An Efficient Algorithm for Finding a Path Subject to Two Additive Constraints*

Turgay Korkmaz   Marwan Krunz   Spyros Tragoudas
Department of Electrical & Computer Engineering
University of Arizona
Tucson, AZ 85721
{turgay,krunz,spyros}@ece.arizona.edu

June 14, 1999

Abstract

One of the key issues in providing end-to-end quality-of-service (QoS) guarantees in packet networks is how to determine a *feasible* route that satisfies a set of constraints while simultaneously maintaining high utilization of network resources. In general, finding a path subject to multiple *additive* constraints (e.g., delay, delay-jitter) is an NP-complete problem. In this paper, we provide a fast heuristic algorithm for finding a path subject to two constraints. The worst-case computational complexity of our algorithm is $\log W$ times the complexity of Dijkstra's shortest path algorithm, where W is an upper bound on the cost of the longest path with respect to one of the two constraints. In practice, the average complexity of the algorithm is even much lower, since the worst-case complexity is rarely needed. The performance of the proposed algorithm is justified via theoretical bounds that are provided for the optimal version of the path selection problem. To achieve further performance improvement, several extensions to the basic algorithm are also provided at very low computational cost. Extensive simulations are used to demonstrate the high performance of the proposed algorithm and to contrast it with other path selection algorithms.

keywords: Multiple constrained path selection, QoS-based routing, scalable routing.

1 Introduction

Integrated network services (e.g., ATM, Intserv) are being designed to provide quality-of-service (QoS) guarantees to various applications such as audio, video, and data. Many of these applications have multiple QoS requirements in terms of bandwidth, delay, delay-jitter, loss, etc. One of the important problems in QoS-based service offerings is how to determine a route that satisfies multiple constraints (or QoS requirements) while simultaneously achieving efficient utilization of network

*This work was supported by the National Science Foundation under Grant ANI 9733143 and Grant CCR 9815229. Authors names are listed in alphabetical order.

resources. This problem is known as QoS-based (or constrained-based) routing, and is being extensively investigated in the research community [20, 15, 16, 13, 3, 9, 17, 10, 12]. In [5] the authors survey various aspects of QoS-based routing, including the maintenance of state information, routing strategies, and path selection algorithms. In general, routing consists of two basic tasks [5]: distributing the state information of the network and searching this information for a feasible path with respect to given constraints. In this paper, we focus on the second task, and assume that the true state of the network is available to every node (e.g., via link-state routing) and that nodes use this state information to determine an end-to-end feasible path. Each link in the network is associated with multiple QoS parameters. These parameters can be classified into additive and non-additive constraints [2, 18]. For additive constraints (e.g., delay), the cost of an end-to-end path is given, exactly or approximately, by the *sum* of the individual link values of the constraint along that path. In contrast, the cost of a path with respect to a non-additive constraint (such as bandwidth) is determined by the value of that constraint at the bottleneck link. Non-additive constraints can be easily dealt with as a preprocessing step by pruning all links that do not satisfy the requested non-additive QoS values [19]. Hence, in this paper we will mainly focus on additive constraints. The underlying problem of path selection subject to two additive constraints can be stated as follows.

Definition 1 *Multiple Constrained Path Selection (MCP)*: Consider a network that is represented by a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of links. Each link $(i, j) \in E$ is associated with two non-negative additive QoS values: $w_1(i, j)$ and $w_2(i, j)$. Given two constraints c_1 and c_2, the problem is to find a path p from a source node s to a destination node t such that $w_1(p) \leq c_1$ and $w_2(p) \leq c_2$, where $w_1(p) \stackrel{\text{def}}{=} \sum_{(i,j)\in p} w_1(i, j)$ and $w_2(p) \stackrel{\text{def}}{=} \sum_{(i,j)\in p} w_2(i, j)$. Assume that $c_1 \geq c_2$; otherwise, we can exchange the order of the two link weights.

The MCP decision problem is known to be NP-complete [8, 11]. In other words, there is no efficient (polynomial-time) algorithm that can *surely* find a feasible path that simultaneously satisfies both constraints. To cope with the NP-completeness of the problem, researchers have resorted to two approaches: pseudo-polynomial-time algorithms in which the complexity depends on the values of the link weights in addition to the number of nodes (or links), and polynomial-time heuristics that search fast for a feasible path but cannot guarantee finding every feasible path in the network. In [11] the author presents a pseudo-polynomial-time algorithm for MCP. The computational complexity of this algorithm is $\mathcal{O}(n^5 b \log nb)$, where n is the number of nodes and b is the maximum weight of a link. Its space complexity is $\mathcal{O}(n^3 b \log nb)$. If b is large, this algorithm is very expensive in terms of computational time and memory requirement. To search fast for a feasible path, the author in [11] proposes a heuristic approximation algorithm. This approximation approach is related to our work, and it will be discussed later. Another related work on MCP is presented in [4], where the authors propose a heuristic algorithm that scales down the values of one link weight to bounded integers while maintaining the values of the other link weight. They show that the modified problem can be solved by using Dijkstra's (or Bellman-Ford) shortest path algorithm and that the solution for the modified problem is also a solution for the original one. When Dijkstra's algorithm is used, the computational complexity of the proposed algorithm in [4] is $\mathcal{O}(x^2n^2)$;[1] when Bellman-Ford algorithm is used, the complexity is $\mathcal{O}(xnm)$, where x is an adjustable positive integer, n is the number of nodes, and m is the number of links. The space complexity is $\mathcal{O}(xn)$. The value of x determines the performance and overhead of the algorithm. If x is small, the algorithm is fast but it does not have good performance. If x is large, the algorithm has good performance but it is too slow. To increase the probability of finding a feasible path, the value of x can be as large as $10n$, resulting in $\mathcal{O}(n^4)$ computational complexity with Dijkstra's algorithm or $\mathcal{O}(n^2m)$ with Bellman-Ford algorithm. The modified problem in [4] is very similar to the well known restricted shortest path (RSP) problem, which can be solved via a dynamic-programming algorithm [8, 14]. In fact, the dynamic-programming algorithm in [14] can be used to solve the modified problem in [4] with computational complexity of $\mathcal{O}(xm)$ and space complexity of $\mathcal{O}(xn)$.

Due to their computational and space complexities, the above approaches are not practical for large networks. An algorithm that approximates the constraints but runs sufficiently fast with some performance guarantee would be more appropriate for such networks.

In this paper, we provide an efficient heuristic algorithm with a good approximation bound that yields high performance in finding feasible paths. To search fast for a feasible path under two constraints, a natural approach is to approximate the constraints by minimizing $\alpha w_1(p) + \beta w_2(p)$, where $\alpha, \beta \in Z^+$ [11]. For given α and β, one can define a combined weight $l(e) = \alpha w_1(e) + \beta w_2(e)$ for every link e and solve the minimization problem using Dijkstra's shortest path algorithm. However, an optimal path with respect to (w.r.t.) $l(e)$ may not satisfy the individual constraints. The key issue here is to determine the appropriate α and β such that an optimal path w.r.t. $l(e)$ is likely to satisfy the individual constraints from theoretical and practical perspectives. In [11] the author uses an objective function of the form $f(p) = \max\{w_1(p), c_1\} + \max\{w_2(p), c_2\}$ as a basis for determining the appropriate α and β. In some sense, this function measures the amount by which a path p violates the constraints c_1 and c_2. By minimizing f for $\alpha = \beta = 1$ and for $\alpha = 1$ and $\beta = \sqrt{c_1/c_2}$, theoretical bounds can be provided on the performance of the path selection algorithm [11]. Note that these bounds indicate how far a path p is from the optimal one *according to the objective function*. However, no performance guarantee for path p is given w.r.t. individual constraints. Correct interpretation of these bounds is crucial to understanding the novelty of the bounds presented in this paper.

In contrast to the algorithm in [11], we provide (in Section 2) an approximation algorithm that efficiently searches for appropriate values for α and β, and returns a path p with the following properties: (a) $w_2(p) \leq c_2$, and (b) $w_1(p)$ is within a given factor from a feasible path f for which $w_1(f)$ is minimum. The approximation bound in (b) indicates that the returned path p is very likely to satisfy the constraint c_1. The worst-case complexity of our basic algorithm is $\log W$ times the complexity of Dijkstra's algorithm, where W is an upper bound on the longest path with respect

[1] Computational complexity of Dijkstra's algorithm can be improved using appropriate data structures [1].

to one of the link weights. Its average complexity is observed to be much less than that. The space complexity is $\mathcal{O}(n)$, where n is the number of nodes. By proper interpretation of the bounds in (a) and (b), we present in Section 3 two extensions to our basic algorithm that achieve further improvement in the routing performance at small extra computational cost. Simulation results, which are provided in Section 4, demonstrate the high performance of our algorithm and contrast it with other path selection algorithms.

2 Basic Approximation Algorithm For MCP

Our algorithm consists of two phases (see Figure 1). Phase 2 is executed only if Phase 1 fails. In Phase 1, the algorithm assigns each edge $e \in E$ a weight $l_1(e) \stackrel{\text{def}}{=} kw_1(e) + w_2(e)$ ($\alpha = k$ and $\beta = 1$). In Phase 2, the algorithm assigns each edge $e \in E$ a weight $l_2(e) \stackrel{\text{def}}{=} w_1(e)+kw_2(e)$ ($\alpha = 1$ and $\beta = k$). In each phase, the algorithm runs a hierarchical version of Dijkstra's shortest path algorithm which returns one of the shortest paths w.r.t. $l_1(e)$ or $l_2(e)$ at a given k and finds the minimum $w_2()$ among all these shortest paths.

The standard Dijkstra's algorithm maintains two labels for each node [7]: $d[u]$ to represent the estimated total cost of the shortest path from the source node s to node u w.r.t. the combined weight, and $\pi[u]$ to represent the predecessor of node u along the shortest path. The hierarchical version of Dijkstra's algorithm maintains additional labels: $w_1[u]$ and $w_2[u]$ to represent the cost of the shortest path from s to u w.r.t. the individual weights, and label $min\_w_2[u]$ to represent the minimum w_2 weight among all shortest paths from s to u.[2] To carry out these tasks, some modifications are needed in the relaxation process of the Dijkstra's algorithm (lines 4–9 in Figure 2). The standard relaxation process (lines 1–3 in Figure 2) tests whether the shortest path found so far from source node s to node v can be improved by passing through node u. If so, $d[v]$ and $\pi[v]$ are updated [7].

If the returned shortest path w.r.t. $l_1(e)$ or $l_2(e)$ is not feasible, the algorithm repeats the hierarchical Dijkstra's algorithm using up to a logarithmic number of different values of k in the range $[1, n \cdot \max\{w_i(e) \mid e \in E\}]$, where $i = 2$ in Phase 1 and $i = 1$ in Phase 2. Note that $n \cdot \max\{w_1(e) \mid e \in E\}$ and $n \cdot \max\{w_2(e) \mid e \in E\}$ are upper bounds on the total cost of the longest path w.r.t. link weights w_1 and w_2, respectively. Two lemmas, one per phase, will show that a binary search argument in the above ranges can be used to determine the maximum value of k in Phase 1 and the minimum value of k in Phase 2. We will prove the lemma for Phase 2. Similar arguments can be used for Phase 1.

If the algorithm fails to return a feasible path in Phase 1 where k is maximized, it returns a path p such that $w_2(p) \le c_2$ and $w_1(p) \le w_1(f) + (w_2(f) - w_2(p))/k$, where f is some feasible path w.r.t. both constraints c_1 and c_2. If the algorithm fails in Phase 2 where k is minimized, it returns a path p such that $w_2(p) \le c_2$ and $w_1(p) \le w_1(f) + k(w_2(f) - w_2(p))$. This is a reasonable

[2]Notice that $w_i[.]$ indicates a node label, whereas $w_i(.)$ indicates the weight of a link or the cost of a path.

Find a path p from s to t in the network $G = (V, E)$
such that $w_1(p) = w_1[t] \leq c_1$ and $w_2(p) = w_2[t] \leq c_2$.

Phase 1

```
k_min := 1
k_max := n · max{w_2(e) | e ∈ E}
while( k_min ≠ k_max ) do
    k := ⌈(k_min + k_max)/2⌉
    Set l_1(e) := k w_1(e) + w_2(e) ∀e ∈ E
    Execute hierarchical Dijkstra's algorithm with link weights {l_1(e) : e ∈ E}
    if w_2[t] ≤ c_2 and w_1[t] ≤ c_1 then
        return SUCCESS
    end if
    if min_w_2[t] ≤ c_2 then
        k_min := k                /* increase k */
    else
        k_max := k                /* decrease k */
    end if
end while
return FAILURE
end Phase 1
```

Phase 2 /* execute only if **Phase 1** fails */

```
k_min := 1
k_max := n · max{w_1(e) | e ∈ E}
while( k_min ≠ k_max ) do
    k := ⌈(k_min + k_max)/2⌉
    Set l_2(e) := w_1(e) + k w_2(e) ∀e ∈ E
    Execute hierarchical Dijkstra's algorithm with link weights {l_2(e) : e ∈ E}
    if w_2[t] ≤ c_2 and w_1[t] ≤ c_1 then
        return SUCCESS
    end if
    if min_w_2[t] ≤ c_2 then
        k_max := k                /* decrease k */
    else
        k_min := k                /* increase k */
    end if
end while
return FAILURE
end Phase 2
```

Figure 1: Two main phases of the basic approximation algorithm for finding a feasible path subject to two additive constraints.

```
Relax(u,v)
/* i = 1 in Phase 1; i = 2 in Phase 2 */
if d[v] > d[u] + l_i(u, v) then
    d[v] := d[u] + l_i(u, v)
    π[v] := u
    w_1[v] := w_1[u] + w_1(u, v)
    w_2[v] := w_2[u] + w_2(u, v)
    min_w_2[v] := w_2[v]
    else if d[v] = d[u] + l_i(u, v) then
        if min_w_2[v] > min_w_2[u] + w_2(u, v) then
            min_w_2[v] := min_w_2[u] + w_2(u, v)
        end if
    end if
end if
```

Figure 2: New relaxation procedure for the hierarchical version of Dijkstra's algorithm.

scenario for searching fast for a feasible path that satisfies c_2 and that is very likely to satisfy c_1. One lemma per phase will show that the algorithm returns a path p which is either feasible or satisfies the above approximation bounds. Again we will prove the lemma for Phase 2, with an analogous argument applicable to Phase 1. In addition to maximizing or minimizing k, the algorithm may attempt to minimize the difference $(w_2(f) - w_2(p))$ to make the approximation bound tighter. This is an extension to the basic algorithm, which will be presented in Section 3.

In the rest of this section, we first illustrate how our algorithm works and contrast it with the one in [11]. Second, we present the binary search argument with the related lemmas and proofs. Finally, we prove the performance bounds that are associated with our basic algorithm.

2.1 How the Algorithm Works

Figure 3 describes how an approximation algorithm that minimizes $w_1(p) + kw_2(p)$ scans the search space for a feasible path at a given value for k. The shaded area indicates the feasibility region. Black dots represent the costs of different paths from source node s to destination node t. Each line in the figure shows the equivalence class of equal-cost paths w.r.t. the combined weight. The approximation algorithm determines a line for the given value of k, and then moves this line outward from the origin in the direction of the arrow. Whenever this line hits a path (i.e., black dot in the figure), the algorithm returns this path which is the shortest path w.r.t. the combined weight at the given k. The approximation algorithm in [11] makes a good guess for k (e.g., $k = 1$) and returns a path based on this k. However, if this path is infeasible the algorithm in [11] cannot proceed. As shown in Figure 3, the likelihood of finding a feasible path is much higher if one tries different

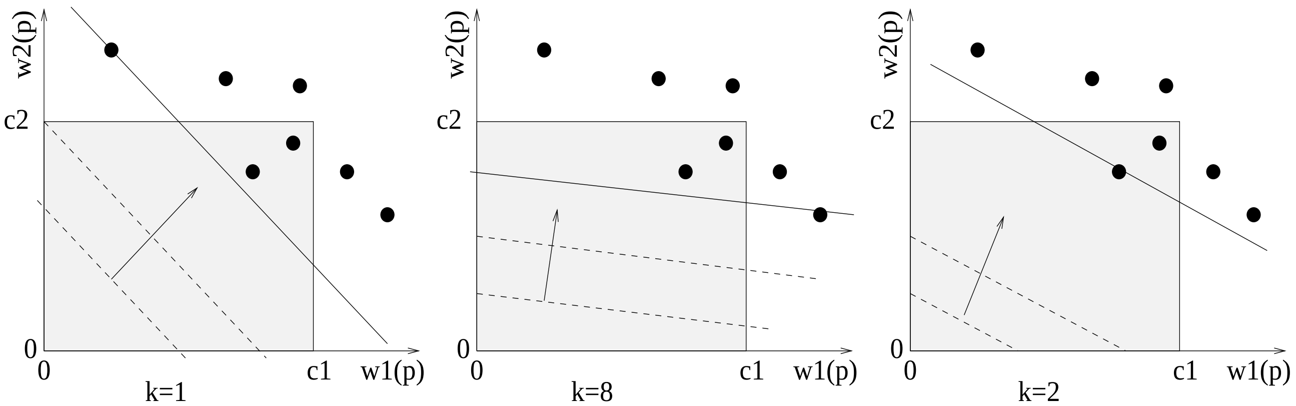

Figure 3: How the approximation algorithm searches the feasible region using different values for k.

values of k (e.g., $k = 2$ in this example results in a feasible path). The advantage of our algorithm over the one in [11] is that ours searches systematically for a good value for k, instead of fixing it in advance. Our algorithm scans the search space in two phases. In Phase 1, the algorithm minimizes $kw_1(p) + w_2(p)$ for a given k. If the returned path p is not feasible, then the algorithm decides to increase or decrease the value of k based on whether $w_2(p) \leq c_2$ or not. If $w_2(p) \leq c_2$, then the algorithm increases k; otherwise, the algorithm decreases k. The systematic adjustment of k during

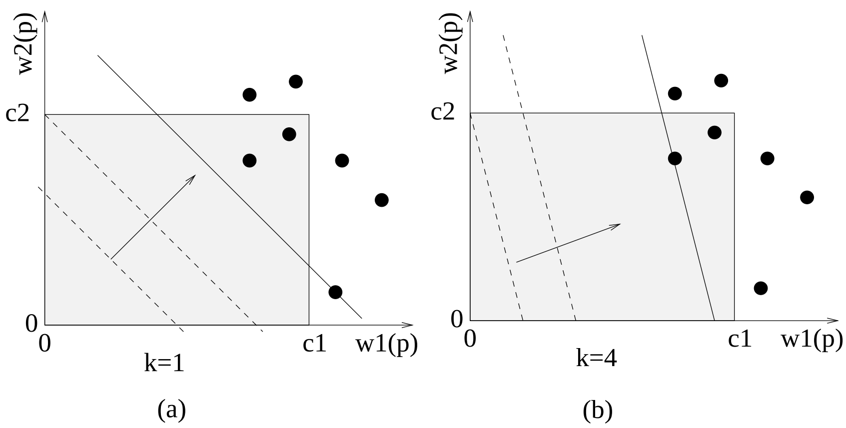

Figure 4: Searching for a feasible path in Phase 1.

Phase 1 is illustrated in the example in Figure 4. In part (a) of this figure, the algorithm starts with $k = 1$, but it returns an infeasible path (one that intersects the solid line). Since this path satisfies c_2 but not c_1, the algorithm increases k ($k = 4$) and returns a feasible path, as shown in Figure 4(b). Now suppose that Phase 1 cannot return a feasible path. Figure 5(a) illustrates one such scenario where the same infeasible path is being returned for all possible values of k, $k \geq 1$. Since Phase 1 fails in this case, the algorithm tries Phase 2, which minimizes $w_1(p) + kw_2(p)$ for a given k. If the returned path p is not feasible in Phase 2, then the algorithm again decides to increase or decrease the value of k based on whether $w_2(p) \leq c_2$ or not. If $w_2(p) \leq c_2$, the algorithm decreases k; otherwise, it increases k. Figure 5 illustrates how Phase 2 finds a feasible path that could not

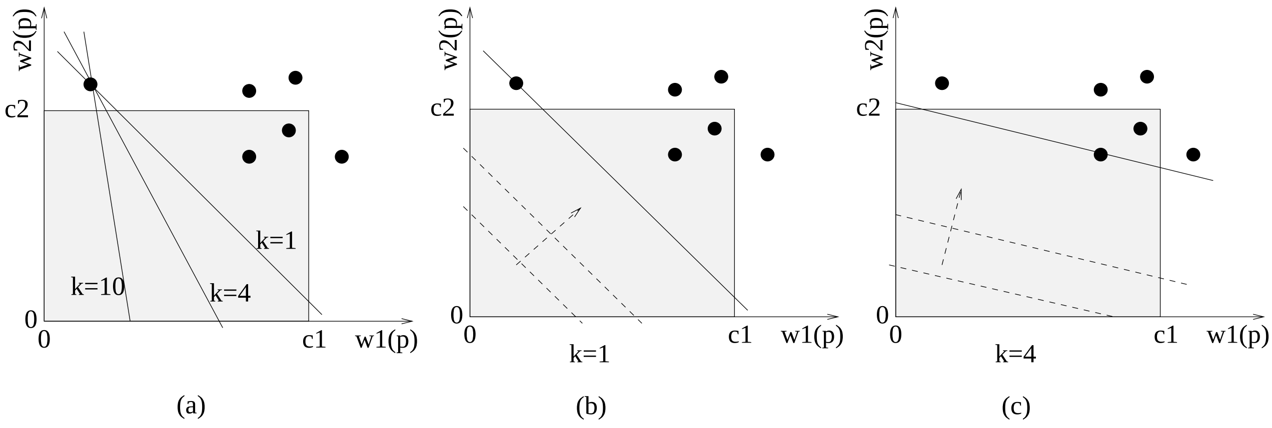

Figure 5: Searching for a feasible path in Phase 2. Part (a) shows a scenario in which Phase 1 fails to find a feasible path. Such a path is later found during the search process of Phase 2 (Parts (b) and (c)).

be found in Phase 1. In Figure 5(b), the algorithm executes Phase 2 using $k = 1$, but returns an infeasible. Since this path does not satisfy c_2, the algorithm increases k and returns a feasible path in Figure 5(c). If Phase 2 also fails, the basic algorithm stops even though there might be a feasible path in the network. In Section 3, we illustrate such a case and provide possible remedies to it based on a scaling extension to the basic algorithm.

2.2 Binary Search

Lemma 1 *(Maximum k from Phase 1) Suppose that each link $e \in E$ is assigned a weight $l_1(e) = kw_1(e) + w_2(e)$ in Phase 1, where k is an integer. After executing Phase 1, if the algorithm cannot find a path p for which $l_1(p)$ is minimum and $w_2(p) \leq c_2$, then such a path p cannot be found with larger values of k.*

Lemma 1 implies that using a binary search, the algorithm can determine an appropriate value for k. Although in the worst-case this search requires $\log(n \cdot \max\{w_2(e) \mid e \in E\})$ iterations of the hierarchical Dijkstra's algorithm, on average it requires significantly fewer iterations than that, since the worst-case complexity is rarely needed in practice. We will empirically show how many iterations of Dijkstra's algorithm is needed, on average.

Lemma 2 *(Minimum k from Phase 2) Suppose that each link $e \in E$ is assigned a weight $l_2(e) = w_1(e) + kw_2(e)$ in Phase 2, where k is an integer. After executing Phase 2, if the algorithm cannot find a path p for which $l_1(p)$ is minimum and $w_2(p) \leq c_2$, then such a path p cannot be found with smaller values of k.*

Lemma 2 also implies that the algorithm can determine an appropriate value for k using a binary search. In the worst-case, this search requires $\log(n \cdot \max\{w_1(e) \mid e \in E\})$ iterations of the hierarchical Dijkstra's algorithm. Again the average complexity is much lower than that.

Proof of Lemma 2: Phase 2 applies binary search to find the smallest k such that there exists a shortest path p w.r.t. $l_2(e) = w_1(e) + kw_2(e)$ with $w_2(p) \le c_2$. Assume that $k = 2j$ for some integer j. Let $\mathcal{P}$ be the set of all paths from source s to destination t w.r.t. $l_2()$ and let p be a path that the algorithm selects during a repetition of the binary search in Phase 2. In the repetition of the binary search at $k = 2j$, since all edges are assigned weights $l_2(e) = w_1(e) + 2jw_2(e)$, we have

$$l_2(p) = \min_{q \in \mathcal{P}} \{\sum_{e \in q} w_1(e) + 2jw_2(e)\}.$$

In order to prove the lemma, it suffices to show that if

$$\sum_{e \in p} w_2(e) > c_2$$

then the algorithm should never search for a path p' that satisfies the c_2 constraint by assigning weights $l_2(e) = w_1(e) + jw_2(e)$.

By explicitly checking $min\_w_2[t]$ in line 10 of Figure 1, the algorithm guarantees that $\sum_{e \in q} w_2(e) > c_2$ for all shortest paths $q \in \mathcal{P}$, where $l_2(q) = l_2(p)$. Thus, it suffices to show that if the algorithm assigns weights $l_2(e) = w_1(e) + 2jw_2(e)$ and fails to find a feasible path w.r.t. constraint c_2, then no path p' for which

$$\sum_{e \in p'} w_1(e) + 2jw_2(e) > \sum_{e \in p} w_1(e) + 2jw_2(e) \tag{1}$$

will satisfy both

$$\sum_{e \in p'} w_1(e) + jw_2(e) < \sum_{e \in p} w_1(e) + jw_2(e)$$

and

$$\sum_{e \in p'} w_2(e) < c_2$$

when the value of k is reduced to j. In other words, it is useless to weight with the rule $l_2(e) = w_1(e) + jw_2(e)$ in order to search for a path p' whose $\sum_{e \in p'} w_1(e) + 2jw_2(e)$ is not minimum but satisfies the c_2 constraint.

Since path p violates the c_2 constraint, in order for the path p' to satisfy the c_2 constraint, we must have:

$$\sum_{e \in p} w_2(e) - \sum_{e \in p'} w_2(e) > 0. \tag{2}$$

Observe that Equation (1) can be rewritten as

$$\sum_{e\in p'} w_1(e) - \sum_{e\in p} w_1(e) > 2j(\sum_{e\in p} w_2(e) - \sum_{e\in p'} w_2(e)). \tag{3}$$

¿From (3) and (2), we have

$$\sum_{e\in p'} w_1(e) - \sum_{e\in p} w_1(e) > 0. \tag{4}$$

Based on (2) and (4), we know that the right-hand side and the left-hand side of the inequality (3) are positive. Thus, it can be implied that

$$\sum_{e\in p'} w_1(e) - \sum_{e\in p} w_1(e) > j(\sum_{e\in p} w_2(e) - \sum_{e\in p'} w_2(e)) \tag{5}$$

from which we conclude that

$$\sum_{e\in p'} w_1(e) + jw_2(e) > \sum_{e\in p} w_1(e) + jw_2(e).$$

This, in turn, implies that p' will not be selected by the algorithm, and thus the claim about the optimality of the proposed binary search scheme is true. ■

2.3 Approximation Bounds

Lemma 3 *If Phase 1 fails to return a feasible path w.r.t. both constraints, then it returns a path p that satisfies the c_2 constraint and whose $w_1()$ cost is upper bounded as follows:*

$$w_1(p) \le w_1(f) + (w_2(f) - w_2(p))/k$$

where f is a feasible path and k is the maximum value that the binary search determines at the end of Phase 1.

Lemma 4 *If Phase 2 fails to return a feasible path w.r.t. both constraints, then it returns a path p that satisfies the c_2 constraint and whose $w_1()$ cost is upper bounded as follows:*

$$w_1(p) \le w_1(f) + k(w_2(f) - w_2(p))$$

where f is a feasible path and k is the minimum value that the binary search determines at the end of Phase 2.

We will give the proof for Lemma 4. A similar argument can be used to prove Lemma 3.

Proof of Lemma 4: Let f be a feasible path for which $w_1(f)$ is the smallest possible among all feasible paths. Furthermore, assume that f is not dominated by any other feasible path. (We say that a feasible path dominates another path if it has shorter costs w.r.t. both $w_1()$ and $w_2()$ weights.) This implies that $w_2(f) > w_2(f')$ for any other feasible path f'. We know that $w_2(f')$ cannot be more than c_2 and it is less than $w_2(p)$; otherwise, the algorithm would have returned f'. Assume that the path p is infeasible. Since it is the shortest path, we have

$$w_1(p) + kw_2(p) \leq w_1(f) + kw_2(f). \tag{6}$$

In addition, the path p satisfies the constraint c_2. ¿From (6), we can write a bound on $w_1(p)$ as follows

$$w_1(p) \leq w_1(f) + k(w_2(f) - w_2(p))$$

■

These are the first approximation bounds ever proposed for the optimization version of the MCP problem. They can be used to obtain heuristic solutions for this problem.

3 Extensions of the Basic Algorithm

3.1 Finding a Path with the Closest Cost to a Constraint

From Lemmas 3 and 4, it is clear that one way to improve the performance of the basic algorithm is to minimize the difference $(w_2(f) - w_2(p))$. This minimization can be achieved by obtaining a path p for which $w_2(p)$ is as close as possible to c_2. This can be done with the following modification to the basic algorithm of Section 2.

After finding k, a DAG (directed acyclic graph) that contains all possible shortest paths w.r.t $l_1(e)$ (or $l_2(e)$, depending on the phase) is constructed. In fact, this can be done during the execution of the hierarchical Dijkstra's algorithm at no extra cost. A path p from this DAG is selected in such a way that $w_2(p)$ is maximized but is still less than or equal to c_2. Although, a path p with the maximum or minimum $w_2()$ weight can be found in the DAG, it is not easy to find a path p for which $w_2(p)$ is as close as possible to c_2 in polynomial-time. However, very efficient heuristics can be developed based on the fact that we can compute the maximum and the minimum $w_2()$ from the source to every node and from every node to the destination. Let the following labels be maintained for each node u: $M[u]$, $m[u]$, $\widetilde{M}[u]$, and $\widetilde{m}[u]$. Labels $M[u]$ and $m[u]$ indicate, respectively, the maximum and minimum $w_2()$ from the source to every node u. Labels $\widetilde{M}[u]$ and $\widetilde{m}[u]$ indicate, respectively, the maximum and minimum $w_2()$ from every node u to the destination. Labels $M[u]$, $m[u]$, $\widetilde{M}[u]$ and $\widetilde{m}[u]$ are determined by using a simple forward and backward topological traversal algorithm [1]. Considering the pairwise sum of these labels as follows, we can estimate a non-additive weight $\sigma(u, v)$ for every link (u, v) in the DAG to indicate how close $w_2()$ of the paths passing through the link (u, v):

$$\sigma(u,v) = \text{min\_non\_negative} \left\{ \begin{array}{l} c_2 - (M[u] + w_2(u,v) + \widetilde{M}[u[v]]), c_2 - (M[u] + w_2(u,v) + \widetilde{m}[u[v]]), \\ c_2 - (m[u] + w_2(u,v) + \widetilde{M}[u[v]]), c_2 - (m[u] + w_2(u,v) + \widetilde{m}[u[v]]), \\ +\infty \end{array} \right\}$$

where min_non_negative returns the minimum non-negative value. Then, the closest path to c_2 can be found via a simple graph traversal algorithm as follows. Starting from the source node s, the algorithm selects the link (s, u) with the minimum σ. It then goes to node u and again selects the link (u, v) with the minimum σ. The algorithm keeps selecting links with minimum σ until it hits the destination node. Figure 6 depicts an example of how a DAG of shortest paths is constructed. The original network is shown in Figure 6(a). Suppose a path p is to be found from s to t such that $w_1(p) \leq c_1 = 10$ and $w_2(p) \leq c_2 = 10$. Consider the case when $k = 1$, i.e., the algorithm

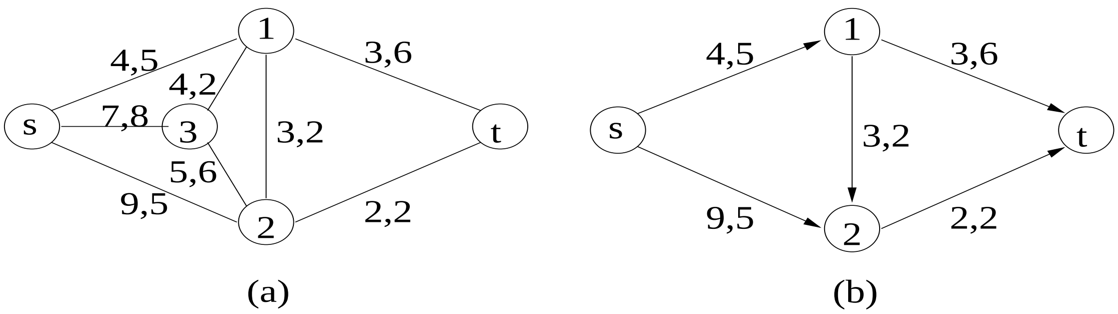

Figure 6: An example of a network and the DAG containing the three shortest paths from s to t.

minimizes $w_1(p) + w_2(p)$. There are three shortest paths from s to t: $p_1 =< s, 1, t >$ with $w_1(p_1) = 7$ and $w_2(p_1) = 11$, $p_2 =< s, 2, t >$ with $w_1(p_2) = 11$ and $w_2(p_2) = 7$, and $p_3 =< s, 1, 2, t >$ with $w_1(p_3) = 9$ and $w_2(p_3) = 9$. The corresponding DAG which contains all these shortest paths is shown in Figure 6(b). By traversing forward and backward on this DAG, we compute the labels $M[u]$, $m[u]$,

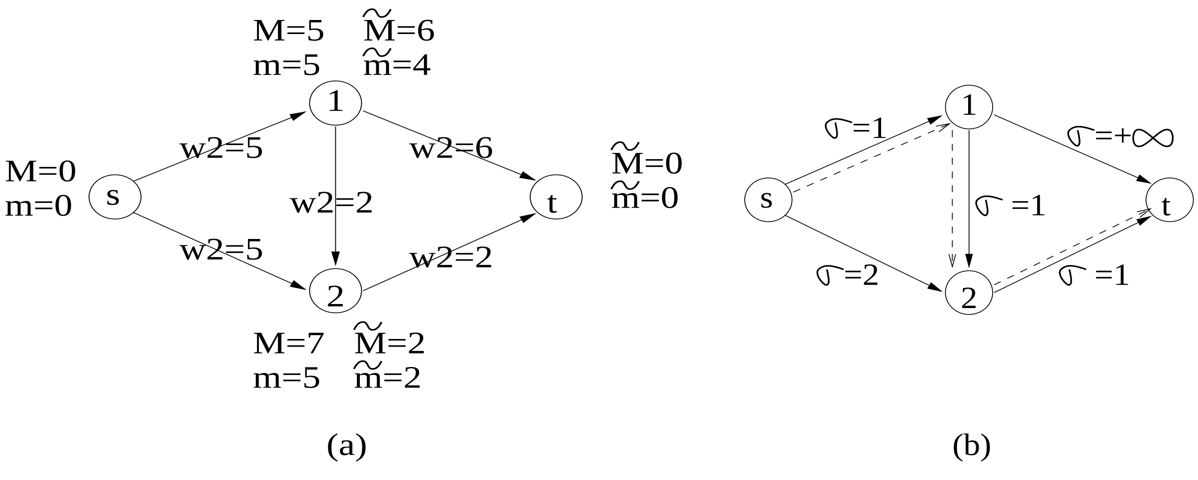

Figure 7: Finding the closest path to c_2.

$\widetilde{M}[u]$, and $\widetilde{m}[u]$ (see Figure 7(a)). After calculating σ for each link as shown in Figure 7(b), the algorithm first selects link $(s,1)$, followed by link $(1,2)$, and finally link $(2,t)$. Thus, the closest path p_3 is found. Since this heuristic step tends to minimize the additive difference in the approximation bounds presented in Lemmas 3 and 4, the returned path p is very likely to satisfy both c_1 and c_2.

3.2 Scaling

In some pathological cases, our basic algorithm will fail to return a feasible path that does exist. We illustrate this situation by an example. Consider the network in Figure 8(a). Suppose a path p is to be found from s to t such that $w_1(p) \leq c_1 = 10$ and $w_2(p) \leq c_2 = 10$. As shown in Figure 8(b),

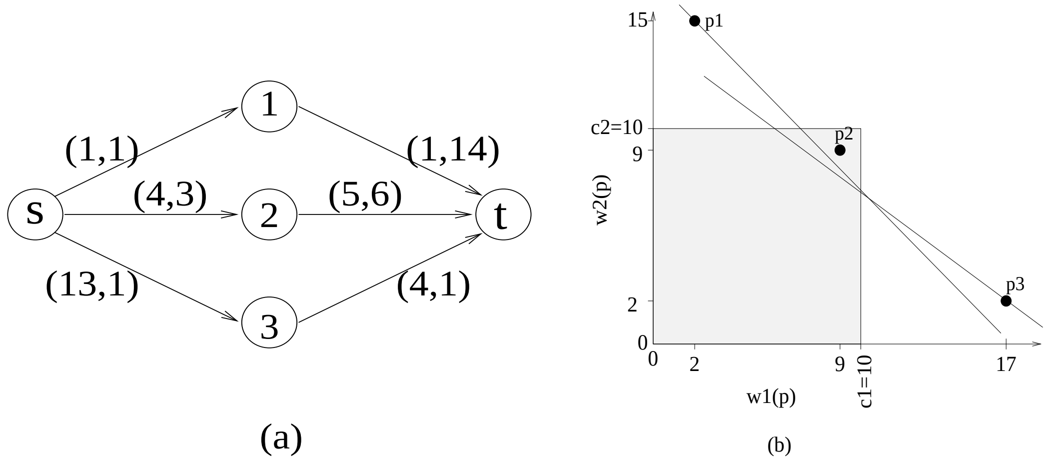

Figure 8: A scenario in which the basic algorithm fails to find a feasible path from s to t.

there are three paths from s to t: $p_1 =< s, 1, t >$ with $w_1(p_1) = 2$ and $w_2(p_1) = 15$, $p_2 =< s, 2, t >$ with $w_1(p_2) = 9$ and $w_2(p_2) = 9$, and $p_3 =< s, 3, t >$ with $w_1(p_3) = 17$ and $w_2(p_3) = 2$. Only p_2 is feasible. The approximation algorithm returns a path based on the minimization of the combined weight $l_1(p) = kw_1(p)+w_2(p)$ or $l_2(p) = w_1(p)+kw_2(p)$. To return the feasible path p_2, the algorithm needs to find an appropriate value for k such that $l_1(p_2)$ is less than both $l_1(p_1)$ and $l_1(p_3)$, or $l_2(p_2)$ is less than both $l_2(p_1)$ and $l_2(p_3)$. The value of k needs to be greater than 7/6 to satisfy $(l_1(p_2) = 9 + 9k) < (l_1(p_1) = 2 + 15k)$ and also less than 8/7 to satisfy $(l_1(p_2) = 9 + 9k) < (l_1(p_3) = 17 + 2k)$ (similar arguments are applicable to the $l_2()$ weight). However, it is impossible to find a value for k such that $7/6 < k < 8/7$. In other words, the approximation algorithm can not find the feasible path p_2 in the original graph. To circumvent such pathological cases, we provide an extension to our basic algorithm based on the scaling in [4]. A new weight $w_2'(e)$ is assigned to every link in the original graph as follows:

$$w_2'(e) = \lceil \frac{w_2(e) \cdot x}{c_2} \rceil \tag{7}$$

where x is an adjustable positive integer in the range $[1, c_2]$. The problem reduces to finding a path in the scaled graph such that $w_1(p) \leq c_1$ and $w_2'(p) \leq x$. It has been shown that a solution in the scaled graph is also a solution in the original one [4]. If we scale the network in Figure 8(a) by

$x = 3$, the scaled graph is shown in Figure 9(a). If the approximation algorithm uses the combined

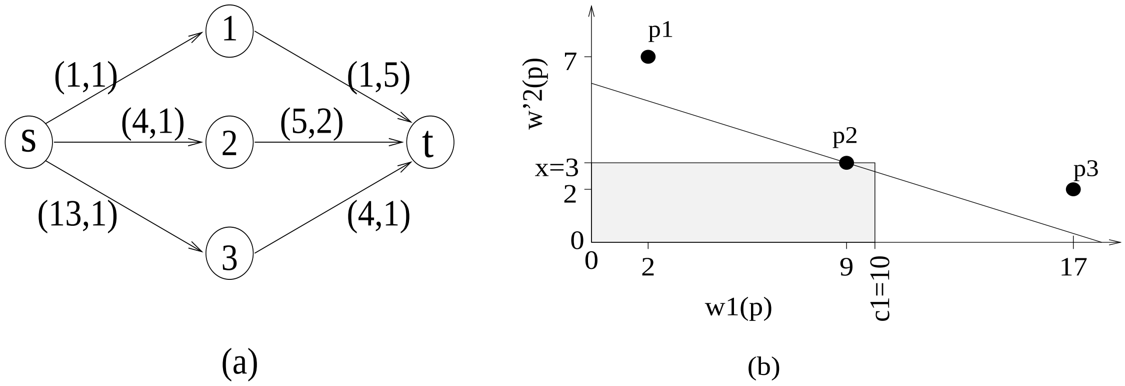

Figure 9: Scaling the network in Figure 8 by $x = 3$ allows the algorithm to find a feasible path.

weights $l_2(p) = w_1(p) + kw'_2(p)$ in the scaled graph with $k = 3$, it will return the feasible path p_2 (see Figure 9(b)), since $l_2(p_2) = 18$ is less than both $l_2(p_1) = 20$ and $l_2(p_3) = 23$.

Using the above scaling function, one may increase the number of shortest paths in the scaled graph. If we apply our basic approximation algorithm to the scaled graph, the algorithm will consider more shortest paths (in the scaled graph) in each iteration of Phase 1 (and Phase 2). It is intuitively true that the algorithm will terminate with a better value of k; a smaller k for Phase 2 and a larger k for Phase 1. Note that scaling is done only if both phases of the basic algorithm (i.e., without scaling) fail. If after scaling the algorithm fails again, we can try with a different value of x, and so on.

It is important to note that in contrast to the algorithm in [4], the value of x does not effect the complexity of our algorithm. Choosing x as small as possible may increase the number of shortest paths as desired. However, this also decreases the number of paths for which $w'_2(p) \leq x$, i.e., the algorithm may not return a feasible path. The tradeoff between the value of x and the associated performance improvement after scaling by x is shown in Figure 10. Here, we measure the performance of the path selection algorithm by the success ratio (more on that is discussed in Section 4).

When the basic algorithm fails to return a feasible path, we scale the graph using different values of x and run the algorithm again. Clearly, one needs to select an appropriate value for x to improve the performance. The following lemma shows that a binary search argument can be used to determine an appropriate x in the range $[1, c_2]$.

Lemma 5 *If the algorithm cannot find a path p for which $w'_2(p) \leq x$ in the scaled graph by x, then such a path cannot be found in a graph that is scaled by $x' < x$.*

Proof of Lemma 5: Let the graph G be scaled by $x = 2j$ for some integer j, and let $\mathcal{P}$ be the set of all possible paths in the scaled graph. If the algorithm fails to return a path p for which $\sum_{e\in p} \lceil \frac{w_2(e)\cdot 2j}{c_2} \rceil \leq 2j$, then

$$\sum_{e\in p} \lceil \frac{w_2(e)\cdot 2j}{c_2} \rceil > 2j \ \ \forall p \in \mathcal{P}. \tag{8}$$

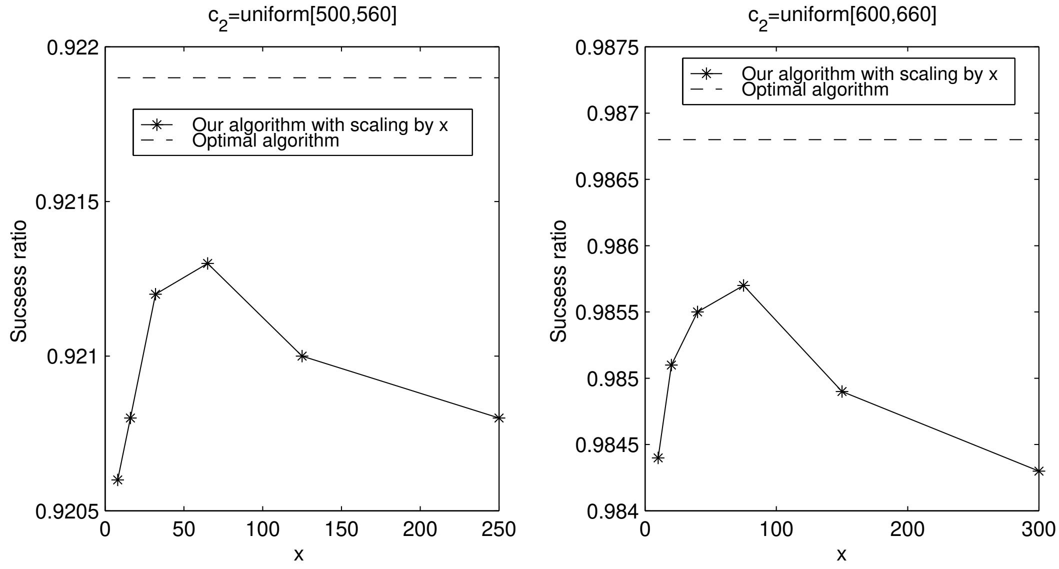

Figure 10: Performance of the path selection algorithm for different values of the scaling factor x.

In order to prove the lemma, it suffices to show that if (8) is true then the algorithm should never search for a path p' for which $w'_2(p') \stackrel{\text{def}}{=} \sum_{e\in p'} \lceil \frac{w_2(e)\cdot j}{c_2} \rceil \leq j$ when the links of the graph are scaled down by $x = j$. Since we know that

$$2\lceil \frac{w_2(e)\cdot j}{c_2} \rceil \geq \lceil \frac{w_2(e)\cdot 2j}{c_2} \rceil \tag{9}$$

we can rewrite (8) as

$$2\sum_{e\in p'} \lceil \frac{w_2(e)\cdot j}{c_2} \rceil > 2j \ \ \forall p' \in \mathcal{P} \tag{10}$$

from which we conclude

$$\sum_{e\in p'} \lceil \frac{w_2(e)\cdot j}{c_2} \rceil > j \ \ \forall p' \in \mathcal{P}. \tag{11}$$

This, in turn, implies that no path $p' \in \mathcal{P}$ will be selected by the algorithm, and the claim is true. ■

4 Simulation Results and Discussion

In this section, we contrast the performance of our basic algorithm with the heuristic algorithm in [4] and one of the two approximation algorithms in [11]. In [11] Jaffe presents two approximation algorithms for the MCP problem based on the minimization of $w_1(p) + dw_2(p)$, where $d = 1$ in the first algorithm and $d = \sqrt{c_1/c_2}$ in the second. Of the two approximations, the latter one provides better performance, and hence will be used in our comparison. As a point of reference, we also report the results of the optimal (exponential-time) algorithm, which considers all possible paths in the graph to determine whether there is a feasible path or not. The performance has been measured for various network topologies. For brevity, we report the results for two of these topologies.

4.1 Simulation Model and Performance Measures

In our simulation model, a network is given as a directed graph. Link weights, the source and destination of a connection request, and the constraints c_1 and c_2 are all randomly generated. If the path selection algorithm returns a feasible path for a connection request, we count this request as a *routed connection request*. In order to contrast the performance of various path selection algorithms, we use a measure called the *success ratio* (**SR**), which shows how often an algorithm finds a feasible path [4]:

$$\mathrm{SR} = \frac{Total\ number\ of\ routed\ connection\ requests}{Total\ number\ of\ connection\ requests}$$

Another important performance measure is the computational complexity. Since Dijkstra's algorithm is the basis for all path selection algorithms compared in this paper, we use the number of Dijkstra's iterations performed by these algorithms to compare their computational complexities. While the algorithm in [11] requires only one iteration, the algorithm in [4] always requires x^2 iterations, where x is an adjustable positive integer. The number of iterations in our algorithm varies in the range $[1, \log W]$, where W is the upper bound on the longest path according to one of the link weights. For our algorithm, the average number of Dijkstra's iterations (ANDI) per connection request is measured and compared with the deterministic number of Dijkstra's iterations in the other algorithms. It should be noted that our algorithm runs at its worst-case complexity only if it is deemed to fail, i.e., if the algorithm succeeds in finding a path, it will do so with much fewer Dijkstra's iterations than the ones needed in the worst case. As a result, the ANDI is expected to be much smaller than $\log W$. This is confirmed in the simulation results.

The computational complexity and performance of the basic algorithm can be significantly improved if it is possible to eliminate a large number of the infeasible paths in the network or to recognize, in advance, the unavailability of a feasible path from a source s to a destination t. This can be done using the following preprocessing step, which is executed before running the path selection algorithm. Let the following labels be maintained for each node u: $B_1[u]$, $B_2[u]$, $\widetilde{B}_1[u]$, and $\widetilde{B}_2[u]$. Labels $B_1[u]$ and $B_2[u]$ represent the total costs of the shortest paths from the source node s to node u based on the individual link weights w_1 and w_2, respectively. Labels $\widetilde{B}_1[u]$ and $\widetilde{B}_2[u]$ represent the total costs of the shortest paths from node u to the destination node t based on the individual link weights w_1 and w_2, respectively. Labels $B_1[u]$ and $B_2[u]$ are determined by using Dijkstra's algorithm [7], while labels $\widetilde{B}_1[u]$ and $\widetilde{B}_2[u]$ are determined by using Reverse-Dijkstra [1]. If $B_k[t] > c_k$ for any $k = 1, 2$ (i.e., the shortest path from source to destination exceeds the constraint c_k), then definitely there is no feasible path; thus, there is no need to run the path selection algorithm. Otherwise, one can eliminate most of the infeasible paths by performing the following graph reduction procedure. Consider an arbitrary link $(i, j) \in E$. If $B_k[i] + w_k(i, j) + \widetilde{B}_k[j] > c_k$ for any $k = 1, 2$, then the link (i, j) cannot be on any feasible paths since the total cost of any path that includes the link (i, j) already exceeds c_k. Thus, the link (i, j) can be pruned. This preprocessing step requires four iterations of Dijkstra's algorithm per connection request.

4.2 Results on Irregular Network Topology

We consider the network topology in Figure 11, which has been modified from ANSNET [6] by inserting additional links. Link weights are randomly selected with $w_1(i,j) \sim uniform[0, 50]$ and $w_2(i,j) \sim uniform[0, 200]$. The same network topology was used in [4]. For the different ranges

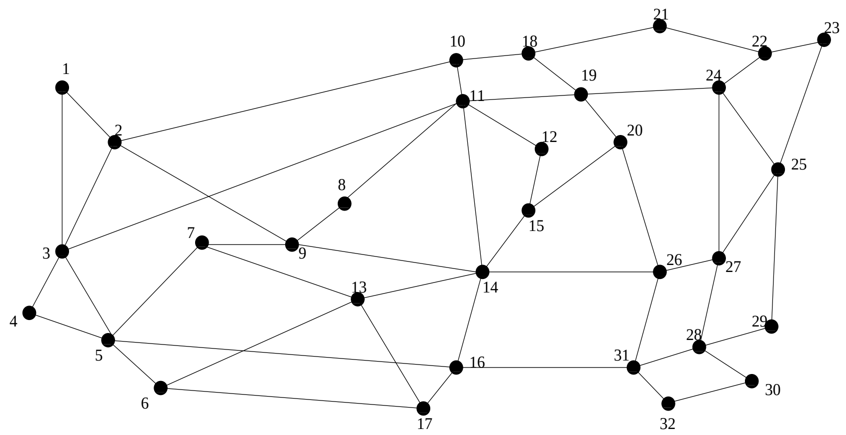

Figure 11: An irregular network topology.

of c_1 and c_2, Table 1 shows the SR and ANDI per connection based on twenty runs; each run is based on 2000 randomly generated connection requests. We run our basic algorithm after the

Range of c_1 and c_2	Optimal	Our Algorithm		Alg in [11]	Alg in [4]			
	SR	SR	ANDI	SR	x	SR	x	SR
$c_1 \sim uniform[100, 115]$ $c_2 \sim uniform[400, 460]$	0.7595	0.7589	7.23	0.7088	3	0.5043	118	0.7591
$c_1 \sim uniform[50, 65]$ $c_2 \sim uniform[200, 260]$	0.2594	0.2594	6.88	0.2505	3	0.2213	118	0.2594
$c_1 \sim uniform[75, 90]$ $c_2 \sim uniform[300, 360]$	0.5220	0.5220	8.10	0.4906	3	0.3851	120	0.5218
$c_1 \sim uniform[125, 140]$ $c_2 \sim uniform[500, 560]$	0.9219	0.9208	7.33	0.8674	3	0.5827	100	0.9213
$c_1 \sim uniform[150, 165]$ $c_2 \sim uniform[600, 660]$	0.9868	0.9845	5.64	0.9524	3	0.5974	30	0.9834

Table 1: Performance of several path selection algorithms under two additive constraints for the irregular topology in Figure 11.

above preprocessing step which incurs four additional Dijkstra's iterations per connection request.

For our algorithm, we compute the SR and ANDI which includes four Dijkstra's iterations in the preprocessing. In terms of SR, our algorithm performs as good as the optimal one at an average cost of about seven Dijkstra's iterations per connection request.

We compare our algorithm with the one in [4] from two points of view. First, we compare the SRs of the two algorithms under the same computational complexity. To do that, we set x in [4] to an appropriate value such that x^2 (number of Dijkstra's iterations in [4]) is approximately equal to the ANDI of our algorithm. In this case, the SR of the algorithm in [4] is significantly less than our algorithm (compare columns 3 and 7 in Table 1). Second, we compare the computational complexities that are needed to achieve the same SR under both algorithms. For this, we increase the value of x in [4] until both algorithms give *almost* the same SR. In this case, the algorithm in [4] requires about 1000 times more iterations of Dijkstra's algorithm than ours (compare column 4 and the square of column 8 in Table 1).

We also compare our algorithm with the one in [11]. Since link weights are uniformly distributed in these simulations, the algorithm in [11] performs good but slightly worse than our algorithm (compare columns 3 and 5 in Table 1). However, the algorithm in [11] does not perform well when link weights are distributed in a skewed manner, while our algorithm does. For example, consider the same network topology in Figure 11. Suppose we divide the network into three parts as shown in Figure 12, where now we assume skewed link weights as follows: $w_1(i,j) \sim uniform[70,85]$

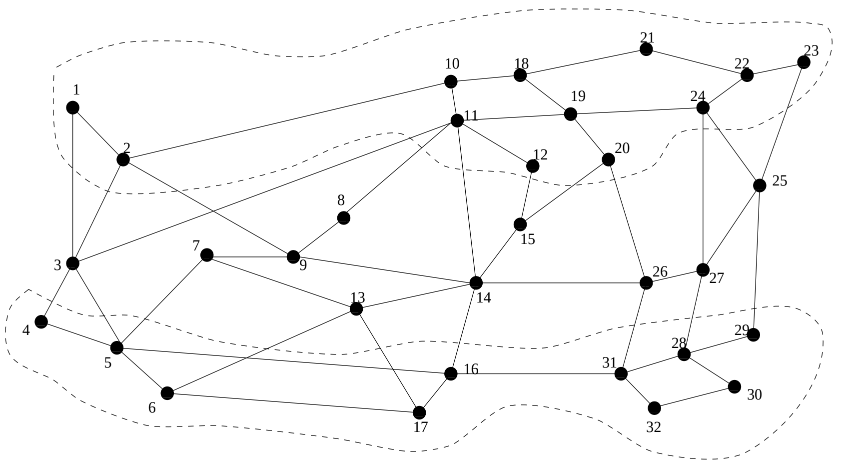

Figure 12: An irregular network topology is divided into three parts.

and $w_2(i,j) \sim uniform[1,5]$ if i is a node in the upper part; $w_1(i,j) \sim uniform[45,55]$ and $w_2(i,j) \sim uniform[45,55]$ if i is a node in the middle part; $w_1(i,j) \sim uniform[1,5]$ and $w_2(i,j) \sim uniform[70,85]$ if i is a node in the lower part. The source node is randomly chosen from nodes 1 to 7. The destination node is randomly chosen from nodes 20 to 32. For the different ranges of c_1 and c_2, Table 2 shows the resulting SR and ANDI. Our algorithm again performs very close to the optimal one and outperforms the SR of the algorithm in [11] by about 50% at the cost of ANDI presented in column 4.

Range of c_1 and c_2	Optimal	Our Algorithm		Alg in [11]	Improvement in SR
	SR	SR	ANDI	SR	
$c_1, c_2 \sim uniform[200, 215]$	0.1278	0.1278	22.90	0.0829	54%
$c_1, c_2 \sim uniform[215, 300]$	0.1739	0.1737	21.87	0.1210	43%
$c_1, c_2 \sim uniform[230, 250]$	0.2841	0.2740	19.84	0.1678	63%
$c_1, c_2 \sim uniform[250, 300]$	0.4572	0.4489	18.62	0.2771	62%
$c_1, c_2 \sim uniform[300, 360]$	0.8709	0.8323	13.34	0.5805	43%

Table 2: Performance comparison of our algorithm with the one in [11] under skewed link weights for the irregular topology in Figure 12.

The performance of our basic algorithm can be further improved using the extensions of Section 3. Finding the closest path to c_2 is definitely improves the performance if the number of shortest paths is large. However, this extension does not significantly improve the performance in our simulations because the number of shortest paths is very small. On the other hand, the scaling extension significantly improves the performance. The SR of our algorithm with scaling is almost equal to that of the optimal algorithm. For different ranges of c_1 and c_2, Table 3 shows the obtained SR and ANDI. Since a binary search is used to find an appropriate scaling factor x in the range $[1, c_2]$,

Range of c_1 and c_2	Optimal	Our Basic Alg		Our Alg with Scaling	
	SR	SR	ANDI	SR	ANDI
$c_1 \sim uniform[100, 115]$ $c_2 \sim uniform[400, 460]$	0.7595	0.7589	7.23	0.7593	8.71
$c_1 \sim uniform[50, 65]$ $c_2 \sim uniform[200, 260]$	0.2594	0.2594	6.88	0.2594	9.74
$c_1 \sim uniform[75, 90]$ $c_2 \sim uniform[300, 360]$	0.5220	0.5220	8.10	0.5220	9.37
$c_1 \sim uniform[125, 140]$ $c_2 \sim uniform[500, 560]$	0.9219	0.9208	7.33	0.9216	7.70
$c_1 \sim uniform[150, 165]$ $c_2 \sim uniform[600, 660]$	0.9868	0.9845	5.64	0.9862	6.37

Table 3: Performance comparison of our algorithm with and without scaling under two additive constraints for the irregular topology in Figure 11.

the worst-case complexity will be $\log c_1 \log W$ iterations of Dijkstra's algorithm. The improvement in the SR with scaling is achieved at an average extra cost of one or two Dijkstra's iterations per connection request.

4.3 Results on Regular Network Topology

Next, we consider a regular 10x10 mesh topology. As before, link weights are randomly selected with $w_1(i,j) \sim uniform[0,50]$ and $w_2(i,j) \sim uniform[0,200]$. For different ranges of c_1 and c_2, Table 4 shows the SR and ANDI from ten runs; each run is based on 2000 randomly generated connection requests. Once again, our algorithm gives the closest performance to the optimal one. The SR of our

Range of c_1 and c_2	Optimal	Our Algorithm		Alg in [11]	Alg in [4]			
	SR	SR	ANDI	SR	x	SR	x	SR
$c_1 \sim uniform[100,115]$ $c_2 \sim uniform[400,460]$	0.3085	0.3083	6.67	0.2853	3	0.1734	110	0.3081
$c_1 \sim uniform[150,200]$ $c_2 \sim uniform[500,600]$	0.5580	0.5570	7.82	0.5230	3	0.1974	170	0.7567
$c_1 \sim uniform[200,300]$ $c_2 \sim uniform[600,700]$	0.7596	0.7570	7.63	0.7214	3	0.1975	170	0.7575
$c_1 \sim uniform[300,380]$ $c_2 \sim uniform[650,750]$	0.8597	0.8582	6.92	0.8306	3	0.1975	170	0.8577
$c_1 \sim uniform[350,400]$ $c_2 \sim uniform[700,800]$	0.9060	0.9052	7.08	0.8798	3	0.1975	170	0.9048

Table 4: Performance of several path selection algorithms under two additive constraints for the regular 10x10 mesh topology.

algorithm is significantly greater than the algorithm in [4] under the same computational complexities (compare columns 3 and 7 in Table 4). To obtain almost the same SR, the algorithm in [4] requires about 3000 more iterations of Dijkstra's algorithm than ours (compare column 4 and the square of column 8 in Table 4). The SR of our algorithm is again slightly better than the algorithm in [11]. However, the real improvement of our algorithm over the one in [11] appears under nonuniformly distributed link weights and source-destination pairs (as was shown in Table 2).

5 Conclusions and Future Work

QoS-based routing subject to multiple additive constraints is an NP-complete problem that cannot be exactly solved in polynomial time. We presented an approximation algorithm to this problem for the case of two constraints. Our algorithm is supported by performance bounds that reflect the effectiveness of the algorithm in finding a feasible path. These bounds are the tightest ever reported to this problem. We studied the performance of the algorithm via simulations on regular and irregular topologies. Our results show that the proposed algorithm outperforms existing ones in complexity, performance, or both. We also presented two extensions to our basic algorithm that can be used to further improve its performance at little extra computational cost. The first extension, which is motivated by the presented theoretical bounds, attempts to find the closest feasible path to a constraint. The other extension, namely scaling, improves the likelihood of finding a feasible path by perturbing the linearity of the search process (or equivalently, changing the relative locations of the paths in the parameter space). Our basic approximation algorithm runs Dijkstra's algorithm up to $\log W$ times, where W is an upper bound on the longest path w.r.t. one of link weights. Specifically, $W = n \cdot \max\{w_i(e) \mid e \in E\}$, where $i = 2$ in Phase 1 and $i = 1$ in Phase 2. When scaling is used, the algorithm runs Dijkstra's algorithm up to $\log c_2 \log W$ times. These are worst-case complexities that are rarely used in practice. In fact, simulation results indicate much better average complexities for the basic algorithm and its extensions. The space complexity of our algorithm is $\mathcal{O}(n)$.

The path selection problem has been investigated in this paper assuming a flat network topology and complete knowledge of the network state. In practice, the true state of the network is not available to every source node at all times due to network dynamics, aggregation of state information (in hierarchical networks), and latencies in the dissemination of state information. Our future work will focus on investigating the MCP problem in the presence of inaccurate state information and the tradeoffs between the accuracy of the path selection process and that of topology aggregation (for spatial scalability) and/or the frequency of advertisements (for temporal scalability). Another aspect that we plan to investigate is that of renegotiation. When our algorithm fails to return a feasible path, it always returns a path which is close to satisfying the given constraints. Hence, we plan to investigate how such a path can be advantageously used in the renegotiation process to achieve further performance improvements.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Inc., 1993.

[2] A. Alles. ATM internetworking. White Paper, Cisco Systems, Inc., May 1995.

[3] G. Apostolopoulos, R. Guerin, S. Kamat, and S. K. Tripathi. Quality of service based routing: A performance perspective. In *ACM SIGCOMM'98*, Vancouver, British Columbia, Canada, August-September 1998.

[4] S. Chen and K. Nahrstedt. On finding multi-constrained paths. In *Proceedings of ICC'98 Conference*, pages 874 –879. IEEE, 1998.

[5] S. Chen and K. Nahrstedt. An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions. *IEEE Network*, 12(6):64–79, Nov-Dec 1998.

[6] D. E. Comer. *Internetworking with TCP/IP*, volume I. Prentice Hall, Inc., third edition, 1995.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT press and McGraw-Hill book company, sixteenth edition, 1996.

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

[9] R. Guerin and A. Orda. QoS-based routing in networks with inaccurate information: Theory and algorithms. In *Proceedings of the INFOCOM'97 Conference*, pages 75–83. IEEE, 1997.

[10] A. Iwata and et. al. ATM routing algorithms with multiple QOS requirements for multimedia internetworking. *IEICE Trans. Commun.*, E79-B(8):999–1006, August 1996.

[11] J. M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14:95–116, 1984.

[12] K. Lee and et. al. QoS based routing for integrated multimedia services. In *Proceedings of the GLOBECOM'97 Conference*, volume II, pages 1047–1051. IEEE, 1997.

[13] W. C. Lee, M. G. Hluchyi, and P. A. Humblet. Routing subject to quality of service constraints in integrated communication networks. *IEEE Network*, pages 46–55, July/August 1995.

[14] D. H. Lorenz and A. Orda. QoS routing in networks with uncertain parameters. In *Proceedings of the INFOCOM'98 Conference*, volume 1, pages 3–10. IEEE, March-April 1998.

[15] Q. Ma and P. Steenkiste. Quality-of-service routing for traffic with performance guarantees. In *Proceedings of the 4th International IFIP Workshop on Quality of Service*, May 1997.

[16] C. Pornavalai, G. Chakraborty, and N. Shiratori. QoS based routing algorithm in integrated services packet networks. In *Proceedings of the ICNP'97*, pages 167–174. IEEE, 1997.

[17] R. Vogel and et. al. QoS-based routing of multimedia streams in computer networks. *IEEE Journal on Selected Areas in Communications*, 14(7):1235–1244, September 1996.

[18] Z. Wang. On the complexity of quality of service routing. *Information Processing Letters*, 69(3):111–114, 1999.

[19] Z. Wang and J. Crowcroft. Bandwidth-delay based routing algorithms. In *Proceedings of the GLOBECOM'95 Conference*, volume 3, pages 2129–2133. IEEE, Nov. 1995.

[20] Z. Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1228–1234, September 1996.