mathsf{cost}(p), C\}}{C}, \frac{\mathsf{max}\{\mathsf{delay}(p), D\}}{D}\}$$

Observe that for any path p, $M(p) \geq 1$ and $M(p) = 1$ iff p is feasible. But even if a feasible path does not exist or is hard to find, by trying to minimizing $M(p)$ we can get a path that comes "close" to satisfying the constraints.

Formally, we define the error of a path p as

$$\mathsf{error}(p) = \frac{M(p) - M(p^*)}{M(p^*)}$$

where p^* is the path which minimizes M (in case more than one paths minimize M, we pick p^* arbitrarily among them, since the minimal value $M(p^*)$ is the same for all of them).

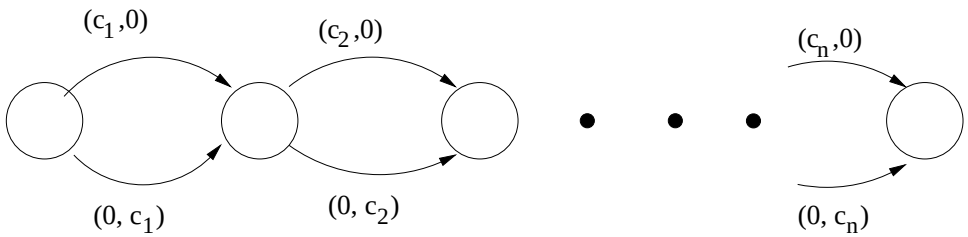

Figure 2: Graph obtained from the knapsack problem

Notice that $\mathsf{error}(p) \geq 0$ and $\mathsf{error}(p) = 0$ iff p is feasible. Also note that if $\frac{\mathsf{cost}(p)-C}{C} \leq \epsilon$ and $\frac{\mathsf{delay}(p)-D}{D} \leq \epsilon$, then $\mathsf{error}(p) \leq \epsilon$. Indeed, the two above conditions imply that $M(p) \leq 1+\epsilon$ and, since $M(p^*) \geq 1$, we get $\mathsf{error}(p) \leq \epsilon$.

In case it is too difficult to find p^*, we look for a path p for which $\mathsf{error}(p)$ is small. We will next consider the complexity of finding a feasible path, and algorithms for finding a feasible path and for finding paths for which $\mathsf{error}(p)$ is small.

3 Complexity

We show that the routing problem with two constraints is NP-Complete.

Theorem 3.1 *The routing problem with two constraints is NP-Complete.*

Proof: We will provide a reduction from the knapsack problem. Recall that in the knapsack problem, we are given positive integers $c_1, c_2, \ldots, c_n$, and N, and the objective is to find a subset $S \subset \{1, \ldots, n\}$ such that $\sum_{i \in S} c_i = N$.

From the knapsack problem, we construct a graph with vertices $\{1, \ldots, \mathrm{n}\}$. There are two edges from vertex i to vertex $i+1$: edge $(i, i+1, c_i, 0)$ and edge $(i, i+1, 0, c_i)$. Figure 2 shows the scenario. Our objective is to find a path from vertex 1 to vertex n with cost constraint N and delay constraint $\sum_{i=1}^{n} c_i - N$. It is easy to check that there is a path that satisfies the constraints iff there is a solution to the knapsack problem. ■

4 An $O(|V| \cdot |E| \cdot \mathsf{min}\{C, D\})$ pseudo-polynomial algorithm

In this section, we propose an algorithm for the problem of routing under two constraints with worst-case complexity $O(|V| \cdot |E| \cdot \mathsf{min}\{C, D\})$. That is, the algorithm is polynomial on the size of the graph (quadratic on the number of nodes and linear on the number of edges), but also linearly depends on the smaller of the bounds C and D. Therefore, it is a *pseudo*-polynomial algorithm.

Let us begin by making a safe hypothesis. Consider a graph $G = (V, E)$ and let $\mathsf{cost}_{max} = \mathsf{max}\{c \mid (\_, \_, c, \_) \in E\}$ and $\mathsf{delay}_{max} = \mathsf{max}\{d \mid (\_, \_, d, \_) \in E\}$ be the maximum cost and delay associated with any edge of G. Now, assume that $n \cdot \mathsf{cost}_{max} \leq C$. Then, given $u, v \in V$, there exists a feasible path from u to v iff there exists a path p from u to v such that $\mathsf{delay}(p) \leq D$. To see this, observe that if there is a path p from u to v such that $\mathsf{delay}(p) \leq D$, then there is a *simple* path (i.e., with no cycles) from u to v. Assuming p to be simple, p has length at most $|V|$, thus, $\mathsf{cost}(p) \leq n \cdot \mathsf{cost}_{max} \leq C$, which implies that p is feasible. The inverse direction is trivial.

Given this observation, finding a feasible path in G from u to v comes down to finding the smallest-delay path from u to v, that is, the path p that minimizes $\mathsf{delay}(p)$. This can be easily done using a shortest-path algorithm, with cost $O(|V| \cdot |E|)$. Since this is less than $O(|V| \cdot |E| \cdot \mathsf{min}\{C, D\})$, this case is not interesting. The case where $n \cdot \mathsf{delay}_{max} \leq D$ is symmetric.

So, from now on we assume that $n \cdot \mathsf{cost}_{max} > C$ and $n \cdot \mathsf{delay}_{max} > D$. We also assume that the greatest common divisor of $\{C, \mathsf{cost}(e) \mid e \in E\}$ is 1, and similarly for the delays (otherwise we could just divide all costs/delays by their GCD, without affecting the problem).

Informally, the algorithm works as follows. For each node w, we compute a set of cost-delay pairs F_w. Each $(c, d) \in F_w$ will represent the cost and delay of a possible path from w to the destination node v. To keep the size of F_w manageable, we eliminate from F_w all elements corresponding to infeasible paths (i.e., all (c, d) such that $c > C$ or $d > D$). Moreover, we eliminate from F_w all redundant elements, that is, all elements which have greater both cost and delay from some other element. Let us make these more precise below.

Cost-delay sets. A *cost-delay set* for a node w is a set $F_w \subseteq \mathsf{N} \times \mathsf{N}$. An element (c, d) of F_w is called *infeasible* if either $c > C$ or $d > D$. An element (c, d) of F_w is called *redundant* if there exists a different $(c', d') \in F_w$ such that $c' \leq c$ and $d' \leq d$.

A cost-delay set F is said to be *minimal* if it contains no infeasible or redundant elements. The following properties hold (assuming C and D fixed):

Proposition 4.1 *If F is minimal, then $|F| \leq \mathsf{min}\{C, D\}$. To every cost-delay set F corresponds a unique greatest minimal subset $F' \subseteq F$.*

We write $\mathsf{min}(F)$ to denote the greatest minimal subset of F.

Figure 3 displays the typical structure of a cost-delay set and its minimal. Black and grey bullets are infeasible and redundant elements, respectively.

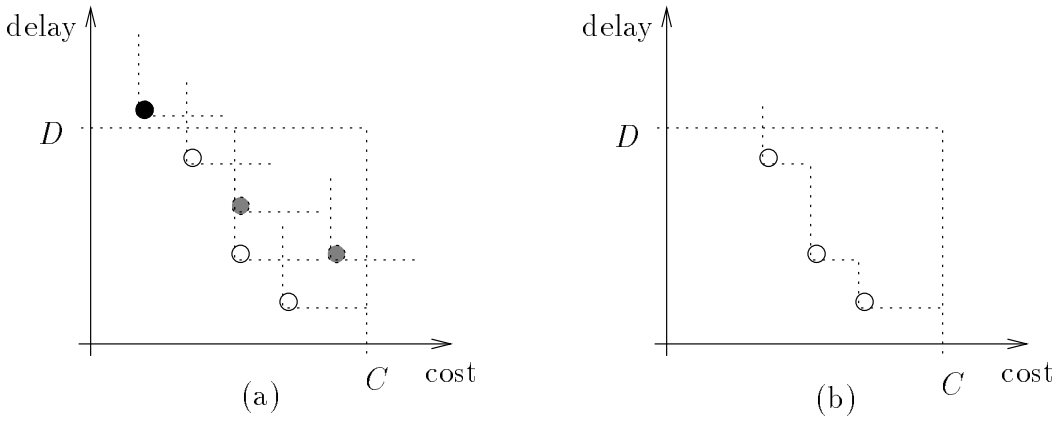

Figure 3: A cost-delay set (a) and its minimal (b)

The algorithm we present below works with minimal cost-delay sets. It is unnecessary to consider non-minimal sets, since we are interested in an exact algorithm (that is, which computes only feasible solutions) and redundant points represent "worse" paths in both cost and delay.

Moreover, minimal sets admit an efficient canonical representation in terms of sorted lists. Consider a minimal set $F = \{(c_1, d_1), (c_2, d_2), ..., (c_n, d_n)\}$ and assume, without loss of generality, that $c_1 \leq c_2 \leq \cdots \leq c_n$. Then, $d_1 \geq d_2 \geq \cdots \geq d_n$ must hold, otherwise there would be at least one redundant element in F. Consequently, F can be represented as the list $(c_1, d_1)\ (c_2, d_2)\ \cdots\ (c_n, d_n)$, sorted using cost as the

"key". This representation is canonical in the sense that two minimal sets F_1, F_2 are equal iff their list representations are identical.

In the algorithm that we present below, we use two operations on cost-delay sets, namely, set union and translation with respect to a vector $(c, d) \in \mathsf{N}^2$. We now consider these operations and discuss how they can be implemented using the sorted-list representation in order to preserve minimality.

Given minimal (i.e., feasible and non-redundant) F_1, F_2, the union $F_1 \cup F_2$ is always feasible, but not necessarily non-redundant. In order to compute $F = \mathsf{min}(F_1 \cup F_2)$ directly from the list representations L_1, L_2 of F_1, F_2, we can use a simple modification of a usual merge-sort algorithm on lists. The latter takes as input L_1, L_2 and produces L, the list representation of F. In order to guarantee the absence of redundant points in L, it compares at each step the heads (c_1, d_1) and (c_2, d_2) of (the remaining parts of) L_1, L_2. If $c_1 \leq c_2$ and $d_1 \leq d_2$ then (c_2, d_2) is redundant and is skipped. If $c_2 \leq c_1$ and $d_2 \leq d_1$ then (c_1, d_1) is skipped. Otherwise, the pair with the smallest c_i is inserted in L and the head pointer move one element ahead in the corresponding list L_i. It is easy to see that this algorithm is correct. The cost of the algorithm is $n_1 + n_2$, where n_i is the length of L_i. Therefore, from proposition 4.1, the worst-case complexity of computing the union of cost-delay sets is $O(\mathsf{min}\{C, D\})$.

Translation is defined on a cost-delay set F and a pair $(c, d) \in \mathsf{N}^2$:

$$F + (c, d) \stackrel{\text{def}}{=} \{(c' + c, d' + d) | (c', d') \in F\}$$

That is, $F + (c, d)$ is the set obtained by translating all points in F by (c, d) (equivalently, $F + (c, d)$ is the Minkowski sum of F and $\{(c, d)\}$). If F is minimal, then $F + (c, d)$ will be non-redundant, however, it may contain infeasible points. These can be easily eliminated, however, and this can be done while building the list L' for $\mathsf{min}(F + (c, d))$: the list of F is traversed, adding (c, d) to each of its elements, (c_i, d_i); if $c_i + c \leq D$ and $d_i + d \leq D$ then $(c_i + c, d_i + d)$ is inserted at the end of L', otherwise it is infeasible and it is skipped. At the end L' will be sorted by cost. The complexity of translation is $O(\mathsf{min}\{C, D\})$.

The algorithm. The algorithm iteratively computes the (minimal) cost-delay sets of all nodes in the graph. Let F_w^j denote the cost-delay set for node w at iteration j. Initially, all nodes have empty cost-delay sets, $F_w^0 = \emptyset$, except for the destination node v, the cost-delay set of which is $F_v^0 = \{(0, 0)\}$. At each iteration, each node updates its cost-delay set with respect to all its successor nodes. Computation stops when no

cost-delay set gets updated any more. We now present the operations performed at each iteration at each node w.

Let $w_1, ..., w_n$ be the successor nodes of w, that is, $w \xrightarrow{(c_i,d_i)} w_i$, for $i = 1, ..., n$ (note that $w_1, ..., w_n$ might not be distinct). Then, the cost-delay set of w at iteration $j+1$ will be:

$$F_w^{j+1} \quad = \quad \mathsf{min}\Big(F_w^j \cup \bigcup_{i=1}^{n} (F_{w_i}^j + (c_i, d_i))\Big) \tag{1}$$

That is, we add to the possible cost-delay values for w all values obtained by taking an edge to some successor node w_i, and then continuing with a possible cost-delay value for w_i.

The following proposition proves termination and correctness of the algorithm.

Proposition 4.2 *(Termination) The updating of the cost-delay sets will stabilize after at most $|V|$ iterations, that is, for any node w, $F_w^{|V|+1} = F_w^{|V|}$. (Correctness) A feasible path from w to v exists iff $F_w^{|V|} \neq \emptyset$. For any $(c,d) \in F_w^{|V|}$, there exists a path p from w to v such that $\mathsf{cost}(p) = c$ and $\mathsf{delay}(p) = d$.*

Worst-case complexity of the algorithm. Proposition 4.2 implies that the algorithm stops after at most $|V|$ iterations. At each iteration, the cost-delay set of each node is updated with respect to all its successor nodes. Thus, there is at most $|E|$ updates at each iteration. Each update involves a translation and a union, both of which have complexity $O(\mathsf{min}\{C, D\})$. Therefore, the overall worst-case complexity of the algorithm is $O(|V| \cdot |E| \cdot \mathsf{min}\{C, D\})$.

Incorporating routing information. As defined, cost-delay sets do not contain any routing information, that is, at the end of the algorithm, we know that a point in F_w represents the cost-delay value of a possible feasible path from w to v, but we do not know which path. This information is easy to incorporate, at the expense of associating to each $(c,d) \in F_w$ the edge $e = (w, w_1, c', d')$ and a pointer to the element $(c_1, d_1) \in F_{w_1}$, from which (c,d) was generated. The edge and (c_1, d_1) element are unique, and come from the operation $F_w \cup (F_{w_1} + (c', d'))$. The time complexity of the algorithm is not affected by the extra information. In order to reconstruct the path from w with cost-delay (c,d) we follow the edge e to w_1, then look for the path from w_1 with cost-delay (c_1, d_1), and so on.

5 A bounded-error approximative algorithm

In this section we give an approximative algorithm for the problem of routing under two constraints. The algorithm is approximative in the sense that, it might not yield a feasible path, even if such a path really exists. However, the error in the path p returned by the algorithm can be bounded: $\mathsf{error}(p) \leq \epsilon$, where ϵ is an input parameter. The algorithm has worst-case complexity $O(|V|^2 \cdot |E| \cdot (1 + \frac{1}{\epsilon}))$, which implies that it is worth using only when $|V|$ is (much) smaller than $\frac{\epsilon}{1+\epsilon} \cdot \mathsf{min}\{C, D\}$. Otherwise, the algorithm of section 4, being exact and less expensive, would be preferable. In the rest of this section we assume that $|V| < \frac{\epsilon}{1+\epsilon} \cdot \mathsf{min}\{C, D\}$.

Minimal-distance cost-delay sets. The approximative algorithm is similar to the one of section 4, with the additional fact that it eliminates elements of cost-delay sets which are "too close" to some other element. More formally, for $(c_1, d_1), (c_2, d_2) \in \mathsf{N}^2$, define:

$$||(c_1, d_1), (c_2, d_2)|| \stackrel{\text{def}}{=} \mathsf{max}\{|c_1 - c_2|, |d_1 - d_2|\}$$

Then, a cost-delay set F is said to have *minimal distance* δ iff for all distinct $(c_1, d_1), (c_2, d_2) \in F$, $||(c_1, d_1), (c_2, d_2)|| \geq \delta$.

Given a cost-delay set F and some $\delta \geq 2$, we would like to find a subset $F' \subseteq F$, such that:

1. F' has minimal distance δ, and
2. for all $x \in F - F'$, there exists $y \in F'$ such that $||x, y|| < \delta$.

Condition 2 ensures that no elements of F are dropped unnecessarily (were condition 2 to be missed, the trivial subset $F' = \emptyset$ would satisfy condition 1). A subset $F' \subseteq F$ satisfying the above conditions is called a *maximal δ-distance subset of F*. In general, there may be more than one maximal δ-distance subsets of a given F (any one of them is good for our purposes). We now give a procedure to compute, given F, a maximal δ-distance subset $F' \subseteq F$.

The procedure takes as input the list representation L of F and generates as output the list representation L' for $\mathsf{min\_dist}_\delta(F)$. Assume $L = (x_1, ..., x_n)$. Initially, $L' = (x_1)$. Let y denote the last element of L', at each point during the execution of the procedure. For each $i \geq 2$, if $||x_i, y|| \geq \delta$ then x_i is appended at the end of L' and y is updated to x_i, otherwise, x_i is skipped. It can be shown that the list built

that way represents a legal δ-distance subset of F. From now on, we denote this set by $\mathsf{min\_dist}_\delta(F)$.

Definition 5.1 *We define the* step error, δ_ϵ, *to be* $\frac{\min C,D \cdot \epsilon}{|V|}$.

The algorithm. The approximative algorithm is obtained by the algorithm of section 4 by the following modification. Given $\epsilon \in [0,1]$, instead of keeping a minimal set F_w for each node w, we keep a set B_w such that:

1. B_w has no redundant elements,
2. for each $(c,d) \in B_w$, $c \leq (1+\epsilon) \cdot C$, $d \leq (1+\epsilon) \cdot D$ (that is, the feasibility region is extended by $(\epsilon \cdot C, \epsilon \cdot D)$,
3. B_w has minimal distance δ_ϵ.

That is, in the approximative algorithm, the fix-point equations are as follows:

$$B_w^{j+1} \quad = \quad \mathsf{min\_dist}_{\delta_\epsilon}\Big(\mathsf{min}\Big(B_w^j \cup \bigcup_{i=1}^{n} (B_{w_i}^j + (c_i, d_i))\Big)\Big) \tag{2}$$

As in the case of the exact algorithm, termination of the approximative algorithm is ensured in $|V|$ steps.

Proposition 5.1 *Consider a graph G, nodes u, v of G, and cost-delay constraints C, D. Then, for given ϵ:*

(1) If $B_u = \emptyset$ at the end of the approximative algorithm, then no feasible path from u to v exists.

(2) If $B_u \neq \emptyset$, then for each $(c,d) \in B_u$, there exists a path p from u to v such that $\mathsf{cost}(p) = c$, $\mathsf{delay}(p) = d$ *and* $\mathsf{error}(p) \leq \epsilon$.

Proof (sketch):

Let w be a node and F_w, B_w be the final cost-delay sets computed for w by the exact and approximative algorithms, respectively. The result is based on the fact that, for any $(c,d) \in F_w$, there exists $(c',d') \in B_w$, such that $||(c,d),(c',d')|| \leq |V| \cdot \delta_\epsilon$.

This is because at most δ_ϵ "error" accumulates at each step of the algorithm, when eliminating pairs during the min_dist operation.

By definition of δ_ϵ, we have that $||(c,d),(c',d')|| \leq \mathsf{min}\{C,D\} \cdot \epsilon$. Then, assuming (c,d) to be the cost and delay of an optimal path p^* and (c',d') the cost and delay of a path p computed by the approximative algorithm, it is easy to prove that $\mathsf{error}(p) \leq \epsilon$. For (1), notice that if p^* is feasible then $c' \leq (1+\epsilon)\cdot C$ and $d' \leq (1+\epsilon)\cdot D$, This means that (c',d') is indeed "inside" the extended feasibility region, thus, is not eliminated from B_w during the approximative algorithm. ■

Worst-case complexity. The only difference from the algorithm of section 4 is in the worst-case size of the cost-delay sets B_w. Since the latter have minimal distance δ_ϵ and are bounded by the feasibility region $((1+\epsilon)\cdot C, (1+\epsilon)\cdot D)$, we have $|B_w| \leq \frac{(1+\epsilon)\cdot \min\{C,D\}}{\delta_\epsilon}$. By definition of δ_ϵ, we get $|B_w| \leq \frac{(1+\epsilon)}{\epsilon}|V|$. The union, translation and min_dist operations can be implemented using sorted lists to represent the sets B_w (canonicity is not affected by minimal distance). The cost of the operations is, as previously, linear on the size of the lists, which yields an overall worst-case complexity of $O(|V|^2 \cdot |E| \cdot (1+\frac{1}{\epsilon}))$.

6 Satisfying constraints by using the shortest path algorithm

In this section, we consider an algorithm for finding a path which satisfies the two constraints by using the shortest path algorithm. Our objective will be to use the shortest path algorithm to find a path p which minimizes $M(p)$.

For the rest of this section we assume that we have normalized the costs and delays by dividing the costs by C and the delays by D. Figure 4 shows the figure from Example 2.1 where the cost constraint is 5 and the delay constraint is 2.

A path is feasible in the new graph if $\mathsf{cost}(p) \leq 1$ and $\mathsf{delay}(p) \leq 1$. Note that a path is feasible in the new graph iff it was feasible in the original graph. Furthermore, $M(p)$ is the same in both graphs.

To find a path satisfying two constraints by using the shortest path algorithm, we choose an $0 \leq \alpha \leq 1$ and replace the cost c and the delay d associated with an edge with the weight $\alpha c + (1-\alpha)d$. We then use the shortest path algorithm to find

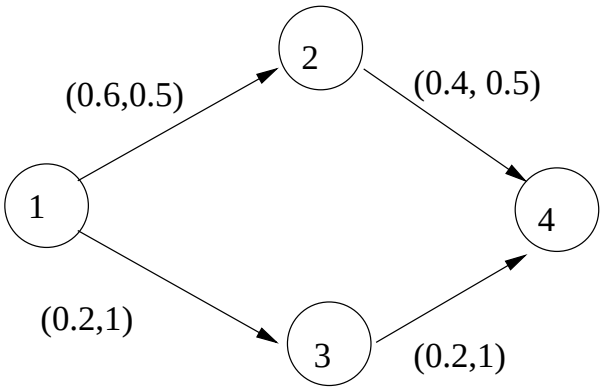

Figure 4: A network with normalized costs and delays

a path with the smallest weight. We refer to this path as $SP(G, \alpha)$. As the next lemma shows, $p = SP(G, \alpha)$ has an error $Error(p)$ of at most 1 for $\alpha = \frac{1}{2}$.

Lemma 6.1 *For a graph $G = (V, E)$, $M(p^*) \leq M(SP(G, \frac{1}{2})) \leq 2M(p^*)$ where p^* is the path which minimizes M.*

Proof: Compute $p = SP(G, \frac{1}{2})$. If $M(p) = 1$, then clearly $M(p) \leq 2M(p^*)$. So assume $M(p) > 1$. Then

$$M(p) \leq \mathsf{cost}(p) + \mathsf{delay}(p) \leq \mathsf{cost}(p^*) + \mathsf{delay}(p^*) \leq 2M(p^*)$$

■

The previous lemma shows that by choosing $\alpha = \frac{1}{2}$, we can obtain a path p with an error of $\mathsf{error}(p) \leq 1$. We now present an algorithm that minimizes $M(SP(G, \alpha))$ by choosing the appropriate α.

The algorithm works by binary search: say we know that the optimal value of α lies in the interval $[l, u]$; we find $p = SP(G, \alpha)$ for $\alpha = \frac{l+u}{2}$; if $\mathsf{cost}(p) \leq \mathsf{delay}(p)$, we eliminate the interval $(\frac{l+u}{2}, u]$ from consideration, otherwise, we eliminate $[l, \frac{l+u}{2})$. The algorithm terminates when $SP(G, l) = SP(G, u)$.

The reason that half of the interval can be eliminated follows from the following lemma.

Lemma 6.2 *Suppose $p = SP(G, \alpha)$ and $\mathsf{cost}(p) \leq \mathsf{delay}(p)$. Then for $\alpha' \geq \alpha$ and $p' = SP(G, \alpha')$, $\mathsf{cost}(p') \leq \mathsf{cost}(p)$ and $\mathsf{delay}(p') \geq \mathsf{delay}(p)$.*

Proof: There are four cases:

1. $\mathsf{cost}(p') > \mathsf{cost}(p)$ and $\mathsf{delay}(p') > \mathsf{delay}(p)$.
2. $\mathsf{cost}(p') < \mathsf{cost}(p)$ and $\mathsf{delay}(p') < \mathsf{delay}(p)$.
3. $\mathsf{cost}(p') > \mathsf{cost}(p)$ and $\mathsf{delay}(p') < \mathsf{delay}(p)$.
4. $\mathsf{cost}(p') \leq \mathsf{cost}(p)$ and $\mathsf{delay}(p') \geq \mathsf{delay}(p)$.

Case 1 is not feasible because then path p improves on $p' = SP(G, \alpha')$. Case 2 is not feasible because then path p' improves on $p = SP(G, \alpha)$. Case 3 is not feasible because $\alpha\mathsf{cost}(p') + (1-\alpha)\mathsf{delay}(p') \geq \alpha\mathsf{cost}(p) + (1-\alpha)\mathsf{delay}(p)$ and $(\alpha' - \alpha)\mathsf{cost}(p') + (\alpha - \alpha')\mathsf{delay}(p') > (\alpha' - \alpha)\mathsf{cost}(p) + (\alpha - \alpha')\mathsf{delay}(p)$, and hence $\alpha'\mathsf{cost}(p') + (1 - \alpha')\mathsf{delay}(p') > \alpha'\mathsf{cost}(p) + (1 - \alpha')\mathsf{delay}(p)$ — a contradiction since $p' = SP(G, \alpha')$. Therefore Case 4 is the only feasible option. ■

Now assume we found $p = SP(G, \alpha)$ and $\mathsf{delay}(p) > 1$ and $\mathsf{cost}(p) \leq \mathsf{delay}(p)$. Then from Lemma 6.2, for $\alpha' \geq \alpha$, for $p' = SP(G, \alpha')$, $\mathsf{cost}(p') \leq \mathsf{cost}(p)$ and $\mathsf{delay}(p') \geq \mathsf{delay}(p)$. Therefore $M(p') \geq M(p)$, and hence the interval $(\alpha, u]$ can be eliminated from consideration. By similar reasoning, if $\mathsf{delay}(p) \leq \mathsf{cost}(p)$, then the interval $[l, \alpha)$ can be eliminated.

Here is a more formal statement of the algorithm:

Algorithm to find α to minimize $M(SP(G, \alpha))$:

$l = 0$
$u = 1$
$p_l = SP(G, l)$
$p_u = SP(G, u)$
Repeat
    $\alpha = \frac{l+u}{2}$
    $p_\alpha = SP(G, \alpha)$
    if ($\mathsf{cost}(p) \leq \mathsf{delay}(p)$)
        $l = \alpha$
        $p_l = p_\alpha$
    else
        $u = \alpha$
        $p_u = p_\alpha$
Until ($p_l = p_u$)

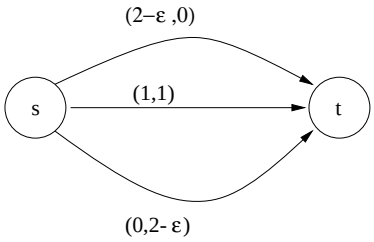

Figure 5: A graph for which the error is $\mathsf{error}(SP(G,\alpha)) = 1 - \epsilon$ for all $0 \leq \alpha \leq 1$

Theorem 6.1 *The above algorithm terminates in polynomial number of steps, and the α^* computed by the algorithm satisfies $M(SP(G,\alpha^*)) \leq M(SP(G,\alpha))$ for $\alpha \in [0,1]$.*

Although the above algorithm computes the α^* which minimizes $M(SP(G,\alpha))$, figure shows an example where the error $\mathsf{error}(SP(G,\alpha)) = 1 - \epsilon$ for all $0 \leq \alpha \leq 1$.

Example 6.1 *Consider the example in Figure 5 where the cost constraint is $C = 1$ and the delay constraint is $D = 1$. It is easy to check that for any $0 \leq \alpha \leq 1$, $\mathsf{error}(SP(G,\alpha)) = 1 - \epsilon$.*

7 A Linear Programming solution to a relaxed problem

In this section, we relax the requirements of our problem. Rather than asking for a single path that satisfies the cost and delay requirements, we allow for the data to be routed over multiple paths. But we require the average delay and average cost to satisfy the constraints.

Let us define f_e to be fraction of the data from the source to the destination that flows over the edge e. We then have the following constraints ($O(v)$ are the outgoing edges and $I(v)$ are the incoming edges from a vertex v):

$$\text{For each} \quad e \in E, f_e \geq 0$$

$$\sum_{e \in O(s)} f_e = 1 \tag{3}$$

$$\sum_{e \in I(t)} f_e = 1 \tag{4}$$

$$\text{For} \quad v \neq s \quad \text{and} \quad v \neq t, \sum_{e \in I(v)} f_e = \sum_{e \in O(v)} f_e \tag{5}$$

$$\sum_{e \in E} f_e \; Cost(e) \leq C \tag{6}$$

$$\sum_{e \in E} f_e \; Delay(e) \leq D \tag{7}$$

Equation 3- 5 are the balance equations for the nodes. Equation 6 states that the average cost must be less than the cost constraint C, and equation 7 states that the average delay must be less than the delay constraint D.

A feasible solution of the above linear program tells us how the data should be routed from the source to the destination so that average cost and delay constraints are satisfied.

Example 7.1 *Consider again Example 2.1 and Figure 1 with cost constraint 4 and delay constraint 3. If we formulate the above set of linear constraints for this problem, we note that* $f_e = \frac{1}{2}$ *for each edge* e *is a solution. This means that half of the data from the source is routed to node 2, and the other half to node 3. The average delay corresponding to this solution is* $\frac{1}{2}(2+2+1+1) = 3$*, and the average cost is* $\frac{1}{2}(3+2+1+1) = 3.5$*. Notice that even though the average cost and delay satisfy the constraints, individual paths may not (i.e., path* $(1,3)(3,4)$ *doesn't satisfy the delay constraint).*

If we restrict ourselves to integer solutions of the above linear program (i.e, a integer programming problem), each solution then represents a single path that satisfies the delay and cost constraints. Ofcourse, checking feasibility of integer linear programs is NP-Complete.

8 Other Extensions

In this section we discuss the extension of the problem to the case with more than two constraints. We also discuss a somewhat different, but sometimes more useful problem in practice, where we minimize the cost subject to a delay constraint.

8.1 More than two constraints

In a problem with k constraints, each edge is labelled with a k-tuple $(d_1, d_2, \ldots, d_k)$. We are required to find a path such that the sum of the ith constraint along the path is less than a bound C_i. The basic idea of the extension is that we store a set of k-tuples $(a_1, \ldots, a_k)$ at a node S when it is possible to get from S to the destination vertex along a path in which the sum of the ith constraint is a_i.

By a straight forward extension of the algorithm in Section 4 and Section 5, it is easy to show that we can get an exact algorithm with complexity $O(|V||E|\prod_{i=1}^{k} C_i)$, and an approximative algorithm with complexity $O(|V|^k|E|(1+\frac{1}{\epsilon})^k)$ where ϵ is the maximum error allowed.

It is also possible to extend some parts of the algorithm in Section 6. It is possible to obtain a path with error $\epsilon = k - 1$ for a problem with k constraints by solving the shortest path algorithm. But it is not clear how to extend the algorithm which iterates over shortest path problems to the case with more than two constraints.

8.2 An alternative formulation

An alternative and sometimes more useful formulation is when a bound is given on the delay, and subject to this, we are required to minimize the cost. The algorithm of Section 4 and Section 5 stores a set $\{(c_i, d_i)\}$ where it is possible to reach the destination vertex with cost c_i and delay d_i. From this set it is possible to find the pair (c_i, d_i) which has minimum cost and meets the delay constraint.

To be able to solve this alternate formulation, we also augmented the algorithm of Section 6. To find a path which meets the delay constraint D and has minimum cost C, we solve a problem with delay constraint D and cost constraint C where C is initially chosen to be large. We then find the minimum cost by performing a binary search on C.

9 Experimental results

We have implemented the bounded-error approximative algorithm (section 5) and the shortest-path based algorithm (section 6). The algorithms, implemented in C, are available to the public. The algorithms can be used

- either as stand-alone programs, taking as input a file containing the description of the graph;
- or as C functions called from a program. In this case the graph is stored in a specific data structure and passed as argument to the functions implementing the algorithms.

The usage of these tools is explained in more detail in the appendix.

In this section, we report experimental results obtained by applying the algorithms, we compare the performance of the algorithms and make a number of observations.

The algorithms were run on a set of multi-weight graphs obtained by elevation maps of physical landscapes. Each map was a 2-dimensional array M of integers, the i,j-th element giving the altitude of the longitude-latitude point corresponding to coordinates i,j (from now on, we use i and j to denote the longitude and latitude of the corresponding point). On top of M, we randomly generated two more arrays D_1, D_2, with same dimension as M, according to a Gaussian distribution, with a given number of "peaks". The value $D_k(i,j)$ of the i,j-th element of array $D_k, k = 1,2$, was supposed to represent the degree of "danger" of point (i,j), for two different types of danger. Said differently, $D_k(i,j)$ represents the probability of a vehicle "failing" for reason k, at point (i,j), so that the sum of $D_k(i,j)$ for all i,j is equal to 1. The probabilities of failing in different points are assumed to be independent, so that the probability of failing along a path is the sum of probabilities of failing on either of the traversed points.

For each map (along with the random arrays D_k) we generated two multi-weight graphs, a 2-weight and a 3-weight graph, according to the following procedure:

- The nodes of the graph were the set of points i,j, $\texttt{lat}_{min} \leq i \leq \texttt{lat}_{max}$, $\texttt{lon}_{min} \leq j \leq \texttt{lon}_{max}$, where $\texttt{lat}_{min}, \texttt{lat}_{max}, \texttt{lon}_{min}, \texttt{lon}_{max}$ are minimum and maximum latitudes and longitudes of the map.
- There was an edge from node (i,j) to each of the nodes $(i-1,j)$, $(i,j-1)$, $(i+1,j)$ and $(i,j+1)$, except when i or j are border coordinates ($i = \texttt{lat}_{min}$ or $i = \texttt{lat}_{max}$, or $j = \texttt{lon}_{min}$ or $j = \texttt{lon}_{max}$), in which case (i,j) has only 3 or 2 edges. So, a central node has 4 neighbors (up, down, left, right), and there are no "diagonal" edges.

n	Number of nodes
m	Number of edges
c	"Cost" of path
d_1	"Delay 1" of path
d_2	"Delay 2" of path
δ_ϵ	Step-error for bounded-error approximate algorithm

Table 1: Notation for tables 2 and 3.

- The first weight ("cost") of an edge $(i,j) \rightarrow (i',j')$ was defined to be $\alpha \cdot |i'-i| + \beta \cdot |j'-j| + \gamma \cdot |e'-e|$, where e, e' were the elevations of points (i,j), (i',j'), and α, β were parameters. This weight was supposed to model the cost, say, in fuel or in time delay, that it takes for a vehicle to move from one point to another. This cost could be defined by any function $f(i,i',j,j',e,e')$. So, the cost of a path can be defined as the sum of costs of each of the edges.

- The second weight ("delay 1") of the 2-weight or the 3-weight graph of an edge $(i,j) \rightarrow (i',j')$ was defined to be $D_1(i,j)$.

- The third weight ("delay 2") of the 3-weight graph of an edge $(i,j) \rightarrow (i',j')$ was defined to be $D_2(i,j)$.

Then, the multi-constrained shortest-path problem was defined for each of the resulting graphs as follows. Given a source node S, a destination node D, and bounds B_1 and B_2, find the minimum-cost path from S to D such that delay i of the path is at most B_i. That is, we were trying to minimize the fuel consumption, or time delay, subject to some upper bounds on the probability of failing. For all the performance results reported in this section, $B_1 = B_2 = 0.1$.

We have generated various 2-weight and 3-weight graphs, of different sizes. The bounded-error algorithm was run on both the 2-weight and 3-weight graphs (according to the extensions described in section 8), where-as the shortest-path algorithm was run only on the 2-weight graphs. The algorithms were run in "batch" mode, reading their input from a file. The notation used below is explained in table 1. We measured the time consumed by the algorithms to actually compute the path (i.e., the time taken for parsing the input was ignored – the portion of this time was not significant in the bounded-error algorithm whereas it was indeed in the shortest-path algorithm). The times for each experiment (measured in seconds) are shown in the first, third

and fifth rows of the tables.

Tables 2 and 3 show the results of applying the bounded-error approximative algorithm and the shortest-path algorithm, respectively, on three different 2-weight graphs. The three graphs were obtained from a 91×51 map, a 183×199 map and a 455×496 map, respectively. For each graph, we run the two algorithms six times, varying the source/destination pairs and the step-error on the second weight. The objective was to see how sensitive the algorithms were to different source/destination pairs, as well as different step-errors.

The first source/destination pair was chosen to be the north-west and south-east points, i.e., $(0,0) \rightarrow (90,50)$, $(0,0) \rightarrow (182,198)$ and $(0,0) \rightarrow (454,495)$. For the second source/destination pair, we chose points closer to the center of the map: $(20,20) \rightarrow (50,50)$, $(50,50) \rightarrow (100,100)$ and $(60,60) \rightarrow (300,250)$. The step-error was chosen to be 10^{-4}, 10^{-5} or 10^{-6}.

From table 2, we observe the following:

- Independently of the step-error δ_ϵ, the algorithm always produced a path satisfying the B_1 constraint $\leq 10\%$.
- Reducing the step-error δ_ϵ by one or two orders of magnitude resulted in dramatic increases in running time, whereas the paths found were only slightly less "dangerous" than before, or the same as before.
- The algorithm consumes more time on the NW/SE source/destination pair. This is reasonable, since these two points are further away from each other than the points of the other source/destination pair. This implies a higher number of iterations until the costs stabilize in the first case than in the second.

From table 3, we observe the following:

- The shortest-path algorithm is two or more orders of magnitude faster than the bounded-error approximative algorithm.
- Although in principle the shortest-path algorithm can yield paths as much as twice as dangerous as a feasible path, in these experiments, the shortest-path algorithm actually gave less or same dangerous paths than the bounded-error approximative algorithm.

	Source/dest.: NW/SE points			Other source/dest.		
	$\delta_\epsilon = 10^{-4}$	$\delta_\epsilon = 10^{-5}$	$\delta_\epsilon = 10^{-6}$	$\delta_\epsilon = 10^{-4}$	$\delta_\epsilon = 10^{-5}$	$\delta_\epsilon = 10^{-6}$
graph 1	70 s.	413 s.	1274 s.	13 s.	51 s.	118 s.
$n = 4641$	$c = 1532$			$c = 722$		
$m \approx 4 \cdot n$	$d_1 = 2.15\%$			$d_1 = 0.2\%$	$d_1 = 0.14\%$	
graph 2	56 s.	1278 s.	16740 s.	153 s.	4668 s.	32477 s.
$n = 36417$	$c = 4152$			$c = 1172$		
$m \approx 4 \cdot n$	$d_1 = 0.2\%$	$d_1 = 0.15\%$		$d_1 = 0.3\%$		
graph 3	725 s.	3503 s.	9971 s.	193 s.	419 s.	1387 s.
$n = 225680$	$c = 9409$			$c = 4298$		
$m \approx 4 \cdot n$	$d_1 = 0.01\%$	$d_1 = 0\%$		$d_1 = 0.02\%$	$d_1 = 0\%$	

Table 2: Results of the bounded-error approximative algorithm on 3-weight graphs.

- In terms of cost, the shortest-path algorithm yields a little more costly paths than the bounded-error approximative algorithm.

From these observations, it seems that although the "best" paths (paths which are feasible in terms of danger and have minimum cost) are indeed obtained using the bounded-error approximative algorithm, the shortest-path algorithm yields comparably good paths in incomparably less time.

Figure 6 illustrates a particular execution of the bounded-error approximative algorithm.

Table 4 shows the results of applying the bounded-error approximative algorithm on three different 3-weight graphs. The three graphs were obtained from a 37×21 map, a 38×31 map and a 42×41 map, respectively. Our objective was to see how adding one more weight/constraint to the problem affects the performance of the algorithm. Our experiments showed a dramatic increase in execution time with respect to 2-weight graphs, and this is why we could only treat 3-weight graphs of relatively small sizes (up to 1700 nodes).

For each graph, we run the algorithm three times, changing the step-error in delay 2 (the step error in delay 1 was kept fixed at 10^{-6}). It is worth noting that the execution time improves a lot as the step-error increases, while at the same time the paths we obtain are exactly the same.

	Source/dest.: NW/SE points	Other source/dest.
graph 1	0.94 s.	0.48 s.
$n = 4641$	$c = 1564$	$c = 723$
$m \approx 4 \cdot n$	$d_1 = 1.27\%$	$d_1 = 0.14\%$
graph 2	7.16 s.	3.48 s.
$n = 36417$	$c = 4220$	$c = 1184$
$m \approx 4 \cdot n$	$d_1 = 0.04\%$	$d_1 = 0\%$
graph 3	47.96 s.	31.56 s.
$n = 225680$	$c = 9411$	$c = 4487$
$m \approx 4 \cdot n$	$d_1 = 0\%$	$d_1 = 0\%$

Table 3: Results of the shortest-path algorithm on 2-weight graphs.

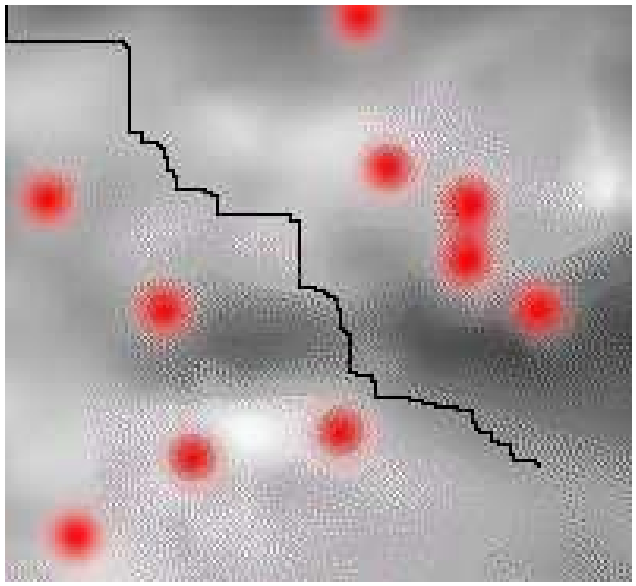

Figure 6: An output of the bounded-error approximative algorithm: the solid black line depicts the path; red dots are "danger" zones; the grey scale background represents the elevation variations of the terrain (white: high, black: low).

	$\delta_\epsilon = 10^{-4}$	$\delta_\epsilon = 10^{-5}$	$\delta_\epsilon = 10^{-6}$
graph 4	1.49 s.	1.80 s.	1.85 s.
$n = 777$	$c = 720$		
$m \approx 4 \cdot n$	$d_1 = 5.6\%$		
	$d_2 = 6.43\%$		
graph 5	4.25 s.	153.97 s.	6095.42 s.
$n = 1178$	$c = 994$		
$m \approx 4 \cdot n$	$d_1 = 1.79\%$		
	$d_2 = 5.39\%$		
graph 6	1352.54 s.	17631.51 s.	29274.12 s.
$n = 1722$	$c = 1024$		
$m \approx 4 \cdot n$	$d_1 = 3.68\%$		
	$d_2 = 4.66\%$		

Table 4: Results of the bounded-error algorithm on 3-weight graphs.

10 Conclusion

In this paper, we have presented several different algorithms for solving the routing problem with multiple constraints. These algorithms vary in their complexity and the accuracy of their solutions. We have implemented these algorithms and compared their performance on different graphs.

References

[1] M.R. Garey and D.S. Johnson, *Computers and Intractability — A Guide to the theory of NP-Completeness*, Freeman, San Francisco, 1979.

[2] R. Hassin, Approximation Schemes for the Restricted Shortest Path Problem, *Mathematics of Operations Research*, Vol. 17, No. 1, Feb. 1992.

[3] J. M. Jaffe, Algorithms for Finding Paths with Multiple Constraints, *Networks*, Vol. 14, 1984, pp. 95-116.

[4] Z. Wang and J. Crowcroft, Quality-of-Service Routing for Supporting Multimedia Applications, *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 7, Sept. 1996.

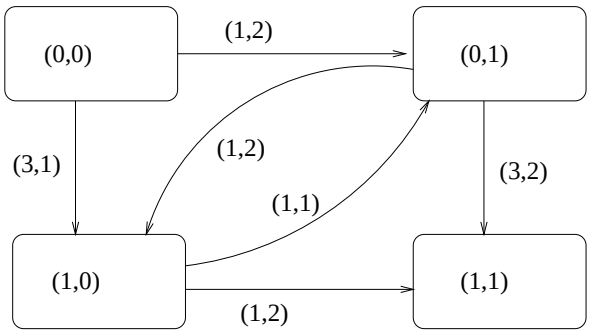

Figure 7: An Example Graph

A Implementation and Input Format

The system consists of a parser, a module implementing the bounded error algorithm and a module implementing the shortest path algorithm. The complete system consists of about 2000 lines of C code. The input to the system is a graph along with a specification of bounds on the cost of the paths. The input to the system can either be provided as a text file, or provided to the system through a function call. Section A.1 describes the format of the text file that describes the graph, and Section A.2 describes the format of the function call and the associated data structures.

A.1 File Format for Graphs

Figure 8 shows the input file for the graph of Figure 7. The field "Num Vertex" specifies the number of vertices in the graph. The field "Num Constraint" specifies the number of constraints (for the shortest path algorithm, this must be set to either 1 or 2). The field "Source" specifies the start vertex, and the field "Destination" specifies the destination vertex. The fields "Constraint 0" and "Constraint 1" specifies the bound on the total cost of constraint 0 and 1 on the path found. The fields "Error Constraint 0" and "Error Constraint 1" specify the errors that the bounded error algorithm can make at each iteration. The "Vertex" field is followed by the vertex name. The field "Num Edges" specifies the number of edges a vertex has. Each edge is then specified by the field "to" to which the edge is going, followed by the cost of constraint i for each i.

The grammar for the input graph is given in Figure 9. All words in boldface are

```
Num Vertex: 4
Num Edges: 0
Num Constraint: 2

Source: (0,0)
Destination: (1,1)

Minimize: 0

Constraint 0: 3.5
Constraint 1: 6.0

Error Constraint 0: 0.001
Error Constraint 1: 0.001

Vertex: (0,0)
Num Edges: 2
   to: (0,1) 0: 1     1: 2
   to: (1,0) 0: 3     1: 1

Vertex: (0,1)
Num Edges: 2
   to: (1,0) 0: 1     1: 2
   to: (1,1) 0: 3     1: 2

Vertex: (1,0)
Num Edges: 2
   to: (1,1) 0: 1     1: 2
   to: (0,1) 0: 1     1: 1

Vertex: (1,1)
Num Edges: 0
```

Figure 8: Input file for a graph

keywords that can be typed either in uppercase or lowercase. A Name is any string of characters beginning with a letter, a "(", ")", "_", or ",", followed by any of the same characters or digits. An integer is a sequence of digits. A float is a floating point number expressed as a seqence of digits followed by a decimal point, followed by more digits.

A.2 Interface through Function Calls

An alternative interface to the program is provided through two function calls. The function call **MakeRoute()** takes as input a graph specification and computes a path using the bounded error algorithm. The function call **MakeRouteWithShortestPath()** also takes as input a graph and uses the shortest path algorithm to find a path. Both algorithms return the path found. The algorithms require that the caller specify the graph in the internal data structures used by our algorithm.

Figure 10 shows the interface to the **MakeRoute()** call and Figure 11 shows the interface for the **MakeRouteWithShortestPath()** call. The parameter **VertexNo** is the number of vertices in the graph, **ConstraintNo** is the number of constraints, **ConstraintToMinimize** is the constraint to be minimized, **StartVertex** is the starting vertex and **EndVertex** is the destination vertex. The parameter **ConstraintBounds** is an array of the bounds on constraints, and **ConstraintErrors** is an array of error allowed for each constraint for each iteration. The parameter **vertices** specifies the graph as an array of vertices.

We next describe how the graph is specified as an array of vertices and the path which is returned by the function calls.

Data Structures for the Graph

A graph is specified as an array of vertices of length **VertexNo** where each vertex is of type **Vertex** shown in Figure 12. Each vertex has a field **sym** which should be set to **NULL** since it is only used when the input is given as a file. The field **NumEdges** is the number of out going edges from the vertex. Each vertex also points to a list of edges with the field **e**.

Each edge is of type **Edge** shown in Figure 13. The field **to** should be set to **NULL** because it is only used when the input is a file. The field **ToVertex** is the vertex to which the edge points. The field **constr** is an array of length **ConstraintNo**

Start: NumVertex NumConstraints SourceVertex DestinationVertex Minimize AllConstraints AllErrorConstraints Vertices

NumVertex: **num vertex :** Integer

NumConstraints: **num constraint :** Integer

SourceVertex: **source :** Name

DestinationVertex: **destination :** Name

Minimize: **minimize :** Integer

AllConstraints: ϵ | AllConstraints Constraint

Constraint: **constraint** Integer **:** Number

AllErrorConstraints: ϵ | AllErrorConstraints ErrorConstraint

ErrorConstraint: **error constraint** Integer **:** Number

Vertices: ϵ | Vertices OneVertex

OneVertex: **vertex :** Name **num edges :** Integer VertexEdges

VertexEdges: ϵ | VertexEdges AnEdge

AnEdge: **to :** Name EdgeConstraints

EdgeConstraints: ϵ | EdgeConstraints SingleEdgeConstraint

SingleEdgeConstraint: Integer **:** Number

Number: Integer | Float

Figure 9: Grammar for the Input Graph

```
ROUTE MakeRoute(
                int VertexNo,   /* Number of vertices */
                int ConstraintNo,   /* Number of constraints */
                int ConstraintToMinimize,   /* Constraint to be minimized */
                double *ConstraintBounds,   /* Array of Constraint bounds */
                double *ConstraintErrors,   /* Errors allowed on Constraints */
                Vertex *vertices,   /* Array of vertices */
                int StartVertex,   /* Starting Vertex */
                int EndVertex   /* Ending Vertex */
        )
```

Figure 10: Function Call for the Bounded Error Algorithm

```
ROUTE MakeRouteWithShortestPath(
                int VertexNo,   /* Number of vertices */
                int ConstraintNo,   /* Number of constraints */
                int ConstraintToMinimize,   /* Constraint to be minimized */
                double *ConstraintBounds,   /* Array of Constraint bounds */
                double *ConstraintErrors,   /* Errors allowed on Constraints */
                Vertex *vertices,   /* Array of vertices */
                int StartVertex,   /* Starting Vertex */
                int EndVertex   /* Ending Vertex */
        )
```

Figure 11: Function Call for the Shortest Path Algorithm

containing the value of the costs labeling the edge. The field **next** points to the next edge from this vertex.

```
/* Data structure for storing a vertex */
typedef struct Vertex {
        struct Symbol *sym;   /* Name of the vertex */
        int NumEdges;   /* number of edges for the vertex */
        struct Edge *e;   /* link to edges */
} Vertex;
```

Figure 12: Type definition for **Vertex**

```
/* Data structure for storing an edge */
typedef struct Edge {
        Symbol *to;   /* where is the edge going to */
        int ToVertex;   /* vertex to which the edge is going */
        Constraint *constr;   /* the value of the costs on the edges */
        struct Edge *next;   /* next edge */
} Edge;
```

Figure 13: Type definition for **Edge**

Data Structure for Returned Path

A call to functions **MakeRoute()** and **MakeRouteWithShortestPath()** returns a variable of type **ROUTE** which is defined in Figure 14 and Figure 15. The field **status** indicates the status of the function call and can take the values **FOUND_ROUTE, MORE_THAN_2_CONSTRAINTS, DESTINATION_NOT_REACHABLE, or PATH_LENGTH_EXCEEDED**. Only when the status equals **FOUND_ROUTE** is a valid path been found. The status equals **MORE_THAN_2_CONSTRAINTS** if the function **MakeRouteWithShortestPath()** is called with more than two constraints. The status equals **DESTINATION_NOT_REACHABLE** if the destination is not reachable in the graph. And the status equals **PATH_LENGTH_EXCEEDED**

if the path was longer than **VertexNo**. This could occur, for example, if some of the edge weights were negative.

When the status equals **FOUND_ROUTE**, a valid path has been found. In that case the **PathLength** field containts the length of the Path found, the field **Path[i]** is the ith vertex on the path and the field **cost[i]** is the total cost of the ith constraint for the path. When the caller is finished using the path, it should deallocate the memory used by the path by calling **freeROUTEData(r)** where **r** was the variable of type **ROUTE** originally obtained.

```
/* pointer to data structure containing the path found */
typedef ROUTEInfo *ROUTE;
```

Figure 14: Type definition for **ROUTE**

```
/* Data structure containing the path found */
typedef struct {
int status;
            double *cost;
            int PathLength;
            int *Path;
} ROUTEInfo;
```

Figure 15: Type definition for **ROUTEInfo**