A*Prune: An Algorithm for Finding K Shortest Paths Subject to Multiple Constraints

Gang Liu, K. G. Ramakrishnan

Abstract—**We present a new algorithm, A*Prune, to list (in order of increasing length) the first K Multiple-Constrained-Shortest-Path (KMCSP) between a given pair of nodes in a digraph in which each arc is associated with multiple Quality-of-Service (QoS) metrics. The algorithm constructs paths starting at the source and going towards the destination. But, at each iteration, the algorithm gets rid of all paths that are guaranteed to violate the constraints, thereby keeping only those partial paths that have the potential to be turned into feasible paths, from which the optimal paths are drawn. The choice of which path to be extended first and which path can be pruned depend upon a projected path cost function, which is obtained by adding the cost already incurred to get to an intermediate node to an admissible cost to go the remaining distance to the destination. The Dijkstra's shortest path algorithm is a good choice to give a good admissible cost. Experimental results show that A*Prune is comparable to the current best known ϵ-approximate algorithms for most of randomly generated graphs. BA*Prune, which combines the A*Prune with any known polynomial time ϵ-approximate algorithms to give either optimal or ϵ-approximate solutions to the KMCSP problem, is also presented.**

Keywords—**shortest paths, constraint based routing, QoS routing, multiple constrained path selection, Dijkstra algorithm, NP complete.**

I. INTRODUCTION

QUALITY of Service (QoS) sensitive routing is of critical importance in achieving a data network with high speed and efficiency (high ratio of capacity/cost), as well as meeting QoS for each connection. While efficient utilization of network resources is important to the service provider, the user is interested in both the speed and the quality of service provided. The need for QoS routing can be justified for both reservation-based services (e.g., Intserv, ATM) as well as reservationless services (e.g., Diffserv). The goal of QoS routing is to find a path that satisfies multiple QoS constraints while achieving overall network resource efficiency [1]. A key challenge in QoS routing is the K-Multiple-Constrained-Shortest-Path (KMCSP) problem, which is to find K feasible Constrained-Shortest-Path (CSP) from a source node to a target node and subject to multiple constraints (e.g., bandwidth, delay, jitter, administrative weight), and list them in order of increasing length. One special case of KMCSP problem is the Multiple-Constrained-Shortest-Path (MCSP), which is the KMCSP problem with $K = 1$.

The MCSP problem is a basic optimization problem that is both theoretically interesting and have many practical applications [2], and hence has received considerable attention in the literature [3] - [8].

Besides all the applications that are applicable to the MCSP problem, the KMCSP problem can also have many other practical applications, such as in the network routing and network design problems.

G. Liu is with the Aerie Networks, 1400 Glenarm Place, Denver, CO 80202. E-mail: GLiu@aerienetworks.com.

K. G. Ramakrishnan is with the Winphoria Networks, 2 Highwood Drive, Tewksbury, MA 01876. E-mail: ram@winphoria.com.

This work was done in 1999 when both authors worked at Bell Labs, Lucent Technologies, 600-700 Mountain Avenue Murray Hill, NJ 07974-0636.

In general, network routing consists of two basic tasks [5]: distributing the network information and searching for best feasible path with respect to given constraints. We focus on the second task and assume that the network information is available to every node (e.g., via link-state routing). Each link in the network is associated with multiple QoS metrics, which can be either static or dynamic (based on whether varying with time), either additive or non-additive (based on the additivity along a path) [7]. For the additive parameters (e.g., delay, hops, jitter), the cost of an end-to-end path is given, exactly or approximately, by the sum of the individual link values along that path. In contrast, the cost of a path with respect to a non-additive parameter, such as bandwidth, is determined by the value of that constraint at the bottleneck link. In the case of dynamic routing, the bandwidth is a dynamic parameter since its value may change each time a demand is routed or torn down, while the length of a link is a static parameter since it remains unchanged. In this paper we will mainly focus on additive and static parameters. The constraints associated with non-additive or dynamic parameters, can be dealt with first solving the KMCSP problem subject to all the additive and static constraints, and then select one path from these K CSPs such that all the other non-additive or dynamic parameters are satisfied. Most of the current approaches handle the non-additive or dynamic constraints by first pruning out all links that do not satisfy these constraints and then solving the MCSP problem in the residual network. The MCSP procedure need to be called each time a new demand need to be routed, and it may take time to find a feasible route if the MCSP procedure is time consuming, while our KMCSP method can speed up the online routing time, since selecting a feasible path from the K precomputed candidate paths is generally much faster than solving a MCSP problem in a residual network.

Generally, a network design problem is to find a least cost or a maximum revenue network, such that a given set of demands are routed through routes which meet some given QoS constraints. The KMCSP algorithms can be used to pre-compute a set of the candidate paths, then the network design problem can be formulated as a path based Linear Programming (LP) or Integer Programming (IP) problem, or can be solved by selecting paths from the precomputed path list through some other heuristic methods. We have applied the KMCSP algorithm in SPIDER, a web based optical network design tool [9].

In the case of single metric, the KMCSP problem becomes a problem of finding K-Shortest-Paths (KSP) from one source node to one target node. The KSP problem has many practical applications [10][11]. In multiple constrained metrics case, both the MCSP and the KMCSP problems are known to be NP-complete [3][4][12][14].

Most of the current approaches are concentrated on developing efficient polynomial or pseudo-polynomial-time algorithms

to give feasible or approximate solutions to MCSP problem. Existing algorithms can be grouped into three approaches:
1. ϵ-approximate solutions given by pseudo-polynomial-time algorithms in which the complexity depends on the actual values of the link weights (e.g., maximum link weight) in addition to the size of the network [4][14].
2. Polynomial-time heuristics that are fast in searching for a feasible path but cannot guarantee finding one in the network [2][3][5].
3. ϵ-approximate algorithms that are polynomial [12][13].

The main contribution of this paper is developing an algorithm to give exact solutions to the KMCSP problem, and can also be applied to its special cases such as MCSP and KSP problems. The algorithms presented in this paper can be applied to multiple constraints and can find any required number of CSPs. To give an exact solution, the running time may be exponential in the worst case. However, our test cases show that the actual running time of our algorithm is comparable to the existing approximate algorithms for most of practical networks.

The rest of the paper is organized as follows. The KMCSP problem and its solutions are further analyzed in Section II. A*Prune, an algorithm for KMCSP problem, is presented in Section III. Section IV analyzes the performance of the A*Prune algorithm. Section V discusses some efficient methods for computing the lower-bounds, which are used to give the look-ahead feature in A*Prune, thus variant of A*Prune corresponding different lower bounds for the KMCSP, such as A*Uniform, A*Dijkstra, are presented. In Section VI, we combine the A*Prune with any known ϵ-approximate algorithm to give the Bounded A*Prune (BA*Prune), which can give either exact or ϵ-approximate solutions to KMCSP problem in polynomial-time. In Section VII we provide some experimental results and the comparisons for the algorithms presented in this paper and some well-known algorithms. Conclusions are presented in Section VIII. All the proofs to the lemmas are given in the Appendix.

II. Problem Formulation

The KMCSP problem can be defined as follows.

Definition 1: KMCSP problem: Consider a network that is represented by a 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 R non-negative and additive QoS values: $w_r(i, j), r = 1, 2, \ldots, R$. A length (sometime called cost) function w_0 is defined as follows:

$$w_0(i,j) = \sum_{r=1}^{R} a_r w_r(i,j) \tag{1}$$

Given a source node s and a target node t, and R constraints $C_r(s,t), r = 1, 2, \ldots, R$. The KMCSP problem is to find either the first K shortest length paths or all the paths (depending on which number is smaller) from a source node s to a target node t such that

$$W_r(p(s,t)) \stackrel{\text{def}}{=} \sum_{(i,j)\in p(s,t)} w_r(i,j) \le C_r(s,t) \quad \forall r \in (1,2,\ldots,R). \tag{2}$$

We use a bold character to represent an $(R+1)$-dimensional vector in this paper, such as $\mathbf{X} \in E^{R+1}$ represents $\mathbf{X} = (X_0, X_1, \ldots, X_R)$. Generally, the constraints vector $\mathbf{C}(i,j)$ depends on the node i and node j, however, we sometimes use $\mathbf{C}$ to represent $\mathbf{C}(s,t)$.

In the case of $K = 1$, the KMCSP, or simply the MCSP problem can be formulated as an integer programming (IP) problem as follows.

Definition 2: MCSP (KMCSP with $K = 1$) as an (edge based) IP problem :

$$\begin{array}{l} given: \\ G(V,E), s \in V, t \in V, R \ge 1, K \ge 1; \\ w_r(i,j) \ge 0, \quad \forall (i,j) \in E, \quad \forall r \in (1,2,\ldots,R); \\ a_r \ge 0, \quad C_r \ge 0, \quad \forall r \in (1,2,\ldots,R); \\ define: \\ w_0(i,j) = \sum_{r=1}^{R} a_r w_r(i,j); \\ W_r = \sum_{(i,j)\in E} w_r(i,j)x(i,j), \forall r \in (0,1,\ldots,R); \\ minimize: W_0; \\ subject \quad to: \\ x(i,j) \in \{0,1\}, \quad \forall (i,j) \in E; \\ \sum_{(i,j)\in E} w_r(i,j)x(i,j) \le C_r, \quad \forall r \in (1,2,\ldots,R); \\ \sum_{j:|(i,j)\in E} x(i,j) - \sum_{j:|(j,i)\in E} x(j,i) = \begin{cases} 1, & if \quad i = s \\ -1, & if \quad i = d \\ 0, & else \end{cases} \end{array} \tag{3}$$

Let $d(p)$ be the destination node of the path p in this paper. To cope with the KMCSP problem, we give notations to some sets of paths as follows.

$$\begin{array}{l} P(V,V) = \{x : |x \text{ is a path in G}\}; \\ P(i,V) = \{x : |x \in P(V,V) \text{ and i is its source node}\}; \\ P(i,j) = \{x : |x \in P(i,V) \text{ and j is its target node}\}; \\ P(i,V,\mathbf{f}(x),\mathbf{C}(i,t)) = \\ \quad \{x : |x \in P(i,V), \mathbf{f}(x) \le \mathbf{C}(i,t)\}; \\ P(i,j,\mathbf{f}(x),\mathbf{C}(i,j)) = \\ \quad \{x : |x \in P(i,V,\mathbf{f}(x),\mathbf{C}(i,j)) \text{ and } d(x) = j\}; \\ P(i,j,\mathbf{f}(x),\mathbf{C}(i,j),K) = \{x : |x \in \text{the first K shortest} \\ \quad \text{length paths of } P(i,j,\mathbf{f}(x),\mathbf{C}(i,j))\}; \end{array} \tag{4}$$

Here, $C_0 = \infty$, $i \in V$, $j \in V$, and $\mathbf{f}(p) \in E^{R+1}$ represents (R+1) given functions of the path p. The constraint C_0 is redundant and thus can be set to infinity.

Using these symbols, the solution sets of different problems can be represented as follows.

$$\begin{array}{rl} KMCSP: & P(s,t,\mathbf{W}(p),\mathbf{C},K) \\ MCSP: & P(s,t,\mathbf{W}(p),\mathbf{C},K)|_{K=1} \\ KSP: & P(s,t,\mathbf{W}(p),\mathbf{C},K)|_{R=1,C_1=\infty} \end{array} \tag{5}$$

Definition 3: Simple path: A simple path is a path without loops.

Definition 4: Head path and tail path: Let node u be an intermediate node of the path $p(i,j)$. Node u divides path $p(i,j)$ into two paths, path $p(i,u)$ and path $p(u,j)$. then path $p(i,u)$ is called a head path of path $p(i,j)$, path $p(u,j)$ is called a tail path of path $p(i,j)$. We represent the path $p(i,j)$ as the combination of a head path and a tail path in the following format:

$$p(i,j) = p(i,u)p(u,j) \tag{6}$$

Definition 5: Additive Parameters: Let $W_r(p(i,j))$ be a parameter associated with the path $p(i,j)$, and $W_r(p(i,j)) = w_r(i,j)$ when $p(i,j)$ is only one link path, then w_r is called an additive parameter if

$$W_r(p(i,u)p(u,j)) = W_r(p(i,u)) + W_r(p(u,j)) \quad (7)$$

remains true for all $i \in V$, $j \in V$, $u \in V$.

Definition 6: lower-bound distance: Let $\mathbf{W}(p(i,j))$ be a vector associated with the path $p(i,j)$ and as defined in (2), then a vector $\mathbf{D}(i,j,\mathbf{C}(i,j))$ is called a lower-bound distance vector from node i to node j and associated with the constraint vector $\mathbf{C}(i,j)$ if

$$D_r(i,j,\mathbf{C}(i,j)) = \min_{\substack{p(i,j)\in P(ij,\mathbf{W}(p),\mathbf{C}(i,j)) \\ \forall r \in (0,1,\ldots,R)}} W_r(p(i,j)) \quad (8)$$

Definition 7: Admissible distance: A vector $\mathbf{A}(i,j,\mathbf{C}(i,j))$ is called an admissible distance from node i to node j and associated with the constraint vector $\mathbf{C}(i,j)$ if

$$\mathbf{0} \le \mathbf{A}(i,j,\mathbf{C}(i,j)) \le \mathbf{D}(i,j,\mathbf{C}(i,j)) \quad (9)$$

Here, $\mathbf{D}$ is a lower-bound distance and the operator $\le$ with two vector operands is defined as:

$$\mathbf{a} \le \mathbf{b} \Longleftrightarrow a_r \le b_r, \qquad \forall r \in (0,1,\ldots,R) \quad (10)$$

Definition 8: Projected distance: Given a path $p(s,i) \in P(s,V)$, a constraint vector $\mathbf{C}(s,t)$ and an admissible distance $\mathbf{A}(i,j,\mathbf{C}(i,j))$, a path function $\mathbf{H}(p(s,i))$ is called a projected distance associated with the node pair (s,t) and the constraint $\mathbf{C}(s,t)$ if

$$\mathbf{H}(p(s,i)) = \mathbf{W}(p(s,i)) + \mathbf{A}(i,t,\mathbf{C}(\mathbf{s},\mathbf{t}) - \mathbf{W}(p(s,i))) \quad (11)$$

Definition 9: Feasible path: A path $p(s,t)$ is called a feasible path associated to the node pair (s,t) and the constraint vector $\mathbf{C}(s,t)$ if $p(s,t) \in P(s,t,\mathbf{W}(p),\mathbf{C}(\mathbf{s},\mathbf{t}))$ Within this meaning, $P(s,t,\mathbf{W}(p),\mathbf{C})$ can also be called the feasible path set.

Definition 10: admissible head path: A path $p(s,i)$ is called an admissible head path associated to the node pair (s,t) and the constraint vector $\mathbf{C}(s,t)$ if $p(s,i) \in P(s,V,\mathbf{H}(p(s,i)),\mathbf{C})$. Here $\mathbf{H}(p(s,i))$ is a projected distance. Within this meaning, $P(i,V,\mathbf{H}(p(s,i)),\mathbf{C})$ can also be called the admissible head path set.

Based on the definitions given above, we make the following observations.

Lemma 1: $P(s,t,\mathbf{H}(p),\mathbf{C},K)$ is the solution set of KMCSP problem, i.e.,

$$P(s,t,\mathbf{H}(p),\mathbf{C},K) = P(s,t,\mathbf{W}(p),\mathbf{C},K) \quad (12)$$

Lemma 2: $\forall p \in P(s,V)$, path p can always be expanded from the trivial path $p(s,s)$.

Lemma 3: A path expanded from an inadmissible head path must be an inadmissible head path, thus can never be a solution path, i.e.,

$$q \notin P(s,V,\mathbf{H}(p),\mathbf{C}) \Longrightarrow qy \notin P(s,V,\mathbf{H}(p),\mathbf{C}) \quad (13)$$
$$\forall q \in P(V,V) \quad and \quad \forall y \in P(V,V)$$

These Lemmas tell us that we can get the solution set of KMCSP by expanding the trivial path $p(s,s)$ and all its extended feasible head paths step by step. This gives the basic ideas of the A*Prune Algorithm.

III. A*Prune Algorithm

A*-search[15][16][17][18], as well as uniform-search, breadth-first-search and depth-first-search are well known searching strategies in Artificial Intelligence [19]. We combine the A*-search with a proper pruning technique to get the A*Prune algorithm, which can be used to solve the KMCSP problem. Using the terminologies described in the previous sections, the A*Prune algorithm can be simply described as: starting from expanding the path $p(s,s)$, potentially, all the paths in $P(s,V)$ can be reached; however, with a proper pruning against the given constraints $\mathbf{C}$, only the paths in admissible head path set $P(s,V,\mathbf{H}(p),\mathbf{C})$ remain as candidate paths for further expanding; furthermore, the candidate paths are ordered properly, such that the path with shortest projected length $H_0(p)$ is selected and expanded first, then we can terminate our expansion procedure once we have found enough number of CSPs or there are no candidate paths left. This procedure will only expand a subset of the admissible head path set $P(s,V,\mathbf{H}(p),\mathbf{C})$.

A pseudo-code of A*Prune algorithm is shown in Figure 1.

The key processes in A*Prune are explained and analyzed as follows.

1. *Pre-compute an admissible distance,* $\mathbf{A}(i,t,\mathbf{C}(i,t))$, $\forall i \in V$: Here $\mathbf{C}(i,t)$ represents some known constraints associated to the node pair (i,t). We can set $\mathbf{C}(i,t) = \infty$ if nothing is known about $\mathbf{C}(i,t)$. Many existing algorithms, such as Dijkstra's shortest path algorithm can be used in finding a good admissible distance. We will look at the question of designing good admissible distances in Section V.

2. *Path expanding:* Suppose we have an admissible head path list, AHP_heap, which is initialized to contain the trivial path $p(s,s)$. The path expanding process first selects and removes a path p from AHP_heap, then expands the selected path one step further to get all possible extended paths and inserts all the admissible head paths into the AHP_heap.

3. *CSP collecting:* Each time we take a path from AHP_heap, we first check if the path is a CSP. If it is, the path is saved as a solution and will not be expanded further.

4. *Candidate path list ordering:* If the candidate path list AHP_heap is ordered in a way such that the path with shortest projected length $H_0(p)$, and the largest length $W_0(p)$ if breaking the tie, is put on the head of AHP_heap, then we can stop the expanding process once we have found enough number of CSPs without any loss of the optimality. Once the candidate list is ordered in this way, the A*Prune search will be best (with shortest $H_0(p)$) first search, and depth (largest $W_0(p)$) first search if several paths has the same value of $H_0(p)$. The heap-sort[20] algorithm is used here for its efficiency. The candidate path list AHP_heap is heap-sorted whenever a path is added or removed.

5. *Inadmissible head path pruning:* Once a new path is generated in the path expanding process, a check is made against the given constraints, using some lookahead features. The newly generated path will be put into the candidate path list AHP_heap if it belongs to the feasible head path set $P(s,V,\mathbf{H}(p),\mathbf{C})$. All the inadmissible head paths are pruned out and will not be expanded further. So, if the path expanding process is combined with the constraint pruning process, all the paths in AHP_heap are admissible head paths,

function $A^*prune(G, s, t, \mathbf{w}(E), \mathbf{C}, K, R)$
inputs:

$G = (V, E)$, a graph with node set V and edge set E;
(s, t): a node pair with source s and target t;
K: number of paths to be found;
R: number of constraints, $R = 1$ if applied to KSP;
$\mathbf{w}(e)$: R metrics associated to each link $e \in E$;
$\mathbf{C} = \mathbf{C}(s, t)$: R constraints;

1. $\forall i \in V, i \neq s, i \neq t$ and $\forall r \in (1, 2, \ldots, R)$, compute:
   $D_r(i, t)$; // length of Dijkstra path from i to t
   $D_r(s, i)$; // length of Dijkstra path from s to i
   $C_r(i, t) \leftarrow C_r(s, t) - D_r(s, i)$;
   $A_r(i, t, \mathbf{C}(i, t)) \leftarrow D_r(i, t)$;//Addmissible distance
2. Initialize:
   $k \leftarrow 0$; // number of CSPs found
   $\mathbf{W}(p(s, s)) \leftarrow \mathbf{0}$;
   $\mathbf{H}(p(s, s)) \leftarrow \mathbf{A}(s, t, \mathbf{C}(s, t))$;
   $AHP\_heap \leftarrow \{p(s, s)\}$; // admissible head path heap
   $CSP\_list \leftarrow \{\}$; // set of found CSPs
3. **while** ($k \leq K$ and AHP_heap is not empty)
4. $q(s, u) \leftarrow$ the first path of AHP_heap;
5. Remove the first path of AHP_heap;
6. $heapsort(AHP\_heap)$;
7. $u \leftarrow$ the end node of $q(s, u)$;
8. **if** $u = t$ **then**
9. insert $q(s, u)$ into CSP_list;
10. $k \leftarrow k + 1$;
11. **goto** 3;
12. **end if** // $q(s, u)$ is saved as a CSP solution
13. $OutEdges \leftarrow$ {all edges outgoing node u};
14. **while** ($OutEdges$ not empty)
15. $e(u, v) \leftarrow$ a removed edge from $OutEdges$;
16. $p(s, v) \leftarrow q(s, u)e(u, v)$;
17. $\mathbf{W}(p(s, v)) \leftarrow \mathbf{W}(q(s, u)) + \mathbf{w}(e(u, v))$;
18. **if** simple path is required and $v \in q(s, u)$
19. **then goto** 14;
20. **end if** //non-simple head paths are pruned
21. **for** $r = 1, 2, \ldots, R$
22. **if** $H_r(p(s, v)) > C_r$ **then goto** 14;
23. **end if**
24. **end for** // inadmissible head paths are pruned
25. Insert $p(s, v)$ into AHP_heap;
26. $heapsort(AHP\_heap)$;
27. **end while** // $p(s, u)$ has been expanded
28. **end while**
29. **return** CSP_list;
30. **stop**

function $H_r(p(s, v))$
returns $W_r(p(s, v)) + A_r(v, t, \mathbf{C}(v, t))$;
function $heapsort(AHP\_heap)$ **returns** a heap AHP_heap such that the path $p(s, i)$ with minimum $H_0(p(s, i))$, and maximum $W_0(p(s, i))$ in case of breaking the tie, is put on the head of the heap AHP_heap.

Fig. 1. Algorithm of A*Prune

i.e., $AHP\_heap \subset P(s, V, \mathbf{H}(p), \mathbf{C})$. Furthermore, all the paths in $P(s, V, \mathbf{H}(p), \mathbf{C})$, will be eventually expanded once the AHP_heap is exhausted to empty.

6. *non-simple path pruning (required for simple path searching only):* This step is required only if simple path is an additional requirement to the CSPs. Any non-simple path does not need to be put on the candidate path list, since any of its extended paths must also be non-simple. Suppose all the paths in the candidate path list are simple paths, then a newly generated path extended from any of the candidate paths is simple if its end node presents only once in the path. So we can prune out all the non-simple paths by checking the appearance of the end node in a newly generated path.

7. *Terminating condition:* The A*Prune program will be terminated either it has found the required number of CSPs or there are no paths left in the candidate path list. The candidate path ordering steps let the candidate path with shortest projected path length, and the longest length path if breaking the tie, to be expanded first. So the ordering steps may result in finding the required number of CSPs as early as possible before exhausting all the paths in AHP_heap. The pruning steps try to keep the AHP_heap to contain candidate paths as less as possible.

The A*Prune algorithm combines all these processes to select, expand, prune the candidate path list step by step, until the required number of CSPs are found or there are no candidate paths left.

IV. The Performance of A*Prune

The performance of an algorithm is usually evaluated in terms of the following four criteria [19]:

- *Completeness:* is the algorithm guaranteed to find a solution when there is one?
- *Optimality:* does the algorithm find the highest-quality solution when there are several different solutions?
- *Time complexity:* how long does it take to find the solution?
- *Space complexity:* how much memory does it need?

*Lemma 4: The completeness of A*Prune:* A*Prune is complete in finding simple KMCSP paths. However, A*Prune may not be complete in finding complex KMCSP paths.

*Lemma 5: The optimality of A*Prune:* A*Prune is optimal in finding either simple or complex KMCSP paths.

Let Q be the number of expanded paths, then, the space complexity of A*Prune is Q, and the time complexity of A*Prune is $dQ(R + h + \log Q)$, Where K is the number of paths to be found, h is the maximum hops of these K shortest paths, d is the degree of G and R is the number of the constrained metrics. However, A*Prune can still be exponential, since Q may be exponential. It has been proved that A* can be sub-exponential growth if the error in the heuristic function grows no faster than the logarithm of the actual path cost[21][22]. Because of the pruning, our A*Prune is more efficient than A*. Thus, the above conclusion for A* also remains true for A*Prune. In mathematical notation, the condition for sub-exponential growth is that

$$|\mathbf{D}(i, t, \mathbf{C}(i, t)) - \mathbf{A}(i, t, \mathbf{C}(i, t))| \leq O(\log |\mathbf{D}(i, t, \mathbf{C}(i, t))|)$$

Therefore, it is always better to use an admissible distance $\mathbf{A}(i, t, \mathbf{C}(i, t))$ with higher values, as long as it does not overestimate the lower-bound distance $\mathbf{D}(i, t, \mathbf{C}(i, t))$.

When applied to KSP problem, A*Prune becomes polynomial growth algorithm and the length of candidate path list is bounded by:

$$Q \leq Khd \leq KN^2 \tag{14}$$

Then the space complexity of A*Prune for KSP is Khd and the time complexity is $O(Khd^2(h + R + \log(Khd)))$.

V. Methods for Computing Admissible Distance

So far we have seen the two extreme ends of admissible distances: $\mathbf{A}(i,t,\mathbf{C}) = \mathbf{0}$ and $\mathbf{D}(i,t,\mathbf{C}(i,t))$ given in (8). The former one does not need any computation but does little help to the look ahead search, while the second one is really helpful to the look ahead search but it also increases the precomputation time. We need some admissible distances between the two extreme ends, such that they are close to the best admissible distance and can be computed efficiently.

A problem with less restrictions than the original problem is called a **relaxed problem**. It can be easily shown that a solution of any relaxed problem of the MCSP problem, as shown in (3), always gives an admissible distance $\mathbf{A}(s,t,\mathbf{C}(s,t))$. In this section, we present some efficient methods for computing admissible distances by solving a relaxed problem of (3).

A. Uniform Distance

Obviously, $\mathbf{A}(i,t,\mathbf{C}(i,t)) = \mathbf{0}, \forall i \in V$, is an admissible distance, which is called **uniform distance**. A*Prune with uniform distance is called **A*Uniform** algorithm.

A*Uniform, like A*Prune, expands all paths from source node. However, instead of using heuristic functions as look-ahead feature, A*Uniform checks the actual path criteria for pruning and the shortest length path is expanded first.

A*Uniform seems too greedy and probably is not as efficient as A*Prune with greater admissible distance. However, in the case of finding CSPs from one source node to N target nodes and with uniform constraints ($\mathbf{C}(i,t) = \mathbf{C}(s,t), \quad \forall i \in V$), A*Uniform maybe more efficient than running N rounds of A*Prune.

B. Dijkstra Distance

For each metric w_r, we can use Dijkstra's algorithm to find the path with shortest $W_r(p)$. We can find R shortest metrics associated with each $r \in (1,2,\ldots,R)$. These R shortest metrics make up a vector, which is called the Dijkstra distance and is represented by the symbol $\mathbf{D}(i,j,\infty)$. By definition and using (8), we have:

$$D_r(i,j,\infty) = \min_{p(i,j)\in P(i,j)} W_r(p(i,j)) \tag{15}$$

$$\forall r \in (0,1,\ldots,R)$$

Lemma 6: The Dijkstra distance is admissible.

Therefore, A*Prune can use Dijkstra distance as an admissible distance and is called A*Dijkstra algorithm in that case. The A*Dijkstra algorithm remaines unchanged as shown in figure 1 except the procedure of updating the admissible distance at lines **??** and **??** can be omitted.

In the case of simple path is also a requirement to KMCSP problem, we can compute the Dijkstra distance in network $G' = G - s$ (the network G with node s and all the edges linked to node s removed), since we really do not want any of the look-ahead paths passing through node s again. Usually, the Dijkstra distance in network G' can give a better admissible distance than the Dijkstra distance in network G.

C. Admissible Distance Given by Other Known Heuristic Algorithms

There are many known algorithms which can give lower bounds to the MCSP problem [12]. Given an ϵ-approximate solutions [14], a strict lower bound can also be retrieved by subtracting the error from the approximate solution. Lagrangian relaxation method [23] can also be used to generate a lower-bound to the IP problem listed in (3). Obviously, any of these algorithms can be used for computing the admissible distance in our A*Prune.

VI. BA*Prune

Despite all the nice properties of the A*Prune algorithm, such as optimality, completeness, ability to solve KMCSP, A*Prune is not polynomial time. Both the time and space complexities of A*Prune remain exponential. Compared with some existing approximate algorithms [12][14], the uncertainty of time and space complexities is the main drawback of A*Prune algorithm. In this section, we combine the A*Prune with any known ϵ-approximate algorithms to give a Bounded A*Prune algorithm, BA*Prune, which can give either exact or ϵ-approximate solutions to KMCSP problem in polynomial-time. However, the BA*Prune can only be applied to $R \leq 2$ cases due to the applicability of known ϵ-approximation algorithms.

Let ϵ-KMCSP be a known ϵ-approximate algorithm to KMCSP problem, BA*Prune will first run A*Prune for a bounded time T and bounded memory S (say the length of the candidate list Q), for example we can select $S = N^3$ and $T = N^3\mu$sec. If A*Prune is successful to give the exact solution within the bounded time and memory, we are done. Otherwise, BA*Prune executes the ϵ-KMCSP to find an ϵ-approximate solution. Since we can select S and T as polynomials of N, then the BA*Prune is polynomial efficient algorithm if the ϵ-KMCSP is polynomial algorithm. However the trade off for getting this efficiency is that BA*Prune is partially optimal rather than optimal: it is optimal if it is run in the bounded time, otherwise is ϵ-approximate. The completeness of BA*Prune is guaranteed by the completeness of ϵ-KMCSP. All the other advantages of A*Prune are remained in BA*Prune, such as find K CSP paths and multiple constraints, as long as BA*Prune can be run in the bounded time and memory.

VII. Comparison of Algorithms and Experimental Results

In this section we compare the following algorithms in terms of CPU times:

- **A*Dijkstra:** The A*Prune algorithm with Dijkstra distance, i.e., the A*Dijkstra algorithm presented in this paper. It gives K exact CSPs from one source node s to one target node t.

TABLE I

ALGORITHM RUNNING TIME COMPARISON ($p = 0.4$, TIME UNIT: MICROSECOND)

Input Data			1-1 target		1-N targets	
N	D	K	A*Dijks	L-Scale	A*Unifo	T-Scale
100	249	1	5357	8443	160734	71020
		10	32773			
		100	336367			
200	241	1	20714	# 18420	1084268	1286781
		10	54749			
		100	365116			
300	238	1	38160	11705	2994072	1619317
		10	186266			
		100	1673267			
400	236	1	114843	# 85745	4940668	1197150
		10	372441			
		100	3291033			
500	234	1	98889	# 111651	9548881	2880185
		10	205030			
		100	3670254			

- **L-Scale:** The length scaling algorithm presented by R. Hassin [14]. It gives one CSP from one source node s to one target node t with length in the range of ϵ-approximation.
- **A*Uniform:** The A*Prune algorithm with uniform distance, i.e., the A*Uniform algorithm presented in this paper. It gives K exact CSPs from one source node s to each of the N target nodes.
- **T-Scale:** The delay scaling algorithm presented in [12]. It gives one CSP from one source node s to each of the N target nodes with delay in the range of ϵ-approximation.

We measured their execution times on a 450MHz SUN Solaris machine. For simplicity, we only tested for the case of two constraints, $w_0 = w_1$ is length and the w_2 is delay. The error tolerance parameter ϵ was set to 5% for the two ϵ-approximate algorithms, L-Scale and T-Scale.

For our experiments we used random graphs of a given node size N and connectivity p which is defined as the probability of a link existence between node pair (i, j). The link length, link delay and delay-limit D are also random numbers uniformly distributed in the open interval (0, 1000). The running times of the four algorithms for some randomly generated graphs are listed in Table I. To see how the running time varies with K, the A*Dijkstra algorithm is tested for K =1, 10 and 100 cases, while the other three algorithms are only tested for $K = 1$ case. The running time prefixed with a # indicates the non-optimal solution.

These experimental results show that the A*Dijkstra is comparable to L-Scale and the A*Uniform is comparable to T-Scale in running time. Considering A*Dijkstra and A*Uniform can find K exact CSP solutions, and can be applied to multiple constraints and all the metrics can be float numbers, while the L-Scale and T-Scale can only give one ϵ-approximate CSP solution and can only be applied to two constraints with integer metrics, we can say that our A*Dijkstra and A*Uniform have better performance on the average than the best known ϵ-approximate algorithms, L-Scale and T-Scale.

VIII. CONCLUSIONS

A*Prune, an algorithm for finding K shortest paths subject to multiple constraints, has been presented. The algorithm grows a candidate path list, which contains the paths starting from the source node s and is initialized to the trivial path of (s, s). The candidate path list is ordered such that the path that most likely to project to a shortest feasible path is extended first. All the extended paths that are guaranteed to violate the constraints are pruned from the candidate list. Dijkstra's shortest path length can be used as lookahead feature in both the candidate path pruning and ordering processes. Experimental results show that A*Prune is comparable to the current best known polynomial-time ϵ-approximate algorithms. BA*Prune, an algorithm combines the A*Prune with any other known ϵ-approximate algorithm to give either optimal or ϵ-approximate solution to the KMCSP problem in polynomial time, is also presented.

APPENDIX

We give proofs to the lemmas given in this paper.

Lemma 1. *Proof:* Since p is a path from node s to node t, then

$$\mathbf{D}(t, t, \mathbf{C}(t, t)) = \mathbf{0}$$

$$\mathbf{H}(p) = \mathbf{W}(p)$$

Hence (12) is true. ■

Lemma 2. *Proof:* Suppose the edges on p is listed in the increasing order of number of hops from the start node s as $\{e_1, e_2, \ldots, e_h\}$, where h is the number of edges on path p, then

$$p = p(s, s)e_1e_2 \ldots e_h$$

Since path $p(s, s)e_1$ is one step expanded path from path $p(s, s)$, $p(s, s)e_1e_2$ is two steps expanded path from path $p(s, s)$, etc., the path p is h steps expanded path from path $p(s, s)$. ■

Lemma 3. *Proof:* Assume path q is an inadmissible head path, i.e., $\exists r \in (1, 2, \ldots, R), H_r(q) > C_r$. Let x be an path expanded from q, i.e., $\exists y \in P(V, V), x = qy$. Then $\exists r \in (1, 2, \ldots, R)$, such that $H_r(x) = W_r(qy) + A_r(d(y)) = W_r(q) + (W_r(y) + A_r(d(y))) \geq W_r(q) + A_r(d(q))) > C_r$. This proves that path x must be an inadmissible head path. Lemma 3 is true considering that x is any extended path of the inadmissible head path q. ■

Lemma 4. *Proof:* As we said that because A*Prune expands paths in the admissible head path set $P(s, V, \mathbf{H}(p), \mathbf{C})$ in the increasing order of $H_0(p)$, and all feasible paths also belong to the admissible head path set, it must eventually expand to reach a feasible path. This is true unless there are infinitely many admissible head paths q with $H_0(q \in P(s, V, \mathbf{H}(p), \mathbf{C})) \leq H_0(p \in P(s, t, \mathbf{H}(p), \mathbf{C}))$. In the simple path search case, the admissible head path set must be limited, since its super set, the set of all simple paths in a graph is limited. However, in the complex path search case, loops with infinitely small length may result in A*Prune searching infinitely many admissible head paths before finding a feasible solution. ■

Lemma 5. *Proof:* Given two paths u and v, both are feasible solutions of the KMCSP problem, and $W_0(u) < W_0(v)$. Here $W_0(p)$ is defined as the length of path p. A search

strategy for KMCSP problems is optimal if it finds u earlier than v.

Assume A*Prune finds v earlier than u. The optimality of A*Prune can be proved by proving this assumption can not be true. We first decompose u into three parts:

$$u = aeb$$

where $a \in P(s, V, \mathbf{H}(p), \mathbf{C})$, $b \in P(d(e), V)$, $e \in E$. Then at the moment that A*Prune finds v, there exist paths a and b and an edge e, such that a is expanded but ae is not. Since u is a feasible path, its head path ae must be an admissible head path. Then ae is put into the candidate path list instead of being pruned when its parent path a is expanded. Since v is expanded but ae is not, then we must have:

$$H_0(v) \leq H_0(ae)$$

$$d(v) = t \Longrightarrow H_0(v) = W_0(v)$$

Hence:

$$\begin{array}{rcl} W_0(v) & \leq & H_0(ae) \\ & \leq & W_0(ae) + D_0(d(ae), t, \mathbf{C}) \\ & \leq & W_0(ae) + W_0(b) \\ & \leq & W_0(u); \end{array} \tag{16}$$

This contradicts to the given condition. Then our assumption, A*Prune finds v earlier than u, can not be true. This proves the optimality of A*Prune. ■

Lemma 6. *Proof:* It is obvious that

$$P(i, j, \mathbf{W}(p), \mathbf{C}(i, j)) \subset P(i, j)$$

then

$$\min_{p(i,j) \in P(ij)} W_r(p(i,j)) \leq \min_{p(i,j) \in P(ij, \mathbf{W}(p), \mathbf{C}(ij))} W_r(p(i,j))$$

$$\forall r \in (0, 1, \ldots, R)$$

or equivalently, $\mathbf{D}(i, j, \infty) \leq \mathbf{D}(i, j, \mathbf{C}(i, j))$ must be remained true. This proves that the Dijkstra distance is admissible. ■

REFERENCES

[1] D. Mitra and K. G. Ramakrishnan. A Case Study of Multiservice, Multipriority Traffic Engineering Design for Data Networks. *Proceedings of the GLOBECOM'99 Conference,* General Conference (Part B), pp 1077-1083. IEEE, 1999.

[2] Dean H. Lorenz and Ariel Orda. QoS routing on networks with uncertain parameters. *IEEE/ACM Transactions on Networking,* 6(6):768-778, December 1998.

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

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

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

[6] W. C. Lee, M. G. Hluchyi, and P. A. Humble. Routing subject to quality of service constraints in integrated communication networks. *IEEE Network,* pp. 46-45, July/August 1995.

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

[8] A. Warburton. Approximation of pareto optima in multiple-objective shortest path problems. *Operations Research,* 35:70-79, 1987.

[9] E. Bouillet, G. Liu and I. Saniee. Algorithms for WEDM Mesh Network Design: Routing and Wavelength Assignment for Dedicated Protection, Ring Auto-Recovery and Optical Cross-Connect Restoration in Core Optical Networks. *Technical Report,* 10009626-000126-01TM, Bell Laboratories/Lucent Technologies, 2000.

[10] D. Eppstein. Finding the *k* shortest Paths. *Proc. 35th IEEE Symp. FOCS,* 154-165. 1994.

[11] D. Eppstein. Finding the *k* shortest Paths. *SIAM J. Computing,* 28(2):652-673. 1999.

[12] A. Goel, D. Kataria, D. Logothetis and K. G. Ramakrishnan. An Efficient Algorithm for Constraint-based Routing in Data Networks. *Technical Report,* BL0112120-990616-11TM, Bell Laboratories/Lucent Technologies, 1999.

[13] D. Raz and D. H. Lorenz. Simple Efficient Approximation Scheme for the Restricted Shortest Path Problem. *Technical Report,* 10009674-991214-04TM, Bell Laboratories/Lucent Technologies, 1999.

[14] Refael Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research,* 17(1):36-42, February 1992.

[15] S. Lin. Computer solutions of the traveling salesman problem. *Bell Systems Technical Journal,* 44(10):2245-2269, 1965.

[16] P. E. Hart, N.J. Nilsson and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics,* SSC-2(2):100-107, 1968.

[17] P. E. Hart, N.J. Nilsson and B. Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". *SIART Newsletter,* 37:28-29, 1972.

[18] A. Newell and G. Ernst, The search for generality. *Information Processing 1965: Proceedings of IFIP Congress,* volume 1, pages 17-24. Spartan. 1965.

[19] Stuart Russell and Peter Norvig. Artificial Intelligence, A Modern Approach. *Prentice-Hall, Inc.,* pp. 70-118, NJ, 1996.

[20] A. V. Aho and J. D. Ullman. *Foundations of Computer Science.* C Edition, pp. 280-285, W. H. Freeman and Company, New York, 1995.

[21] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence 5,* pages 219-236. Elsevier/North-Holland, Amsterdam, London, New York, 1970.

[22] I. Pohl. Practical and theoretical considerations in heuristic search algorithms. *Machine Intelligence 8,* pages 55-72. Ellis Horwood, Chichester, England. 1977.

[23] S. Ahuja and J. Orlin. Network Flows. *Addison-Wesley Publishing Company,* MA, 1998.