UC Santa Cruz

UC Santa Cruz Previously Published Works

Title

Solving The Multi-Constrained PathSelection Problem by Using Depth First Search

Permalink

<https://escholarship.org/uc/item/7k49v3r3>

Author

Garcia-Luna-Aceves, J.J.

Publication Date

2005-08-24

Peer reviewed

Solving The Multi-Constrained Path Selection Problem By Using Depth First Search¹

Zhenjiang Li[†], J.J. Garcia-Luna-Aceves^{† ‡}

[†]Department of Computer Engineering, University of California, Santa Cruz
Santa Cruz, CA 95064, U.S.A.

Phone: 1-831-4595436, Fax: 1-831-4594829

Email: {zhjli,jj}@soe.ucsc.edu

[‡]Palo Alto Research Center (PARC), 3333 Coyote Hill Road, Palo Alto, CA 94304

Abstract

An extended depth-first-search (EDFS) algorithm is proposed to solve the multi-constrained path (MCP) problem in quality-of-service (QoS) routing, which is NP-Complete when the number of independent routing constraints is more than one. EDFS solves the general k -constrained MCP problem with pseudo-polynomial time complexity $O(m^2 \cdot EN + N^2)$, where E and N are the number of links and nodes of a graph respectively, and m is the maximum number of feasible paths maintained for each destination. This is achieved by deducing potential feasible paths from knowledge of previous explorations, without re-exploring finished nodes and their descendants in the process of the DFS search. One unique property of EDFS is that the tighter the constraints are, the better the performance it can achieve, w.r.t. both time complexity and routing success ratio. Analysis and extensive simulation are conducted to study the performance of EDFS in finding feasible paths that satisfy multiple QoS constraints. The main results show that EDFS is insensitive to the number of constraints, and outperforms other popular MCP algorithms when the routing constraints are tight or moderate. The performance of EDFS is comparable with that of the other algorithms when the constraints are loose.

1. Introduction

Routing algorithms supporting QoS differentiation differ from conventional routing algorithms in that, in QoS routing, the path from the source to the destination needs to satisfy multiple constraints simultaneously (e.g., band-

width, reliability, end-to-end delay, jitter and cost), while in conventional routing, routing decisions are made based only on a single metric. QoS-related routing metrics, as well as the corresponding constraints associated with them, can be categorized into minimal (maximal) metrics and additive metrics. A typical minimal metric is bandwidth, for which the end-to-end path bandwidth is determined by the minimal residual bandwidth of the links along the chosen path. It is relatively easy to deal with a routing constraint on minimal metric, because all the links whose residual bandwidth do not satisfy the requirement can simply be dropped. However, it is well known that path selection subject to two or more independent additive metrics is NP-complete [8], which means that there is no efficient (polynomial) exact solution for the general k -constrained MCP (multi-constrained path) selection problem. There also exist multiplicative metrics, such as loss rate, in which the end-to-end path loss is equal to the product of the loss rates of all intermediate links. Multiplicative metrics can be translated into additive metrics, or vice versa, by taking logarithmic or exponential function respectively. Therefore, we only consider additive QoS metrics and constraints in this work.

Amongst all MCP problems, routing subject to two constraints has drawn the most interest, which includes a special case - the **restricted shortest path (RSP)** problem, where the goal is to find the path that satisfies one constraint while optimizes another metric simultaneously. The MCP (with two constraints) and the RSP problems are not strong NP-complete in that there are pseudo-polynomial running time algorithms to solve them exactly, in which the computational complexity also depends on the values of link weight in addition to the network size [2]. However, their complexity is prohibitively high when the values of link weight become large.

Based on the latter observation above, Chen and Nahrstedt [1] proposed to scale one component of the link weight

¹This work was supported in part by the National Science Foundation under Grant CNS-0435522 and by the Baskin Chair of Computer Engineering at UCSC

down to an integer that is less than $\lceil \frac{w_i \cdot x}{c_i} \rceil$, where x is a pre-defined integer and c_i is the corresponding constraint on the weight component w_i . They prove that the problem after weight scaling is polynomially solvable by an extended version of Dijkstra's (or Bellman-Ford) shortest path (SP) algorithm, and any solution to the latter is also a solution to the original MCP problem. The running time is $O(x^2 N^2)$ when the extended Dijkstra's SP algorithm is used; and it is $O(xEN)$ when the extended Bellman-Ford SP algorithm is used, where E and N are the number of edges and nodes, respectively.

Neve and Mieghem proposed TAMCRA [7], where k – *shortest* paths are kept at each node w.r.t. a cost function of link weight components, hoping that one of them will satisfy the routing constraints. TAMCRA has computational complexity $O(kN \log(kN) + k^3 CE)$, in which C is the number of constraints. Obviously, the performance and complexity of this heuristic depends on the value of k . Better performance can be achieved with a larger k at the cost of more execution time. Another common approach to solving the MCP problem is to choose a good aggregation link-cost function based on the link weight components and the given constraints. Then any shortest path algorithm, such as Dijkstra's or Bellman-Ford, can be used to compute the shortest path w.r.t. the aggregated link cost. Jaffe [2] was the first to use a linear link-cost function $l_{u,v} = \alpha w_1(u,v) + \beta w_2(u,v)$, in which $\alpha, \beta \in \mathbb{Z}^+$. Ever since, both linear aggregation functions and non-linear aggregation functions have been proposed to solve the MCP problem.

Compared with RSP, the k –*constrained* MCP problem has received far less attention. If a scheduling algorithm (e.g., weighted fair queuing) is used, then queuing delay, jitter and loss rate can be described as a function of bandwidth, such that the original NP-Complete MCP problem can be reduced to a traditional shortest path routing problem [5]. However, this is not true for propagation delay and is only applicable to networks using specific scheduling mechanisms. Yuan [9] generalized the ideas of link-weight scaling and k – *shortest* paths routing as limited-granularity heuristic and limited-path heuristic, respectively, and also proposed an algorithm to solve the general k –*constrained* MCP problem based on an extended Bellman-Ford algorithm. The running time of Yuan's algorithm is $|X| \cdot NE$, where $|X|$ is the size of the table maintained at each node, for the limited granularity heuristic; and it is $x^2 NE$, where x is the number of paths maintained at each node, for the limited path heuristic. Again, the performance and computational overhead of Yuan's heuristics depends on the number of possible values to which link weight can be scaled down, or the number of paths to maintain at each node.

In this paper, we propose EDFS, an algorithm based on depth-first search, to solve the general k – *constrained*

Figure 1. Incomparable paths in $\{D \times C\}$

MCP. EDFS has time complexity $O(m^2 \cdot EN + N^2)$, where m is the maximum number of non-dominated paths maintained for each destination. This performance is achieved by deducing potential feasible paths from knowledge of previous explorations without re-exploring finished nodes and their descendants. One unique property of EDFS is that the tighter the constraints are, the better the performance it can achieve. In particular, EDFS can achieve almost the same success ratio as an exact solution does (with exponential running time complexity), while having less running time than that of Dijkstra's algorithm when the routing constraints are very tight.

The rest of the paper is organized as follows. First we give the network model and notations we are using in our discussion, including necessary background information about QoS routing. Then the basic idea behind the EDFS algorithm is illustrated and the details are demonstrated by its pseudo-code. The time complexity and performance of our algorithm are analyzed and examined by extensive simulations, in which we show how EDFS solves multiple-constrained path selection problem efficiently and effectively. To conclude, we summarize our work at the end of this paper.

2. Network Model and Problem Formulation

We model the network as a directed graph $G = \{V, L\}$. Here, V is the set of nodes and L is the set of links interconnecting the nodes. That is, for node u and v in V , the link $l_{u,v}$ is in L if u and v are directly connected in G . In our discussion, we assume that each link $l_{u,v}$ is associated with a link weight vector $w_{u,v} = \{w_1, w_2, \dots, w_k\}$, in which w_i is an individual weight component. Accordingly, any path from the source node to the destination node can be assigned a path weight vector $w(p) = \{w_1^p, w_2^p, \dots, w_k^p\}$, where w_i^p equals to the sum of the corresponding weight components of all the links in the path.

It has been pointed out that only those non-dominated paths (or incomparable paths) need to be maintained in multi-constrained routing [6]. Path p is dominated by path

q if

$$w_i^q \leq w_i^p, \text{ for } i = 1, 2 \dots k \quad (1)$$

A path is called non-dominated if it is not dominated by any other path. The concept of path domination allows us to restrict the computational complexity by maintaining only those non-dominated paths, because the capability of QoS provisioning from the source node to the destination node can be represented by the set of non-dominated paths. For example, Figure 1 (a) shows a set of points in $\{\text{Delay} \times \text{Cost}\}$ representing the incomparable paths between a source-destination pair when the link weights are cost and delay. In Figure 1 (a), the path weight represented by point X is dominated by B , which has shorter delay and less cost. However, there exists no clear *better or worse* relation between any two paths of $\{A, B, C, D, E\}$, because none dominates another. Any routing request falls in the feasible area (shadowed area I of Figure 1 (b)) can be supported by at least one of the incomparable paths. The MCP problem can be formally formulated as follows.

MCP: Given a directed graph $G = \{V, L\}$, where V is the set of nodes and L is the set of links. each link $\{l_{u,v} \in L\}$ is assigned a link weight vector $w_{u,v} = \{w_1, w_2 \dots w_k\}$. A corresponding constraint vector is given as $C = \{c_1, c_2 \dots c_k\}$, the problem is to find a feasible path p from the source node s to the destination node t such that

$$w_i^p \leq c_i \text{ for } i = 1, 2 \dots k$$

$$\text{where } w_i^p \triangleq \sum_{l_{u,v} \in p} w_i(u, v) \quad (2)$$

QoS routing consists of disseminating a consistent view of the network (including network topology and resource state information) to each router, and a QoS routing algorithm responsible for finding feasible paths from the source to each destination satisfying multiple constraints. In this paper, we only consider the later, and assume that there exists a link-state routing protocol that disseminates topology and resource information to all routers in a timely manner.

3. Extended Depth-First-Search Algorithm

In depth-first search (DFS), edges (we use edge and link interchangeably) are explored away from the most recently discovered node v that still has unexplored adjacent outgoing edges. When all of v 's edges have been explored, DFS *backtracks* to explore unscanned edges leaving the node from which v was discovered. This process continues, until all the nodes that are reachable from the source are discovered. The edges of a directed graph can be sorted into four groups w.r.t. a DFS search on it: tree, backward, forward and cross edges. An edge $l_{u,v}$ is a tree edge if node v was first discovered by exploring $l_{u,v}$, and the tree having all the tree edges is named a *DFS tree*. Link $l_{u,v}$ is a back edge

Figure 2. Illustration of EDFS¹

if $l_{u,v}$ leads u to a ancestor v in the *DFS tree*. Link $l_{u,v}$ is a forward edge if it connects u to a descendant v in the *DFS tree*. All other edges are cross edges.

Based on the type of edge explored in DFS, we extend DFS into a multi-constrained path searching algorithm based on the following observations. Because a tree edge always leads to a previously discovered node u_d , we can add the path from the source to u_d , together with its capability (path weight, more specifically), into the routing table. Backward edges form cycles and the search proceeds beyond a node that is already in the tree, given that the metrics that we are considering are additive. A forward or cross edge always leads to a finished node u_f , which means that: (a) all the descendants of u_f have been discovered and finished in a previous exploration away from u_f , and (b) one or more paths to u_f and its descendants are already known, node u_f is reached again because a new path p_{new} to node u_f is used (we call p_{new} the *active path*). Therefore, non-dominated paths to u_f and its descendants can be deduced without exploring away from u_f once more. Based on the observations above, we can tell if any improvement on existing paths can be achieved by using p_{new} . This is possible because routing metrics are additive, and paths are *comparable* by using the concept of path domination. Here we

¹(a): Detect loop (2,g,4,i,5,j,2) when exploring backward edge j . Find dominating path (1, b, 6, c, 2) to node 2 via cross edge c , $\Delta = w(1, b, 6, c, 2) - w(1, a, 2) = (-1, -2)$. For each descendant of node 2, improve $w(1, a, 2, f, 3) + \Delta = (6, 7) + (-1, -2) \rightarrow w(1, b, 6, c, 2, f, 3) = (5, 5)$ for 3; improve $w(1, a, 2, g, 4) + \Delta = (8, 7) + (-1, -2) \rightarrow w(1, b, 6, c, 2, g, 4) = (7, 5)$ for 4; improve $w(1, a, 2, g, 4, i, 5) + \Delta = (10, 9) + (-1, -2) \rightarrow w(1, b, 6, c, 2, g, 4, i, 5) = (9, 7)$ for 5. (b): Find incomparable path $w(1, b, 6, e, 7)$ to node 7 via forward edge e , $\Delta = w(1, b, 6, e, 7) - w(1, b, 6, d, 7) = (-2, 2)$. For node 8, the descendant of node 7, compute new incomparable path $w(1, b, 6, d, 7, h, 8) + \Delta = (8, 7) + (-2, 2) \rightarrow w(1, b, 6, e, 7, h, 8) = (6, 9)$.

Algorithm 1 Procedure EDFS and EDFS_Visit

```
1: procedure EDFS( $G, src$ )
2:   for all  $u \in V$  do
3:      $u.color \leftarrow White$ 
4:      $\pi(u) \leftarrow nil$   $\triangleright \pi(u)$  is  $u$ 's predecessor
5:   end for
6:   EDFS_Visit( $src$ )
7: end procedure

8: procedure EDFS_Visit( $u$ )
9:    $u.color \leftarrow Grey$ 
10:   $p.push(u)$   $\triangleright p$  is current active path
11:   $UpdateRouteEntry(u, p, w(p), dTable)$ 
12:  for all  $v \in Adj(u)$  do  $\triangleright$  explore adjacent edge  $l_{u,v}$ 
13:    if  $v.color = White$  then
14:       $\pi[v] \leftarrow u$ 
15:       $w(p) \leftarrow w(p) + w_{u,v}$ 
16:      EDFS_Visit( $v$ )
17:       $w(p) \leftarrow w(p) - w_{u,v}$ 
18:    else if  $v.color = Grey$  then
19:      continue  $\triangleright$  loop, scan next adjacent edge
20:    else if  $v.color = Black$  then  $\triangleright v$  is finished
21:       $w(p) \leftarrow w(p) + w_{u,v}$ 
22:       $p.push(u)$ 
23:       $UpdateRouteEntry(u, p, w(p), dTable)$ 
24:       $p.pop(u)$ 
25:       $w(p) \leftarrow w(p) - w_{u,v}$ 
26:    end if
27:  end for
28:   $u.color \leftarrow Black$ 
29:   $MergeDescendant(\pi(u), u)$ 
30:   $p.pop(u)$ 
31: end procedure
```

have three possibilities. First, the active path p_{new} to the finished node u_f is worse than any existing path p_{old} for u_f . In this case, EDFS just ignores p_{new} and operates as the basic DFS. Second, if p_{new} is an incomparable path from the source to u_f , then we can expect that new incomparable paths may be found to the descendants of node u_f by following p_{new} . Third, if the path p_{new} dominates (i.e. is better than) the path p_{old} , then better path to node u_f and better paths to its descendants can be obtained, if any.

Algorithm 1 shows the main procedures of the extended version of depth-first search (EDFS), and Algorithm 2 defines the functions called by EDFS. As we can see, one new structure - the descendant table $dTable$ is used to keep track of the descendants of each node in the processing of DFS searching, which enables us to deduce possible dominating or incomparable paths when meeting a finished node. Note that EDFS specified in algorithm 1, 2 actually is able to find a set of incomparable paths with different QoS provisioning capabilities for each node reachable from the source. With minor modification, EDFS can stop searching right after a feasible path to the specified destination was found satisfying the given routing request.

Figure 2 illustrates the basic idea of EDFS, in which we show how EDFS detects a routing loop, and finds dominating or incomparable paths for finished nodes and their descendants without re-exploring away from the finished nodes. We assume that each link is associated with two weight components, and when exploring away from a node, the adjacent edges are scanned alphabetically. Nodes are

Algorithm 2 Function $UpdateRouteEntry$, $UpdateDescRoutes$ and $MergeDescendant$

```
1: function UPDATEROUTEENTRY( $dst, p, w(p), dTable$ )
2:   if no path for  $dst$  then
3:     insert  $p$ 
4:   else
5:     for each existing path  $p_i$  for  $dst$  do
6:       if  $p$  worse than  $p_i$  then
7:         return
8:       else if  $p$  better than  $p_i$  then
9:          $UpdateDescRoutes(dst, p, p_i, dTable)$ 
10:        delete  $p_i$ , insert  $p$ 
11:       else if  $p$  incomparable with  $p_i$  then
12:          $UpdateDescRoutes(dst, p, p_i, dTable)$ 
13:        insert  $p$ 
14:       end if
15:     end for
16:   end if
17: end function

18: function UPDATEDESCROUTES( $u, p_{new}, p_{old}, dTable$ )
19:   for each node  $d_i \in desc[u]$  do
20:     for each path  $p_j$  for  $d_i$  beginning with  $p_{new}$  do
21:       if  $p_{new}$  better than  $p_{old}$  then
22:         improve the existing path  $p_j$  for  $d_i$ 
23:       else if  $p_{new}$  incomparable with  $p_{old}$  then
24:         compute new incomparable path for  $d_i$ 
25:       end if
26:     end for
27:   end for
28: end function

29: function MERGEDESCENDANT( $dad, kid$ )
30:    $desc[dad].push(kid)$ 
31:   for each descendant  $d_i \in desc[kid]$  do
32:      $desc[dad].push(d_i)$ 
33:   end for
34: end function
```

also colored white or black to distinguish that they are undiscovered or finished respectively, and double circled nodes are discovered but not finished yet. As seen in Figure 2, a routing loop can be easily detected whenever a grey node is re-visited via backward edges. A cross or forward edge leads the DFS search to a previously finished node u_f . If the current active path p_f^{active} to u_f is dominating or incomparable with any existing path p_f^{old} to u_f , given that routing metrics are additive, we can have new dominating or incomparable path p_d^{new} for each descendant d of u_f by replacing the subpath p_f^{old} of the existing path p_d^{old} for d with the active path p_f^{active} , and compute the corresponding path weight as follows.

$$w(p_d^{new}) = w(p_d^{old}) + [w(p_f^{active}) - w(p_f^{old})] \quad (3)$$

Here '+', '-' are normal addition and subtraction operations on vectors of real numbers.

4. Computational Complexity

In this section, we obtain the worst case running time complexity of EDFS. In Algorithm 1, the loop on lines 2-5 of EDFS takes $O(N)$ time for initialization. The function EDFS_Visit is called exactly once for each node $u \in V$, because EDFS_Visit is invoked only on white

Figure 3. Time complexity of EDFS

nodes and the first thing it does is to paint the node *Grey*. During an execution of *EDFS_Visit*, the loop on lines 12-27 of *EDFS_Visit* is executed $|Adj[u]|$ times, where $|Adj[u]|$ is the number of adjacent outgoing edges of u . Because $\sum_{u \in N} |Adj(u)| = \Theta(E)$, the total cost of executing lines 12-27 of *EDFS_Visit* is $O(E)$. Therefore, within the loop on lines 12-27, function *UpdateRouteEntry* on line 23 is called at most $O(E)$ times, while outside the loop on lines 12-27, the function *UpdateRouteEntry* and *MergeDescendant* are called exactly once for each node because each node is painted *Grey* and *Black* only once. In Algorithm 2, the function *UpdateDesctRoutes* is called at most m times by *UpdateRouteEntry*, where m is the maximum number of non-dominated paths we maintain for each node. The nested loops on line 19-27 take $O(mN)$ time to update existing paths to each descendant of node u . Because the *push* and *pop* operation take constant time, the loop on lines 31-33 of *MergeDescendant* costs at most time $O(N)$. Summarily, we have

$$\begin{aligned} O(EDFS) &= O(N) + O(N) + O(E \cdot m \cdot mN) + O(N^2) \\ &= O(m^2 \cdot EN + N^2) \end{aligned} \quad (4)$$

To verify the correctness of our analysis, we conduct simulations running EDFS on networks of different sizes ranging from 10 nodes up to 5000 nodes. The network topologies in the simulations are randomly generated graphs (pure random graphs with $p_r = 0.11$, as described later), and we do not limit the number of incomparable paths maintained for each node. The number of constraints is three and the source-destination pairs of requests are randomly chosen from the network. From Figure 3 we can see that the simulation time samples of EDFS "resembles" the

curve of the polynomial function $O(EN + N^2)$. Because we do not limit the number of incomparable paths maintained for each node, this result actually confirms the work done by Kuipers and Mieghem [4], where they have shown that, in practice, the worst case under which the number of non-dominated paths grows exponentially large hardly happens. However, to guarantee that the running time is polynomially bounded, we must specify the maximum number of incomparable paths maintained for each node. It is again a trade-off between performance and complexity. In what follows, we assign $m = 5$ in all our simulation configurations unless it is specified otherwise. Simulation results show that this is sufficient to achieve satisfactory performance in most scenarios.

5. Extensions to the Basic Algorithm

5.1. Exploring with different sequences

For a given network, the DFS tree can be different if we scan the outgoing links at each node with different orders. As a consequence, from the same source, we may have a different set of incomparable paths for each reachable node with a different exploring sequence of nodes. By executing EDFS multiple times with different exploring sequences and combining the results of them, we will find better or more incomparable paths for nodes reachable from the source. As we will see shortly, Only one to three runs of EDFS are sufficient to achieve nearly optimal solution when the routing constraints are tight, while three to five runs of EDFS are needed to achieve satisfactory performance when the constraints are loose.

5.2. Crankbacking

If the constraints are known in advance, it is intuitive that we do not need to go further deeper when we detect that current active path has already violated the requirements. It follows immediately that we can either continue to scan the next unexplored outgoing edge of node u , or *crankback* to the predecessor from which u was first discovered when there is no outgoing edge of u we can use to extend the current active path. Based on our simulations, crankbacking has two effects on the performance of EDFS. First, it cuts down the running time of EDFS by orders of magnitude, especially when the constraints are tight. Secondly, the success ratio of EDFS can be improved significantly. Simulation results show that the increase of the success ratio can be as much as 8 to 10% even with only a single execution of EDFS. The reason is that crankbacking actually equals to guiding EDFS searching to avoid unnecessary exploration, from which considerable time is saved, and the feasible path is more likely to be discovered in the first place. As a result,

this also allows us to run more executions of EDFS with different exploring sequences within the same time limit, such that higher success ratio can be achieved.

5.3. Sorting links

When the time does not allow running EDFS multiple times, it is critical to decide which outgoing edge to explore first at each node u . It is preferable for the new active path (obtained after extending it with one outgoing edge of u) to have as large a margin for the given constraints as possible. However, this is hard to tell without actually searching into the network. To deal with this problem, we sort the outgoing edges of every node u w.r.t. a certain parameter, which we call the normalized margin (NM). $NM = \prod_{i=1}^k (1 - \frac{w_i(p)}{c_i})$, where p is the new active path after extending one outgoing edge of node u . The outgoing edge with the maximal NM is explored first. In our simulation, we find that a 3 to 5% increase of success ratio can be achieved when the number of EDFS we execute is small (1 to 3 runs). The performance improvement becomes negligible when EDFS is called more times because most of the possible exploration sequences are exploited in the first few runs.

6. Performance Evaluation

6.1. Performance comparison with 2 constraints

Three topologies and corresponding results are chosen to present here from all topologies we simulated: ANSNET, *Pure-random* graph and *Waxman* graph. ANSNET (32 nodes and 54 links) is widely used by Chen and Nahrstedt of [1] and other researchers to study QoS routing algorithms. In pure-random graphs, the existence of the link between any two nodes is determined by a pre-defined constant probability $\{p_r | 0 < p_r < 1\}$. It is purely random in the sense that p_r is independent of any other factors, such as the distance between the two nodes. However, this usually is not true because, in practice, the probability for two nodes at far apart to have a direct connection is much lower than that for two nodes close by. Waxman's model takes this into account, in which the probability p_r is defined as $p_r = \alpha e^{-\frac{d}{\beta L}}$, $0 < \alpha, \beta < 1$, where d is the distance between these two nodes and L is the maximal distance between any two nodes in the graph. For the Waxman and pure-random graphs used in simulation, the field size of the simulation is a 15×10 rectangle, and the number of nodes is determined by the Poisson distribution with modified parameter λA , where λ is the original intensity rate and A is the field size. All nodes are randomly distributed within the simulation field at randomly chosen positions.

As the first step, we compare the performance of different MCP algorithms subject to only two constraints.

Figure 4. ANSNET, 32 nodes and 54 links

The MCP algorithms we use to compare with EDFS are CN (Chen and Nahrstedt [1]), KKT (Korkmaz, Krunk and Tragoudas [3]) and JSP (a variation of Jaffe's algorithm: Dijkstra's shortest path algorithm w.r.t. the aggregated link cost function $w(u, v) = [\frac{w_1(u, v)}{c_1} + \frac{w_2(u, v)}{c_2}]$). As we mentioned early, CN maps the original MCP problem into a scaled version, in which one of the link weight components is scaled down within the range of $(0, x]$ by $w'_i(u, v) = \lceil \frac{w_i(u, v) \cdot x}{c_i} \rceil$. They prove that there exists polynomial solution to the *scaled* version of MCP, and any solution to the latter is also a solution to the original MCP problem. The basic idea of KKT is to run Dijkstra's SP algorithm w.r.t. a linear aggregation function of the two link weight components $\{w_{u,v} = w_i + k * w_j, i = 1 \text{ or } 2\}$, and the coefficient k is self-adjustable according to whether the algorithm can find a feasible path for a given request with current values of i and k . Another execution of Dijkstra's shortest path algorithm is invoked with new i and k if no feasible path was found with current values. As the baseline, we also implement an exact solution, which has exponential running time, but can give all feasible paths for a given routing request.

For ANSNET, the first link weight component is uniformly distributed in $(0, 50]$, while the second is uniformly distributed in $(0, 200]$. For the pure-random and Waxman networks, both link components are uniformly distributed in $(0, 20]$. The performance of MCP algorithms is measured by the success ratio, which is defined as follows

$$SR = \frac{\text{number of routing requests being routed}}{\text{number of total routing requests}} \quad (5)$$

We also record the running time for each of the algorithms under consideration. As the baseline, we take the running time of Dijkstra's as one, then the running time of other algorithms is measured by their multiples of the baseline: Dijkstra's shortest path algorithm. The results for ANSNET, pure-random and Waxman networks are presented in Tables 1, 2 and 3 respectively. The source and destination

Table 1. Success ratio (SR) and running time, ANSNET

KKT (Korkmaz's alg.) JSP (Jaffe's alg.), CN (Chen's alg.)		Exact	EDFS		JSP	KKT	CN	
			#1	#2			x = 3	x = 10
$c_1 \sim \text{unif}[50, 65]$	SR	0.2486	0.2480	0.2486	0.2422	0.2480	0.1890	0.2274
$c_2 \sim \text{unif}[200, 260]$	time	37.39	0.2893 (1)	0.8262 (3)	1	5.855	11.91	98.37
$c_1 \sim \text{unif}[75, 90]$	SR	0.5078	0.4966	0.5060	0.4896	0.5052	0.3294	0.4522
$c_2 \sim \text{unif}[300, 360]$	time	37.76	1.177 (2)	2.318 (4)	1	5.31	11.95	99.01
$c_1 \sim \text{unif}[100, 115]$	SR	0.7522	0.7352	0.7471	0.7224	0.7470	0.4460	0.6664
$c_2 \sim \text{unif}[400, 460]$	time	38.34	2.649 (3)	4.379 (5)	1	4.08	12.11	101.3
$c_1 \sim \text{unif}[125, 140]$	SR	0.9438	0.9048	0.9256	0.9214	0.9388	0.5422	0.8580
$c_2 \sim \text{unif}[500, 560]$	time	39.35	3.588 (3)	5.882 (5)	1	2.263	12.22	102.3
$c_1 \sim \text{unif}[150, 165]$	SR	0.9880	0.9602	0.9720	0.9714	0.9848	0.6142	0.9232
$c_2 \sim \text{unif}[600, 660]$	time	38.18	4.116 (3)	6.699 (5)	1	1.804	12.37	103.6

Table 2. Success ratio (SR) and running time, Pure-random

$p_r = 0.11$ (39 nodes and 75 links)		Exact	EDFS		JSP	KKT	CN	
			#1	#2			x = 3	x = 10
$c_1, c_2 \sim \text{unif}[10, 20]$	SR	0.0985	0.0980	0.0985	0.0975	0.0985	0.0780	0.0920
	time	37.08	0.1674 (1)	0.4892 (3)	1	4.882	11.52	94.71
$c_1, c_2 \sim \text{unif}[20, 30]$	SR	0.3810	0.3720	0.3805	0.3655	0.3755	0.2405	0.3355
	time	36.61	0.5851 (1)	1.669 (3)	1	5.913	11.61	95.48
$c_1, c_2 \sim \text{unif}[30, 40]$	SR	0.6715	0.6105	0.6640	0.6385	0.6635	0.3925	0.5880
	time	42.12	1.091 (1)	3.114 (3)	1	4.665	11.69	96.47
$c_1, c_2 \sim \text{unif}[40, 50]$	SR	0.9325	0.9100	0.9195	0.9075	0.9245	0.5925	0.8765
	time	35.98	4.749 (3)	7.813 (5)	1	2.372	11.83	98.37
$c_1, c_2 \sim \text{unif}[50, 60]$	SR	0.9880	0.9670	0.9770	0.9770	0.9845	0.7145	0.9650
	time	37.56	5.43 (3)	8.925 (5)	1	1.722	11.97	99.58

nodes are randomly chosen in all simulation configurations, and all the results presented here are averaged over 5000 randomly generated routing requests for different ranges of the routing constraints. For different ranges of constraints, the first row gives the success ratio (SR) and the second row gives the corresponding running time. For EDFS, the number in parenthesis indicates the number of executions of EDFS (with different exploring sequences).

As we can see, CN generally performs well only when the x is large enough, configurations with small values of x lag far behind all other algorithms in all simulation scenarios. Extremely high computational complexity is the main drawback of CN and makes it infeasible in practice. Note that, in our implementation, we only choose w_2 to be scaled down, and it turns out that the time complexity of CN is already far more expensive than the other algorithms. According to Chen and Nahrstedt [1], another run of the algorithm must be performed in which another weight component w_1 is scaled down instead, if we cannot find a feasible path when w_2 is scaled. The performance of CN can catch up by specifying larger x or invoking another execution with w_1 being scaled down, as shown by the work done by Chen and Nahrstedt [1] and other researchers [3], at much higher cost of execution time. KKT seems to be the best when the constraints are very loose (when more than 90% routing requests are actually routable), which is mainly due to the self adaptation of the coefficient k in the linear aggregation function. Korkmaz et al [3] show that the worse case complexity of

their algorithm is bounded by $\log(B(E + N \log(N)))$, where $\{B = N \cdot \max(w_i(u, v)), i = 1 \text{ or } 2, l_{u,v} \in E\}$. However, in our experiments, we notice that the running time of the algorithm becomes unpredictable when the basic approximation proposed by Korkmaz et al cannot find a feasible path to a destination. The reasons for this are the following. When the constraints are tight, multiple calls to the Dijkstra's shortest path algorithm must be made, until the proper coefficient k is found. The tighter the constraints are, the more calls we need to make. If no feasible path can be found by the basic KKT, two heuristic extensions will be invoked, for which the running time is no long bounded by logarithmic times calls to Dijkstra's shortest path algorithm. This explains why KKT generally takes a long time to find a feasible path when the routing constraints are tight.

Our algorithm EDFS outperforms all the other algorithms when the routing constraints are tight and moderate w.r.t. running time, success ratio SR or both. As noted before, KKT performs better when the routing constraints are very loose. One unique property of EDFS is that the tighter the constraints are, the better its performance becomes. Particularly, EDFS takes even less time than Dijkstra's shortest path algorithm to achieve nearly optimal success ratio (compared with the exact algorithm) when the constraints are very tight. We attribute this to the three extensions we made to the basic EDFS algorithm, especially *crankbacking*. Although KKT performs better when the constraints are very loose, its running time can become unpredictable when the basic approximation does not work. More importantly,

Table 3. Success ratio (SR) and running time, Waxman

$\alpha = 0.45, \beta = 0.25$ (40 nodes and 95 links)	Exact	EDFS		JSP	KKT	CN	
		#1	#2			x = 3	x = 10
$c_1, c_2 \sim \text{unif}[10, 20], SR$	0.1130	0.1125	0.1130	0.1120	0.1130	0.9350	0.1080
	time	73.72	0.2134 (1)	0.6156 (3)	1	5.711	93.6
$c_1, c_2 \sim \text{unif}[20, 30], SR$	0.4665	0.4455	0.4625	0.4410	0.4615	0.3130	0.4050
	time	74.63	0.8363 (1)	2.424 (3)	1	5.885	94.49
$c_1, c_2 \sim \text{unif}[30, 40], SR$	0.8650	0.7930	0.8520	0.8065	0.8405	0.5415	0.7870
	time	75.19	1.739 (1)	4.979 (3)	1	2.968	96.53
$c_1, c_2 \sim \text{unif}[40, 50], SR$	0.9920	0.9800	0.9865	0.9755	0.9870	0.7130	0.9675
	time	75.2	6.909 (3)	11.37 (5)	1	1.721	98.21

KKT can deal only with a 2 – constrained MCP, while EDFS can solve the general k – constrained MCP problem. We also note that, in all simulation configurations, it takes at least 10 times (up to 30 times) longer than EDFS for KKT SR to be comparable with that of EDFS when the constraints are very tight, while EDFS lags no more than 1.3% of SR behind KKT by using 7 times longer time than KKT does at most, when the constraints are very loose. Surprisingly, Dijkstra’s SP algorithm has good performance at the lowest cost (except when the constraints are very tight, EDFS takes less time than Dijkstra’s SP), even w.r.t. a simple linear link cost function $w(u, v) = \sum \frac{w_i(u, v)}{c_i}$, with which the gap of SR between Dijkstra’s SP and exact algorithm is no more than 6% in all configurations.

6.2. Performance with multiple constraints

In this section, we investigate the performance of EDFS subject to three or more constraints. Two new metrics are used to measure the performance of EDFS, which were first introduced by Yuan [9]. The first metric is the existence percentage EP , which is defined as

$$EP = \frac{\text{number of requests routed by exact algorithm}}{\text{number of total routing requests}} \quad (6)$$

EP actually equals to the success ratio of exact algorithm, and indicates how likely a feasible path can be found to meet the given request. Small EP means that it is difficult to find a feasible path for the given constraints. Secondly, competitive ratio CR is defined as

$$CR = \frac{\text{number of requests routed by heuristic algorithm}}{\text{number of requests routed by exact algorithm}} \quad (7)$$

CR indicates how well a heuristic algorithm can work against the exact algorithm with the same EP . In what follows, we use ANSNET as the simulation topology yet with three weight components for each link, and each of them is uniformly distributed in $(0, 20]$. Different constraints are chosen such that the existence percentage varies from 0.06 to 0.98, then the corresponding competitive ratios of EDFS with one, three and five executions are measured and plotted respectively in Figure 5. For each point in Figure 5, we use routing requests (source and destination are randomly

Figure 5. CR of EDFS with 3 constraints

chosen) with the same constraints over 2000 randomly generated ANSNET topologies (link weights are different for each topology). As we can see, EDFS again performs well when the constraints are tight. One execution of EDFS almost has 100% CR when EP is less than 25%, and three runs of EDFS is enough to achieve no less than 95% competitive ratio when the constraints are very loose.

To study the impact of the number of constraints on the performance of EDFS, we also conduct two sets of simulations on ANSNET, with the number of constraints varying from two to eight. In the first set of simulations, constraints are chosen such that the existence percentages are low, which are between 0.21 and 0.33. While in the second set, the existence percentages are high, which are between 0.71 and 0.82. The results are shown in Figure 6 and Figure 7 respectively. Again, every point is the average of requests using the same constraints over 2000 different ANSNET topologies. As we can see from Figure 6 and Figure 7, when the EP s are low (tight constraints), one execution of EDFS can achieve no less than 98% competitive ratios for all numbers of constraints, while three executions of EDFS already can have almost 100% competitive ratios. When the

Figure 6. Low EP $\sim [0.21, 0.33]$, ANSNET

EPs are high (loose constraints), three executions of EDFS are enough to have about 98% competitive ratios when the number of constraints varies from 2 to 8. Another advantage of EDFS is that its performance is insensitive to the number of constraints. Both the CRs (competitive ratio) with high EPs and CRs with low EPs do not vary much with different number of constraints. This is superior to heuristics using link weight scaling or limited granularity heuristic approach, whose performance may drop drastically as the number of constraints increases, as shown by Yuan [9].

7. Conclusion

We present EDFS, where the key idea is to maintain the descendant table $dTable$ to deduce possible dominating or incomparable paths to a finished node and its descendants, without exploring away from the finished node again. The running time of EDFS is polynomially bounded by $O(m^2 \cdot EN + N^2)$. We showed through extensive simulations that EDFS outperforms other popular MCP algorithms when the routing constraints are tight or moderate, and is comparable with them when the constraints are loose, and its performance is insensitive to the number of constraints. Another attractive aspect of EDFS is that the tighter the constraints are, the better the performance it can achieve, w.r.t. both time complexity and routing success ratio (running time is even less than that of Dijkstra's algorithm when the constraints are very tight).

References

- [1] S. Chen and K. Nahrstedt. On Finding Multi-constrained Paths. In *Proceedings of ICC'98*, pages 874–879, June, 1998.

Figure 7. High EP $\sim [0.71, 0.82]$, ANSNET

- [2] J. M. Jaffe. Algorithms for Finding Paths with Multiple Constraints. *IEEE Networks*, 14:95–116, 1984.
- [3] T. Korkmaz, M. Krunz, and S. Tragoudas. An Efficient Algorithm for Finding a Path Subject to Two Additive Constraints. In *Proceedings of the ACM SIGMETRICS*, pages 318–327, 2000.
- [4] F. Kuipers and P. V. Mieghem. The Impact of Correlated Link Weights on Qos Routing. In *Proceedings of IEEE INFOCOM*, 2003.
- [5] Q. Ma and P. Steenkiste. Quality-of-Service Routing for Traffic with Performance Guarantees. In *Proceedings of the IFIP Fifth International Workshop on Quality of Service*, pages 115–126, May, 1997.
- [6] P. V. Mieghem and F. A. Kuipers. Concepts of Exact Qos Routing Algorithms. *IEEE/ACM Trans. Netw.*, 12(5):851–864, 2004.
- [7] H. D. Neve and P. V. Mieghem. TAMCRA: A Tunable Accuracy Multiple Constraints Routing Algorithm. *Computer Communications*, 23:667–679, 2000.
- [8] Z. Wang and J. Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1228–1234, 1996.
- [9] X. Yuan. Heuristic Algorithms for Multiconstrained Quality-of-Service Routing. *IEEE/ACM Trans. Netw.*, 10(2):244–256, 2002.