Fast Exact MultiConstraint Shortest Path Algorithms

Yuxi Li Janelle Harms Robert Holte
Department of Computing Science, University of Alberta, Edmonton, AB, Canada T6G 2E8
{yuxi, harms, holte}@cs.ualberta.ca

Abstract— **QoS routing has been shown to be NP-hard. A recent study of its hardness shows that the "worst-case" may not occur in practice [13]. This suggests that there may exist fast exact algorithms for the multi-constraint shortest path (MCSP) problem, an instance of QoS routing.**

Search techniques such as A* and IDA* may solve hard problems exactly in polynomial time. In [14], we deploy the idea of iterative deepening search to design IDA*_MCSP, and show its efficiency by extensive empirical study. In this paper, we show that for infeasible cases, IDA*_MCSP may not be as efficient as A*Prune. This motivates us to design an algorithm that is efficient in both feasible and infeasible cases.

We design an exact MCSP algorithm A*_MCSP, which introduces the state notion and dominance relationship between states. Furthermore, we design an exact MCSP algorithm FringeMCSP. It can be regarded as an integration of IDA*_MCSP and A*_MCSP. Extensive empirical study shows that FringeMCSP has good performance in both feasible and infeasible cases; while IDA*_MCSP still shows its superiority among the proposed MCSP algorithms in feasible cases.[1]

I. Introduction

The multi-constraint path (MCP) problem is to find one or several feasible paths subject to multiple constraints on a given network topology with link weights, such as delay, loss, administrative cost, etc. For instance, we may want to find the shortest path in a DiffServ network or an overlay network with the delay and the loss ratio below certain level. The multi-constraint shortest path (MCSP) problem is to find the shortest path with respect to hop count that satisfies the constraints. The MCP and the MCSP problems are instances of QoS routing. Their NP-hardness property [22] led to the proposals of many heuristic algorithms or approximate algorithms. See [6] and [12] for overviews. On the other hand, Cheeseman et al. found that typical cases of many NP-hard problems are tractable in practice [5]. Kuipers et al. studied that the hardness of QoS routing occurs in topologies with special characteristics, and the problem may be easy to solve in realistic communication networks [13]. This suggests that there may exist fast exact algorithms for MCP and MCSP problems. Recently, there are research efforts on designing exact algorithms for them. See [12] for an overview.

QoS routing has two components: QoS information collection and QoS routing computation. We assume QoS information collection has been done by a mechanism such as the OSPF extension in [3] or by a measurement approach such as that in [2]. We concentrate on QoS routing computation. A link may be a physical link in a network, or a virtual link in a Virtual Private Network or an overlay network. The information gathered may be related to delay, loss rate or bandwidth. Link weights are defined by these metrics. Link weights can be additive, multiplicative or concave. By additive weight, we mean the weight of a path is the sum of the weights of the links on the path. For a multiplicative weight, the weight of a path is the product of the weights of the links on the path. Multiplicative weights such as loss rate can be transformed to additive weights by taking the logarithm of the path weight. A concave weight of a path is the minimum of the link weights of the path. Concave weights such as bandwidth can be dealt with by using a preprocessing procedure to remove ineligible links. Without loss of generality, we concentrate on additive weights.

The constraints may be specified by the Service Level Agreements (SLAs) or customers' requirements. Given the link weights and constraints, each node can compute the shortest path to a destination subject to multiple constraints. In a DiffServ architecture, paths may be computed centrally at the Bandwidth Broker [18]. In an overlay network [2], paths are computed at overlay nodes. Once a path is computed, source routing or MPLS may be used to forward packets on that route.

Search techniques from the artificial intelligence community such as A* [9] and IDA* [10] have shown their strength in solving some hard problems in practice [20]. A search algorithm usually uses look-ahead information to speed up the search. When the look-ahead information always underestimates the solution cost, the search algorithm is guaranteed to find the optimal solution. With the assistance of a good look-ahead function to predict solution length, A* and IDA* may solve hard problems exactly in polynomial time [20], [11].

A*Prune [15] and SAMCRA [17] are two exact QoS routing algorithms. A*Prune is designed for the MCSP problem and SAMCRA is designed for the MCP problem, although it can also be used for MCSP. They borrow ideas from A* and conduct best first searches using a priority queue. They use look-ahead information to cut off the part of search space that won't lead to a feasible solution. In [14], we design a fast exact MCSP algorithm, IDA*_MCSP, and show its superiority over A*Prune in feasible cases. We also show the high accuracy of look-ahead information in [14], which determines the efficiency of a search-based MCSP algorithm.

Previous work in QoS routing focuses on designing efficient algorithms when a problem is feasible, i.e., there is at least a solution. However, an MCSP may be infeasible, due to one or more constraints. This can happen in the service negotiation phase. Thus, it is desirable to design an efficient

[1]The paper contains several large figures. It may takes some time to download and to print. Thank you for your patience.

algorithm for both feasible cases and infeasible cases, which can either find the feasible path fast or can determine quickly that there is not a feasible path. In this paper, we first design A*_MCSP, which introduces the state notion to replace partial paths in A*Prune and SAMCRA. Furthermore, we design FringeMCSP, which can be regarded as an integration of IDA*_MCSP and A*_MCSP. Extensive performance study shows that FringeMCSP performs well in both feasible and infeasible cases; while IDA*_MCSP still shows its superiority among the proposed MCSP algorithms in feasible cases.

Our work of designing exact MCSP algorithms is different from the work in QoS routing that proposes a heuristic and looks for an approximate solution. Our algorithms IDA*_MCSP, A*_MCSP and FringeMCSP are guaranteed to find the exact solution once it terminates (if it is feasible). Their optimality makes them different from the polynomial time approximation algorithms.

We give the notation used in the paper. A network is represented by a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each edge $(u, v) \in E$ is associated with m non-negative additive weights, $w_k(u, v),\ k = 1, 2, ..., m$. The m weights on edge (u, v) form an m-dimensional weight vector $W(u, v) = \{w_1(u, v), w_2(u, v), ..., w_m(u, v)\}$. We denote the source node as src and the destination node as dst. We have an m-dimensional constraint vector $C = \{c_1, c_2, ..., c_m\}$, corresponding to the m weights. The i-th weight of a path P is $w_i(P) = \sum_{(u,v)\in P} w_i(u, v), \forall 1 \leq i \leq m$. The weight vector of a path P is $W(P) = \{w_1(P), w_2(P), ..., w_m(P)\}$. When we say *shortest path*, we mean the shortest path with respect to hop count, if not explicitly stated. The path length is the number of hops of a path. The definitions of the MCP and MCSP problems follow.

Definition 1: MCP problem: Given a network represented by a graph G, m associated non-negative additive weights on edges, the source node src, the destination node dst, and the m-dimensional constraint vector C, the MCP problem is to find a path P such that $w_i(P) \leq c_i, \forall 1 \leq i \leq m$.

Definition 2: MCSP problem: Given a network represented by a graph G, m associated non-negative additive weights on edges, the source node src, the destination node dst, and the m-dimensional constraint vector C, the MCSP problem is to find the shortest path P such that $w_i(P) \leq c_i, \forall 1 \leq i \leq m$.

In Section II, we present an overview of search techniques. In Section III, we present an approach to compute look-ahead information [14]. We overview IDA*_MCSP in Section IV. We investigate the performance of IDA*_MCSP for infeasible cases in Section V. We present A*_MCSP in Section VI and FringeMCSP in Section VII. We also present the comparison results. Then we draw conclusions.

II. Overview of Search Techniques

A. Look-ahead

A search algorithm usually deploys some look-ahead function to predict the quality of a potential solution. A good look-ahead function plays an important role in enhancing the performance of a search algorithm, by facilitating decision making of whether to further search a branch. We use the Dijkstra's algorithm to calculate the shortest paths for *hop* or for each of the weight metrics to obtain look-ahead information.

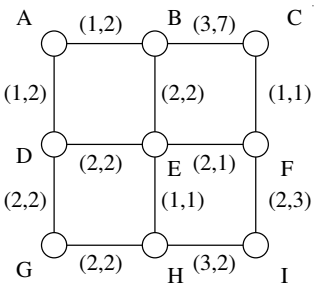


Fig. 1. Example Graph

We give an example for illustration. In Figure 1, we are to find the shortest path from A to C subject to the constraint $\{7, 8\}$. The vectors on the edges are edge weight vectors, such as $W(A, B) = (1, 2)$. Denote these two edge weights w_1, w_2, and the two constraints, c_1, c_2. Denote the look-ahead information for node u and metric w as $lb(w, u)$, where w can be the hop count or one of the two edge weights w_1 and w_2. It is easy to compute that for node A, $lb(hop, A) = 2$, $lb(w_1, A) = 4$, and $lb(w_2, A) = 6$; while for node E, $lb(hop, E) = 2$, $lb(w_1, E) = 3$, and $lb(w_2, E) = 2$, etc.

B. A* Search

A* algorithm is a well-known search strategy in artificial intelligence. It is a best-first search algorithm. For each node in the search tree, A* maintains the *distance traversed* so far and the *estimate distance*, which is the sum of the distance traversed and the distance of look-ahead.

A* maintains an $openlist$ and a $closedlist$. It takes the first node from the $openlist$ and expands it to the neighbors. Newly expanded nodes are put into $openlist$, sorted in increasing order by the estimate distance. Ties are broken with traversed distance, giving preference to larger values, since a larger value means more reliable information. The state just removed from $openlist$ is put in $closedlist$. A* terminates if the goal node is found or there is no node on the $openlist$. A* always finds the shortest path given that the heuristic does not over-evaluate the distance from a node to the destination. That is, admissibility of heuristics guarantees the optimality of the solution [20]. However, it requires exponential space since it stores all the temporary search results.

Liu and Ramakrishnan [15] extend A* to A*Prune for the problem of finding K shortest paths subject to multiple constraints. It maintains partial paths on a list, in increasing order of the estimates of distances to the destination, breaking ties with the lengths of partial paths. The first path on the list is expanded to the neighbors of the tail node. The resulting partial paths that pass the eligibility tests against all the constraints are put into the list for further expansion. The algorithm terminates once it finds K shortest paths or there are no more partial paths on the list.

We will introduce a new exact MCSP algorithm based on A*, A*_MCSP, which introduces the state notion and dominance relationship between states. It eliminates the cost to maintain partial paths as in A*Prune.

C. Iterative Deepening Search

An iterative deepening search algorithm [10] conducts a series of depth-first searches. Different from depth-first search, it has a depth bound for each iteration. That is, when the search algorithm has traveled as far as the depth bound, or it predicts that there wouldn't be a solution within the search bound, it stops searching from that node. An iterative deepening search algorithm updates the depth bound after each iteration, until the solution is found, or it determines that there is no feasible solution. Bounding the search depth avoids searching too deeply. Moreover, by updating the search bound properly, the algorithm guarantees to find the shortest path. The cost to pay is that part of the search space has to be searched redundantly. We design IDA*_MCSP in [14], which will be described in Section IV.

D. Fringe Search

A* has to keep a sorted openlist of all the frontier nodes in the search tree, which may be costly. IDA* has to make iterations with updated search threshold, unless the source has perfect lookahead information of the destination. Many nodes will be searched redundantly. Björnsson et al. [4] recently introduce Fringe search to attempt to overcome shortcomings of A* and IDA*, and show its efficiency on pathfinding on game maps. The basic idea follows. The algorithm keeps $nowlist$ and $laterlist$, and iterates on a depth bound. In each iteration, the algorithm searches on $nowlist$, and put nodes to be considered later on $laterlist$. The head of $nowlist$ is removed from the list and examined. If its estimate distance is greater than the depth threshold, then the head element is put into $laterlist$. Otherwise, add the children of the head to the front of $nowlist$. When $nowlist$ is empty, update the depth threshold, copy $laterlist$ to $nowlist$, and clear $laterlist$. Search on $nowlist$ iteratively, until the goal is found or it is determined that the goal is not reachable. Fringe search can be regarded as an integration of A* and IDA*. It attempts to take advantage of both A* and IDA*; at the same time, it attempts to avoid their shortcomings. We extend Fringe search to MCSP in Section VII.

III. Look-ahead

We discuss the approach we use to compute look-ahead information. The input to a search-based MCSP algorithm is the graph representation of the network topology, the weights on each link, and the constraints.

A 2-dimensional array $lb[m+1][n]$ is used to store the lower bounds, i.e. the look-ahead values, for the $m+1$ metrics $\{hop, w_1, w_2, ..., w_m\}$ (hop count and m weights) for each of the n nodes. We use Algorithm 1 to set the lower bounds at the initialization stage using the $lookahead()$ function, which calls the Dijkstra's algorithm to calculate the shortest paths.

Algorithm 1 $computeLowerbound()$

```
1: for each metric metric ∈ {hop, w_1, w_2, ..., w_m} do
2:     lb[metric][] ← lookahead(metric);
3: end for
```

Function $lookahead(metric)$ calculates the shortest paths for all the nodes to the destination, with respect to $metric$, which can be the hop count or one of the weights. The results are recorded in the array $lb[][]$. The destination has lookahead values of 0 for each metric.

Dijkstra's algorithm is used for calculating the respective shortest path due to its efficiency. More importantly, Dijkstra's algorithm makes underestimates, which is a necessary property for the look-ahead function in order to help a search-based MCSP algorithm find the optimal solution. By an underestimate, we mean that the least cost (length) path found by Dijkstra's algorithm won't exceed the cost (length) of the shortest constrained path. We assume the topology is symmetric, thus we can use Dijkstra's algorithm on the single-source shortest path problem to compute the look-ahead values of all the nodes to the destination. For an asymmetric topology, we may first compute the transpose of the graph [7], and then apply Dijkstra's algorithm. We assume link weights are static, so we need to calculate the lower bounds only once. If there are changes, an efficient, incremental algorithm can be designed based on the work in Ramalingam and Reps [19] to incrementally recalculate lowerbounds after changes.

We study the accuracy of look-ahead information on a wide range of topologies including inferred ISP topologies, Internet-like power-law topologies, and Waxman random topologies. The empirical study on the diverse topologies shows the high accuracy of look-ahead information using the Dijkstra's algorithm, which implies the potential efficiency of a search-based MCSP algorithm. Please see [14] for details.

A. Eligibility Test

A search-based MCSP algorithm can use look-ahead information to do eligibility tests to prune unnecessary portions of the search space. Algorithm 2 presents the function that conducts an eligibility test at node $node$ for the neighbor nbr according to the current accumulated weight vector $curW$. For each weight, w_i, it predicts the path weight from src to dst via $node$ and nbr by adding up the following three components: the i-th element of $curW$; the weight on the link from $node$ to nbr, $w_i(node, nbr)$; and the look-ahead value for the predicted path from nbr to the destination, lb_i. If any constraint is violated, the test fails.

IV. IDA*_MCSP

We overview our extension of IDA* to the MCSP problem IDA*_MCSP [14], on which FringeMCSP in Section VII is based. IDA*_MCSP returns the optimal path if there exists one. Otherwise, it reports a failure.

Algorithm 3 presents one iteration of the IDA*_MCSP algorithm. The algorithm arrives at the current node $node$, with

Algorithm 2 $eligible(node, nbr, curW)$

```
1: for each weight w_i ∈ {w_1, w_2, ..., w_m} do
2:    lb_i = lb[w_i][node];
3:    if curW_i + w_i(node, nbr) + lb_i > C_i then
4:       return false;
5:    end if
6: end for
7: return true;
```

Algorithm 3 IDA*_MCSP($node$, hop, $curW$, $depth\_bound$)

```
1: h ← lb[hop][node];
2: if h is 0 then
3:    return true;
4: end if
5: predict_depth ← h + hop;
6: if predict_depth > depth_bound then
7:    return false;
8: end if
9: for each neighbor nbr of node do
10:   if eligible(node, nbr, curW) is true then
11:      updateW(node, nbr, curW);
12:      done ← IDA*_MCSP(nbr, hop+1, curW, t);
13:      restoreW(node, nbr, curW);
14:      if done is true then
15:         return true;
16:      end if
17:   end if
18: end for
```

distance hop from src and accumulated path weight vector $curW$, and the search threshold $depth\_bound$. If the predicted path length is greater than $depth\_bound$, it stops searching this partial path further. Otherwise, it considers each of the $node$'s neighbors. An eligibility test is conducted first for the neighbor nbr to see whether there is a potential solution via nbr. If the eligibility test succeeds, the algorithm updates the hop count and the weight vector for the traversed partial path, and searches further from nbr. Otherwise, it considers the next available neighbor. Functions $updateW(node, nbr, curW)$ and $restoreW(node, nbr, curW)$ are called before and after making a recursive call to IDA*_MCSP() to properly record the current accumulated weight.

Algorithm 4 presents the main function to use the IDA*_MCSP algorithm. At the initialization stage, it calculates the lower bounds for the hop count and each link weight.

The threshold $depth\_bound$ is initialized as the look-ahead value for hop count from src to dst. If the algorithm fails to reach a solution within the $depth\_bound$, it increases $depth\_bound$ to search further. IDA*_MCSP updates $depth\_bound$ as the value of the least predicted distance ($predict\_length$ in Line 5 in Algorithm 3) in the last iteration to make a close estimate of the constrained path length.

The *search process* from Line 5 to Line 14 of Algorithm 4 finds the shortest constrained path. IDA*_MCSP() is called

Algorithm 4 main()

```
1: Global src, dst, C, maxLength
2: computeLowerbound();
3: depth_bound ← lb[hop][src];
4: done ← false;
5: while done is false do
6:    curW ← 0;
7:    done ← IDA*_MCSP(src, 0, curW, depth_bound);
8:    if done is false then
9:       update depth_bound;
10:      if reach stopping condition then
11:         report failure;
12:      end if
13:   end if
14: end while
```

iteratively with updated (increased) $depth\_bound$ after each iteration. It returns true when it finds the shortest path. The path can be constructed by backtracking the search stack. It reports failure when the stopping condition is reached.

A. An Example

Take the example in Figure 1 in Section II-A. Recall that we are to find the shortest path from A to C subject to the constraint $\{7, 8\}$. IDA*_MCSP starts with depth bound 2, since the shortest path from A to C has length 2. At iteration 1, the algorithm will reach nodes B and D. After reaching node B, it does not expand node C, since using the eligibility test, $c_2 = 8$ will be violated. Neither will node E be expanded, since with the look-ahead information on hop count, the length of the shortest path from E to C, is 2; and $1 + 1 + 2 = 4$ exceeds the depth bound 2 for this iteration. Node D won't be further expanded due to the reason that the depth bound will be violated.

An iteration with depth bound 3 won't find the solution either. In fact, with proper update, this iteration can be skipped to make the search more efficient. The depth bound can be set as 4 for the second iteration, since a potential solution via node E has at least $2 + 2 = 4$ hops, where the 2's are the distance traversed so far and the length of shortest path from E to C, the look-ahead information on hop count.

In the iteration with depth bound 4, again, after reaching B, C won't be expanded. After reaching E, it will expand F and reaches C. The shortest path $P = ABEFC$ will be found, with the weight vector $W(P) = \{6, 6\}$. Path $ADEFC$ may be found, if we expand neighbors in a different way.

V. Performance of IDA*_MCSP in Infeasible Cases

In [14], extensive empirical study shows that, in feasible cases, IDA*_MCSP is much more efficient than A*Prune. We now investigate the performance of IDA*_MCSP and A*Prune in infeasible cases.

It would be desirable to study network problems on realistic Internet topologies. However, ISP topologies are usually

regarded as proprietary information. Fortunately, the Rocketfuel project [21] deployed new techniques to measure ISP topologies and made them publicly available. OSPF/IS-IS weights on the links (inferred weight and latency) are also provided [16]. We also use synthetic topologies, including Internet-like topologies following power-laws [8], [1] and random graphs based on the Waxman model [23]. For power-law topologies, we use sizes of 3037, 5000, 7500 and 10000. For Waxman topologies, we use network sizes of 250, 500, 1000, 2500 and 5000. On these topologies, link weights are set uniformly in the range of $(0, 1)$. There are two link weights, i.e. $m = 2$. We use a *tightness factor*, α, to set the constraint vector for each search. For each source-destination pair, each constraint is set as the tightness factor α times the weight of the least weight path of the source-destination pair by Dijkstra's algorithm with respect to that weight. α is set as 1.1, 1.2 or 1.5. As a consequence, the constraints may be different for different source-destination pairs.

Both IDA*_MCSP and A*Prune do the identical preprocessing to calculate the lower bounds, thus we concentrate on the comparison of the computation time for the search process (for IDA*_MCSP, Line 5 to Line 14 of Algorithm 4). For the inferred ISP topologies, we study each possible source-destination pair. In each synthetic topology, we choose 10 destinations randomly, and 100 random sources for each destination. The experiments are conducted on a Linux machine with 602 MHz CPU and 256MB memory. For each source-destination pair, we run 1000 repetitions and take the average for precision. For performance study for feasible cases in later sections, in each synthetic topology, we choose 100 destinations randomly, and 100 random sources for each destination.

Figure 2 shows the results with scatter plots[2]. A point below the $y = x$ line means for that instance, IDA*_MCSP is faster than A*Prune. The closer to the x-axis, the faster IDA*_MCSP is, vice versa. For inferred ISP topologies, there are not many infeasible cases, esp. when the tightness factor α is large. On ISP topologies and random topologies, IDA*_MCSP is faster than or comparable with A*Prune. However, on power-law topologies, A*Prune can be much faster than IDA*_MCSP in infeasible cases.

A* maintains a priority list. IDA* conducts iterative deepening search. In infeasible cases, the look-ahead information is not accurate (there is no solution at all). Therefore an A*-based MCSP algorithm has the chance to narrow down the search space; while an IDA*-based MCSP algorithm has to do redundant search. This is the reason A*Prune may win IDA*_MCSP in some infeasible cases. It is interesting to design a MCSP algorithm that is efficient in both feasible and infeasible cases. An A*-based algorithm is worth exploiting.

VI. A*_MCSP

It is time-consuming to maintain partial paths in A*Prune. Thus, it is desirable not to maintain partial paths during the search process. There may be several paths reaching a node with the same quality. By path quality, we mean both hop number and accumulated weights. We call such partial paths *equivalent partial paths*. Take the example in Figure 1 in Section II-A. When the search reaches node E by paths ABE and ADE, we have two equivalent partial paths, with the same hop number and the same accumulated partial path weights. We observe that it is a waste if the search algorithm processes two different paths further, $ABEF$ and $ADEF$, since searching two different paths will not improve the path quality of searching only one of them.

We first introduce the concept of state and dominance, then we describe the A*_MCSP algorithm and study its performance. State and dominance are also important components in designing FringeMCSP in Section VII.

A. State and Dominance

The maintenance cost increases by maintaining partial paths formed from equivalent partial paths, without improving the quality of the searched paths. It suffices to remember the quality of paths that have reached a node. We use the $state$ notation to represent a state in the search space in our A* extension to MCSP, rather than $path$ as in A*Prune. In the following, when we mention a state, we mean the state in the state notation we use. A state is a tuple,

$$state = \{node, hop, estimate, W, parent\},$$

where $node$ is the node in question, hop records the number of hops from the source to this state, $estimate$ is the sum of hop and the look-ahead of $node$, W is the weight vector containing the weight values accumulated so far, and $parent$ is a backward pointer to the state that leads to the current state.

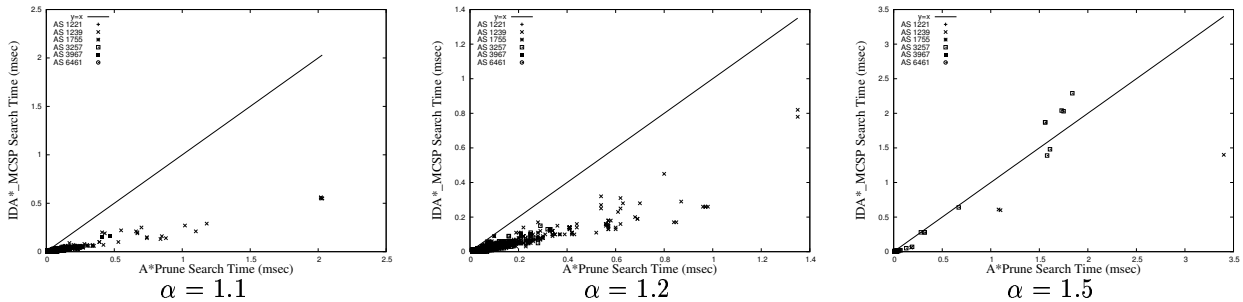
Using the state notation, both path ABE and path ADE can be encoded as a state, $S_E = \{E, 2, 4, \{3, 4\}, S_B\}$, with hop and $estimate$ as 2 and 4 respectively. Here we have the weight vector as $\{3, 4\}$. The $parent$ field is set as S_B, the state notation of node B. We will explain how we process the $parent$ field after introducing the $dominance$ relation.

We have made the search space compact by using state notation. We can check dominance relation of two states of the same node, p, to make further enhancements. $S_p^2 = \{p, hop_2, estimate_2, W_2, parent_2\}$ is dominated by $S_p^1 = \{p, hop_1, estimate_1, W_1, parent_1\}$, if

$$hop_1 \leq hop_2 \text{ and } W_1 \leq W_2,$$

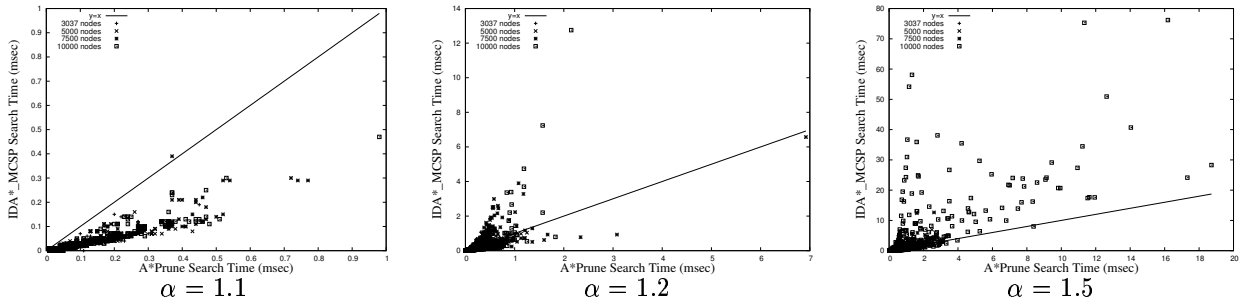
where $W_1 = \{w_{1,1}, w_{1,2}, ..., w_{1,m}\}$, and $W_2 = \{w_{2,1}, w_{2,2}, ..., w_{2,m}\}$. Thus $W_1 \leq W_2$ is defined by $w_{1,i} \leq w_{2,i}, \forall 1 \leq i \leq m$. Note that $hop_1 \leq hop_2$ implies $estimate_1 \leq estimate_2$, since S_p^1 and S_p^2 are of the same node p, thus they have the same look-ahead value for hop count. It is unnecessary to further search a dominated state, since it is impossible for it to lead to a better solution than the state that dominates it. Discarding dominated states can prevent cycles, since a state associated with a path containing a cycle is dominated by a state for the path removing the cycle. Thus a search algorithm with this feature does not need

[2]To save space, we do not show the results for random topologies, for pairwise comparisons, i.e., one algorithm versus another. The average search time over all instances for each topology can be found in Figure 7 and Figure 8.



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

(a) Rocketfuel Topologies



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

(b) Power-law Topologies

Fig. 2. Search Process Time (msec), Infeasible Case: A*Prune (x-axis) vs. IDA*_MCSP (y-axis)

to remove cycles explicitly. Note that a dominance relation is necessary for discarding a state. That is, we can not discard a state S which is not dominated by any other state, even if one or several of its constraints are much larger than that of the other states. In MCSP, such an S may lead to a better path (with respect to hop count) than the others.

Now we give more explanations of the $parent$ field of a state. There are two ways to set the $parent$ field, a single parent or multiple parents. We may store the parent that gives the state the best quality. The quality of a state is measured by both the hop number and the accumulated weights. A state of the best quality dominates all the other states (of the same node). This suffices to find a single optimal path. We may also store some or all the parents that have led to the current state, except those forming cycles. In this example, we may set $parent = \{S_B, S_D\}$. Storing multiple parents, the search algorithm can backtrack recursively to find multiple optimal paths (which may not be node-disjoint). In the following, we store only the best parent. Note, multiple partial paths leading to the same node will be encoded as multiple states, if they do not have dominance relationship with each other.

In the above example, we set $parent$ field as S_B, but not S_D, because a possible state $S_E^{'} = \{E, 2, 4, \{3, 4\}, S_D\}$ via path ADE is dominated by $S_E = \{E, 2, 4, \{3, 4\}, S_B\}$ via path ABE (in fact, they have the same state quality).

We also conduct eligibility tests on a possible state expansion to a neighbor, by checking whether any constraint will be violated using look-ahead values of weights. In Figure 1 in Section II-A, state $\{D, 1, 4, \{1, 2\}, S_A\}$ won't be expanded to state $\{G, 2, 6, \{3, 4\}, S_D\}$, since the weight metric w_1 evaluated for node D is $4 + 6 = 10$, which violates constraint $c_1 = 7$ (c_2 is also violated).

We observe that the number of states will not exceed the number of partial paths. Free of path operations is another benefit of the state approach, which maintains backward pointers to construct the entire path after it is found. However, it needs to check dominance relationship between states.

*B. A*_MCSP*

Algorithm 5 presents the function to check whether a state $s^{'}$ newly created for $node$ is dominated by an old state stored for $node$. Algorithm 6 presents the main algorithm of A*_MCSP, our extension of A* to MCSP. At the initialization stage, $computeLowerbound()$ (Algorithm 1) calculates the lower bounds for the hop count and each link weight. We use $eligible()$ (Algorithm 2) to make eligibility tests.

The search process from Line 3 to Line 18 of Algorithm 6 finds the shortest constrained path. A*_MCSP has an $openlist$ and a $closedlist$. New states are put into $openlist$ for further expansion. On the $openlist$, states are sorted in increasing order by the $estimate$ values, breaking ties with hop values, and giving preference to larger hop values. If a neighbor can not pass the eligibility tests, A*_MCSP will not further consider it. When constructing a new state for an eligible neighbor, A*_MCSP updates $estimate$ using the look-ahead information. It also updates the weight vector W (plus the constraints on the link) and hop (plus 1). Function

Algorithm 5 isDominated($s^{'}$)

```
1: node ← node field of s';
2: S ← all stored states of node;
3: for each s ∈ S do
4:    if s < s' then
5:       return true;
6:    end if
7: end for
8: return false;
```

Algorithm 6 A*_MCSP()

```
1: computeLowerbound();
2: put state {src, 0, lb[hop][src], 0⃗, null} into openlist;
3: while openlist not empty do
4:    s ← first state in openlist;
      /*take s away from openlist*/
5:    put s into closedlist;
6:    if s.node is dst then
7:       construct path;
8:       return true;
9:    end if
10:   for each neighbor nbr of s.node do
11:      if eligible(s.node, nbr, curW) then
12:         construct state s' for nbr;
13:         if isDominated(s') is false then
14:            put s' into openlist in order;
15:         end if
16:      end if
17:   end for
18: end while
19: return false;
```

$isDominated(s)$ checks whether state s is dominated by any state on $openlist$ or $closedlist$. We use an efficient data structure to implement the $openlist$ for A*_MCSP. We use a 2-dimensional bucket to store the states, with the assistance of an array and a variable to query proper state to expand.

C. Performance Study of A*_MCSP

Figure 3 and Figure 4 show that A*_MCSP outperforms A*Prune in both infeasible and feasible cases[3]. The main reason is A*_MCSP uses state maintenance, replacing partial path maintenance in A*Prune. In Figure 4 (b) with $\alpha = 1.5$, there are cases where A*Prune is slightly more efficient than A*_MCSP, which is due to that A*_MCSP has to do dominance check. However, A*_MCSP outperforms A*Prune in most cases. We can say A*_MCSP is more efficient.

Experimental results also show that in most cases IDA*_MCSP is much more efficient than A*_MCSP in feasible cases, although there are some cases where A*_MCSP wins IDA*_MCSP (to save space, we do not show the results). It is interesting to investigate whether there is still room to improve A*_MCSP in both feasible cases and infeasible cases. Fringe search is worth exploiting.

[3]For feasible cases, we do not show the results for AS1239 and AS3257 of Rocketfule topologies, to reduce the size of the file.

Algorithm 7 expand(s)

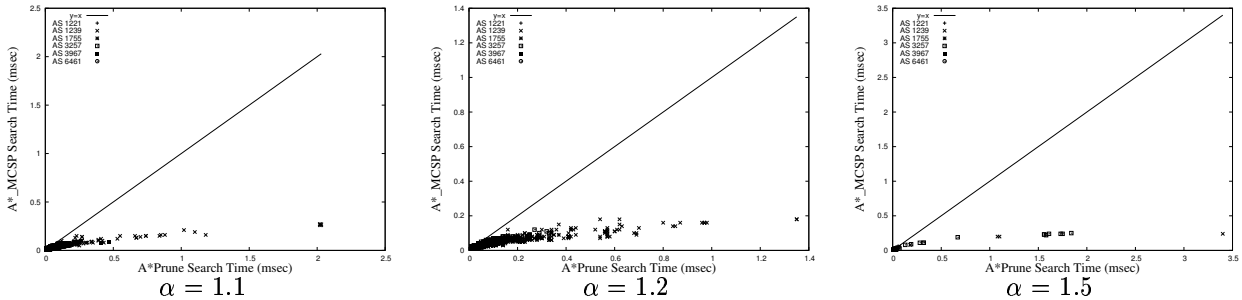
```
1: if s.f > depth_bound then
2:    insert s at the end of laterlist;
3:    return false;
4: end if
5: for each neighbor nbr of s.node do
6:    if eligible(s.node, nbr, curW) then
7:       if nbr is dst then
8:          construct path;
9:          return true;
10:      end if
11:      construct state s' for nbr;
12:      if isDominated(s') is false then
13:         if s'.f < depth_bound then
14:            put s' into nowlist;
15:         else
16:            put s' into laterlist;
17:         end if
18:      end if
19:   end if
20: end for
21: return false;
```

VII. FringeMCSP

Björnsson et al. [4] recently introduce Fringe search to attempt to overcome the shortcomings of A* and IDA*, namely, sorting $openlist$ and redundant search. We extend Fringe search to design an exact MCSP algorithm, FringeMCSP. It searches iteratively according to a threshold like IDA*_MCSP. It uses the state notation and dominance check like A*_MCSP. It uses Algorithm 2 $eligible()$ to make eligibility test. It uses Algorithm 5 $isDominated()$ to check dominance relationship of states of a node.

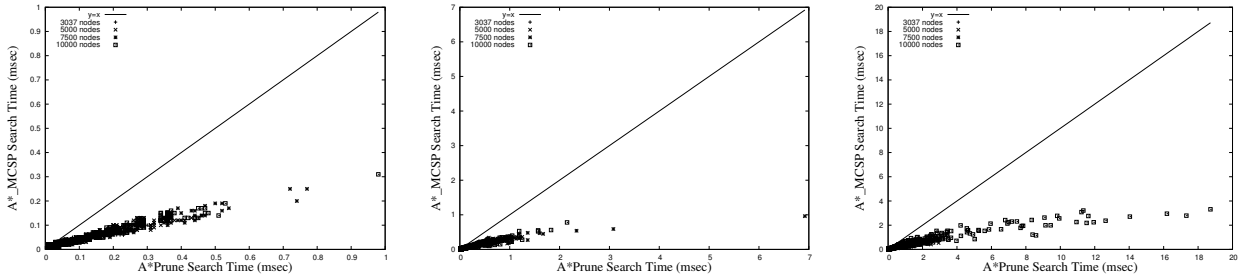
Algorithm 8 presents the FringeMCSP algorithm. It keeps $nowlist$ and $laterlist$ of states. It initializes the lowerbounds for hop and each of the weight metrics. In each iteration, the algorithm searches on $nowlist$, and put states to be considered later on $laterlist$. The head state of $nowlist$ is removed from the list and examined using the $expand()$ function in Algorithm 7. We use $s.node$ to denote the node of s, and $s.f$ to denote the estimate distance of state s, which is the sum of the distance traversed so far and the look-ahead value of hop count. If the estimate distance is greater than the depth threshold, then the head state is put into $laterlist$. Otherwise, for each eligible neighbors, construct the state, and check its dominance relationship with states of the same node. For an un-dominated state, if the estimate distance is less than the depth threshold, put it in $nowlist$[4]; otherwise, put it into

[4]If the children states are put into the front of $nowlist$ in order of their expansion, FringeMCSP mimics IDA*_MCSP, except for the elimination of redundant search. We follow this approach in implementation.



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

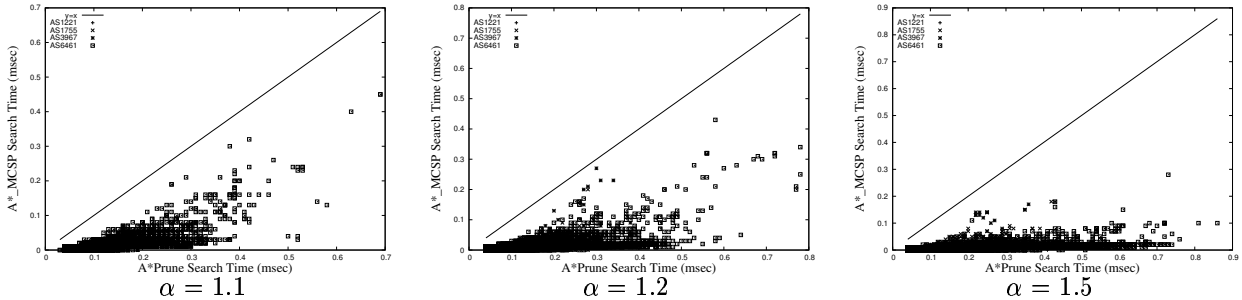
(a) Rocketfuel Topologies



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

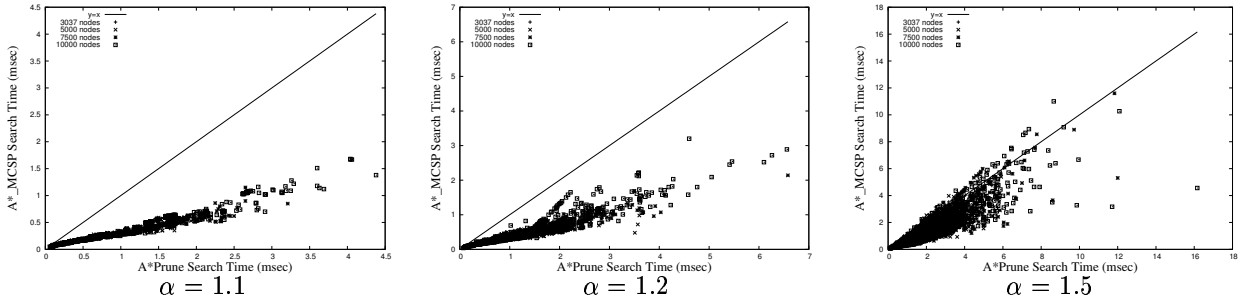
(b) Power-law Topologies

Fig. 3. Search Process Time (msec), Infeasible Case: A*Prune (x-axis) vs. A*_MCSP (y-axis)



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

(a) Rocketfuel Topologies



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

(b) Power-law Topologies

Fig. 4. Search Process Time (msec), Feasible Case: A*Prune (x-axis) vs. A*_MCSP (y-axis)

Algorithm 8 FringeMCSP()

```
1: computeLowerbound();
2: put state {src, 0, lb[hop][src], 0⃗, null} into laterlist;
3: depth_bound ← lb[hop][src];
4: done ← false
5: while done is false do
6:    nowlist ← laterlist
7:    while nowlist not empty and done is false do
8:       s ← first state in nowlist;
         /*take s away from nowlist*/
9:       done ← expand(s);
10:   end while
11:   if done is false then
12:      update depth_bound;
13:      if reach stopping condition then
14:         report failure;
15:      end if
16:   else
17:      return true;
18:   end if
19: end while
```

$laterlist$ to consider in the next iteration. When $nowlist$ is empty, update the depth threshold, copy $laterlist$ to $nowlist$, and clear $laterlist$. Search on $nowlist$ iteratively, until the goal is found or it is determined that the goal is not reachable.

The *search process* from Line 5 to Line 19 of Algorithm 8 finds the shortest constrained path.

Take the example in Figure 1 in Section II-A again. FringeMCSP searches in a similar way as IDA*_MCSP does, with the exception that FringeMCSP won't search node A, B and E for the redundantly second time.

A. Optimality and Completeness

In [14], we analyze the optimality and completeness of IDA*_MCSP. The analysis of A*_MCSP and FringeMCSP will follow a similar approach. It is based on the optimality and completeness of A* and IDA*, the search algorithms our MCSP algorithms are based on. Please see the references of [9] and [10] for the rigorous analysis of A* and IDA*. Fringe search is an integration of A* and IDA*. It only change the search process of IDA* and A*–it combines the ideas of iterative deepening and list of frontier nodes, such that the net result is that it eliminates the redundant search in IDA*–thus it preserves the property of optimality and completeness. In A*_MCSP and FringeMCSP, similar to A*Prune and IDA*_MCSP, we introduce the eligibility test, which prevents a search algorithm from searching an unnecessary space. Therefore, A*_MCSP and FringeMCSP are optimal (if feasible) and complete.

B. Performance Study of Fringe Search

Figure 5 shows that FringeMCSP has better or similar performance to A*_MCSP in infeasible cases. Figure 6 shows the results of FringeMCSP and A*_MCSP for feasible cases. Although there are some cases where FringeMCSP is not as fast as A*_MCSP, FringeMCSP is faster in most cases.

For feasible cases, IDA*_MCSP has better or comparable performance as FringeMCSP. To save space, we do not show the results. Figure 7 and Figure 8 give the average search time.

C. Summary

We present the average and the confidence interval of the search time. Figure 7 shows that for infeasible cases, FringeMCSP has a good performance. IDA*_MCSP may not perform well in infeasible cases, due to the inaccurate look-ahead information, which causes lots of redundant search. FringeMCSP makes a good tradeoff. It uses the depth bound to restrict the search depth at each iteration. It uses $nowlist$ and $laterlist$ to avoid redundant search. It does not need to sort states on the lists.

Figure 8 shows that in feasible cases, IDA*_MCSP is the most efficient. The rank is, IDA*_MCSP, FringeMCSP, A*_MCSP and A*Prune. FringeMCSP is close to IDA*_MCSP. For feasible cases, IDA*_MCSP is fast due to the accurate look-ahead information and the iterative deepening search. As a result, IDA*_MCSP does not do much redundant search. As well it does not need to maintain partial paths or states and their ordering.

In short, for a feasible MCSP problem, IDA*_MCSP is the best choice. For a MCSP problem uncertain about its feasibility, FringeMCSP may be a good option.

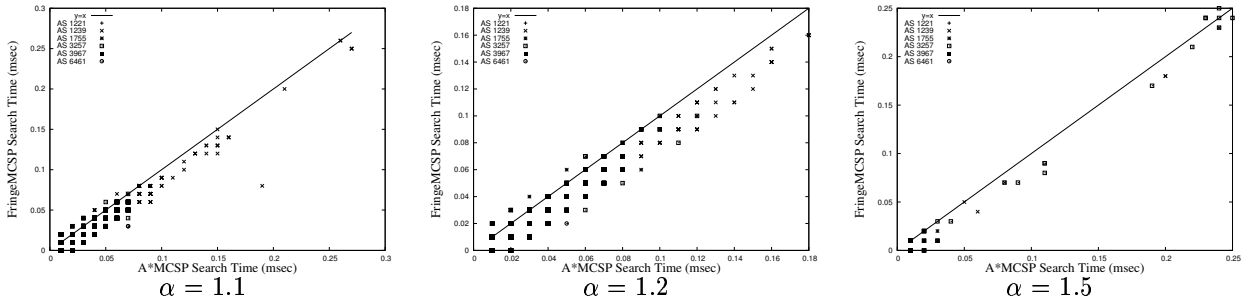
VIII. Conclusions

We deploy search techniques to design fast exact algorithms for MCSP, which was shown as NP-hard. We use highly accurate look-ahead information to predict path length, and use eligibility test to prune unnecessary search space.

We show that for infeasible cases, IDA*_MCSP [14] may not be as efficient as A*Prune. This motivates us to design an algorithm that is efficient in both feasible and infeasible cases.

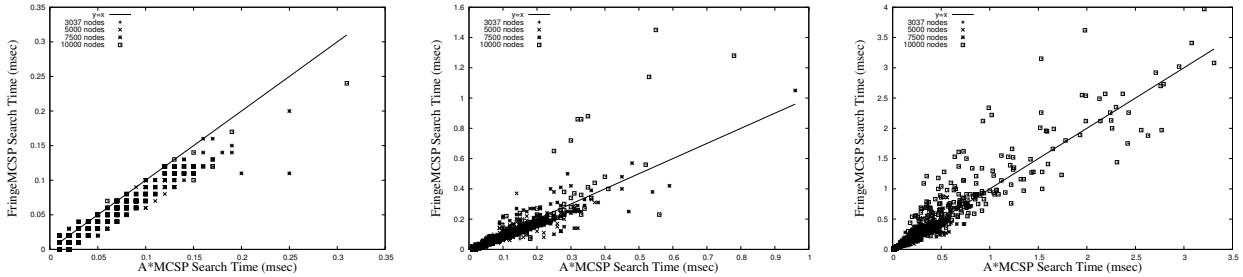
We design an A*-based exact MCSP algorithm, A*_MCSP, which introduces the state notion and dominance relationship between states. Furthermore, we design an exact MCSP algorithm FringeMCSP. This is based on a recently proposed search technique, Fringe search, which can be regarded as an integration of A* and IDA*. Consequently, FringeMCSP can be regarded as an integration of IDA*_MCSP and A*_MCSP. Extensive empirical study shows that FringeMCSP has good performance in both feasible and infeasible cases; while IDA*_MCSP still shows its superiority among the proposed MCSP algorithms in feasible cases.

We plan to study the time complexity of search-based MCSP algorithms for feasible cases. We have a conjecture that, on realistic topologies like inferred Internet ISP topologies and power-law topologies, and random topologies like those generated following the Waxman model, the look-ahead accuracy is high and these algorithms, especially IDA*_MCSP and FringeMCSP, behave in a polynomial manner in practice.



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

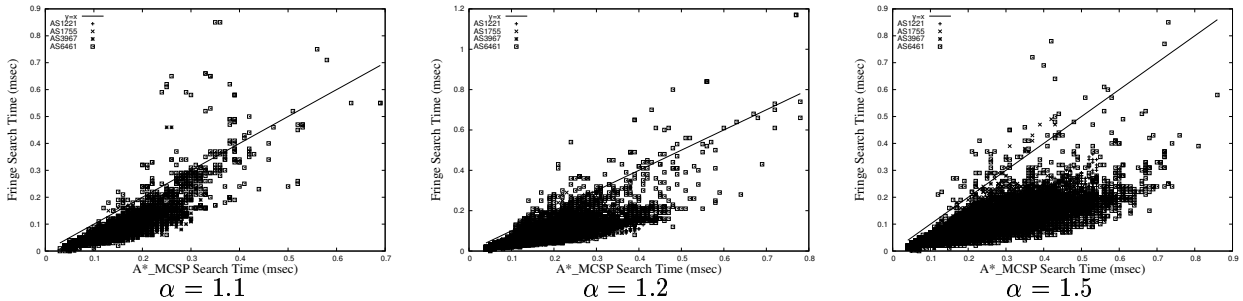
(a) Rocketfuel Topologies



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

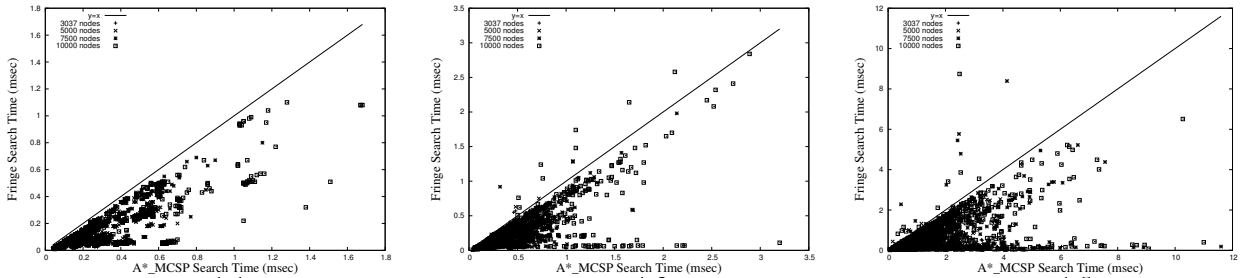
(b) Power-law Topologies

Fig. 5. Search Process Time (msec), Infeasible Case: A*_MCSP (x-axis) vs. FringeMCSP (y-axis)



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

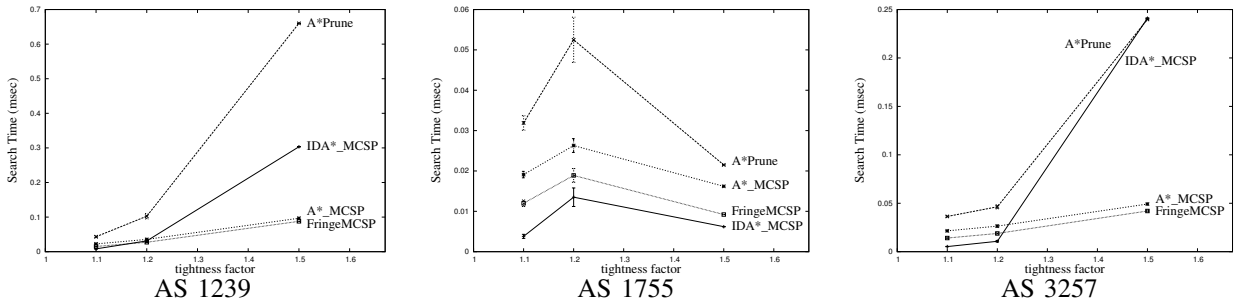
(a) Rocketfuel Topologies



$\alpha = 1.1$ $\alpha = 1.2$ $\alpha = 1.5$

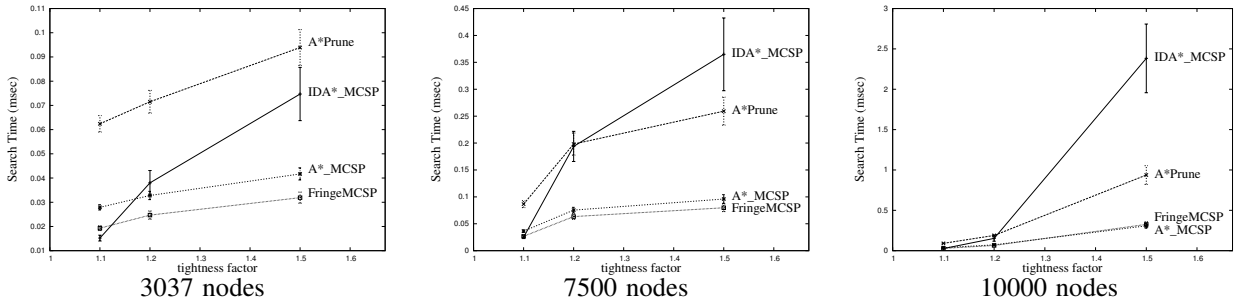
(b) Power-law Topologies

Fig. 6. Search Process Time (msec), Feasible Case: A*_MCSP (x-axis)vs. FringeMCSP (y-axis)

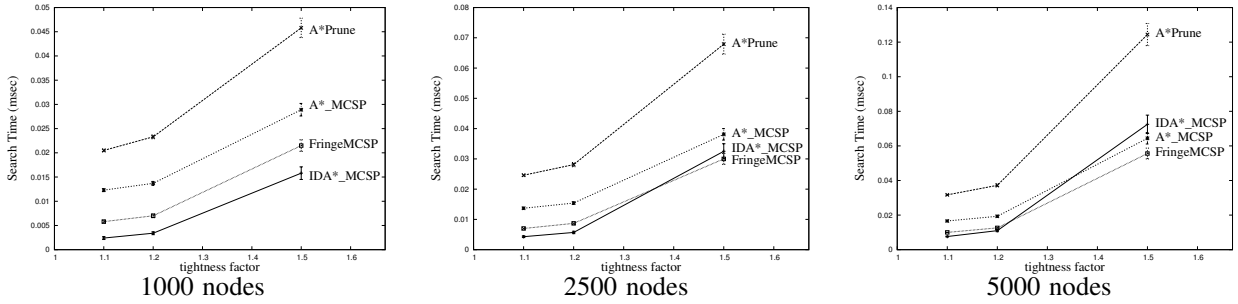


(a) Rocketfuel Topologies

(Confidence interval not reported due to small sample sizes for tightness factor $\alpha = 1.5$.)



(b) Power-law Topologies



(c) Waxman Random Topologies

Fig. 7. Average Search Process Time (msec) with 95% Confidence Interval, Infeasible Case

REFERENCES

[1] *INET Topology Generator*. http://topology.eecs.umich.edu/inet/.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of 18th ACM SOSP*, October 2001.

[3] G. Apostolopoulos, R. Guerin, S. Kamat, A. Orda, T. Przygienda, and D. Williams. Qos routing mechanisms and ospf extensions. *IETF RFC 2676*, August 1999.

[4] Y. Björnsson, M. Enzenberger, R. Holte, and J. Schaeffer. Fringe search: beating A* at pathfinding on game maps. In *Proceedings of IEEE Symposium on Computational Intelligence and Games*. Colchester, Essex, UK, April 2005.

[5] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI'91*. Sidney, Australia, 1991.

[6] S. Chen and K. Nahrstedt. An overview of quality-of-service routing for the next generation high-speed networks: problems and solutions. *IEEE network magazine*, 12(6), 1998.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[8] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of SIGCOMM'99*, 1999.

[9] P. Hart, N. Nilsson, and B. Paphael. A formal basis for the heuristic determination of minimum cost path. *IEEE Transaction on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[10] R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[11] R. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129(1-2):199–218, June 2001.

[12] F. Kuipers, T. Korkmaz, M. Krunz, and P. V. Mieghem. An overview of constraint-based path selection algorithms for qos routing. *IEEE Communications Magazine*, 40(12), December 2002.

[13] F. Kuipers and P. V. Mieghem. The impact of correlated link weights on qos routing. In *Proceedings of INFOCOM'03*, March 2003.

[14] Y. Li, J. Harms, and R. Holte. IDA*_MCSP: a fast exact MCSP algorithm. In *Proceedings of IEEE ICC'05*, May 2005.

[15] G. Liu and R. Ramakrishnan. A*prune: An algorithm for finding k shortest paths subject to multiple constraints. In *Proceedings of INFOCOM'01*, 2001.

[16] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link

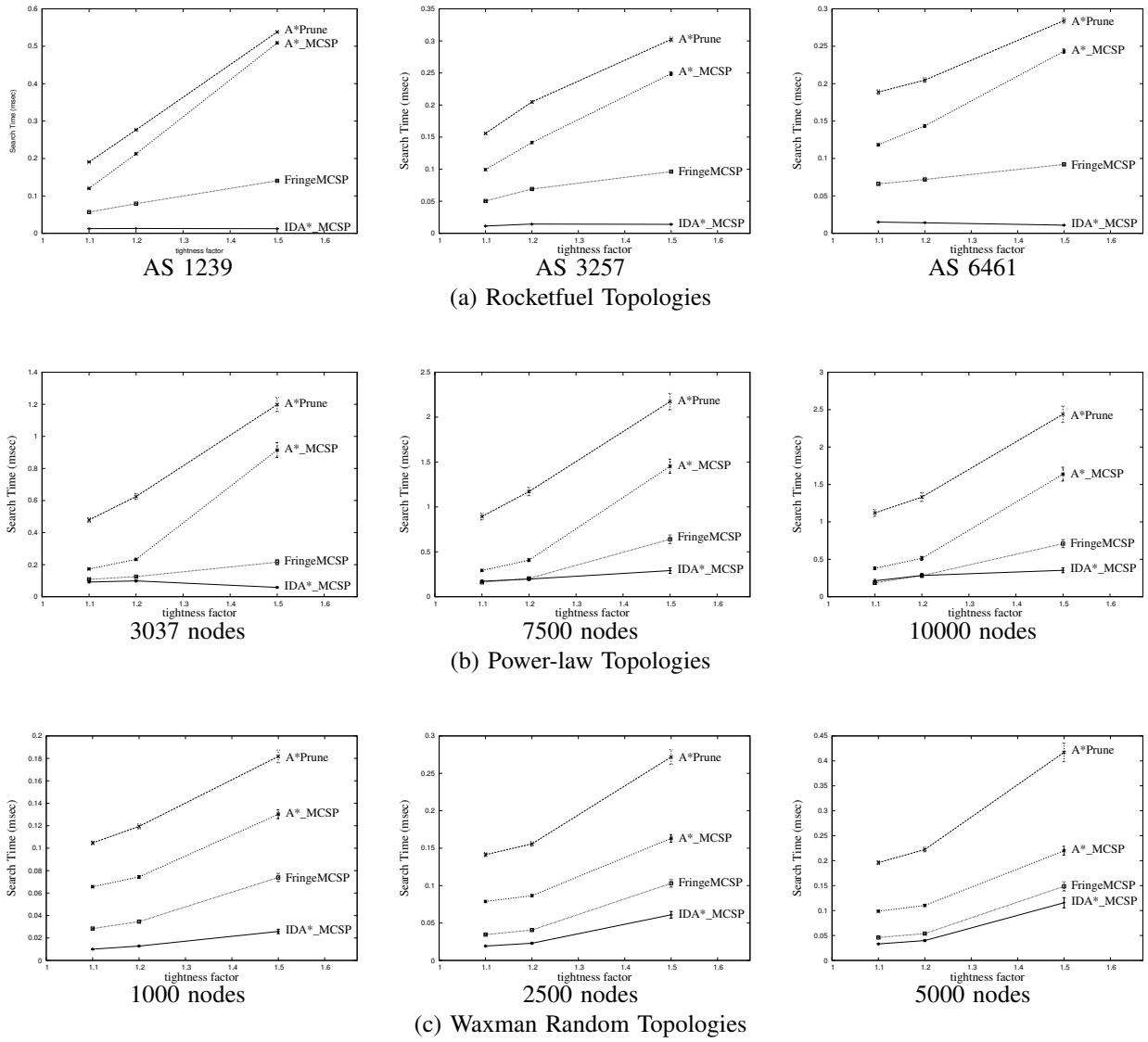


Fig. 8. Average Search Process Time (msec) with 95% Confidence Interval, Feasible Case

weights using end-to-end measurements. In *Proceedings of IMW'02*, 2002.

[17] P. V. Mieghem and F. A. Kuipers. Concepts of exact quality of service algorithms. *IEEE/ACM Transactions on Networking*, 12(5):851–864, October 2004.

[18] K. Nichols, V. Jacobson, and L. Zhang. A two-bit differentiated services architecture for the internet. *IETF RFC 2638*, July 1999.

[19] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, September 1996.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[21] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *Proceedings of ACM SIGCOMM'02*, August 2002.

[22] 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.

[23] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of INFOCOM'96*, 1996.