Search Space Reduction in QoS Routing*

LIANG GUO IBRAHIM MATTA†
Computer Science Department
Boston University
Boston, MA 02215
{guol, matta}@cs.bu.edu

December 2001

Abstract

To provide real-time service or engineer constrained-based paths, networks require the underlying routing algorithm to be able to find low-cost paths that satisfy given Quality-of-Service (QoS) constraints. However, the problem of constrained shortest (least-cost) path routing is known to be NP-hard, and some heuristics have been proposed to find a near-optimal solution. However, these heuristics either impose relationships among the link metrics to reduce the complexity of the problem which may limit the general applicability of the heuristic, or are too costly in terms of execution time to be applicable to large networks. In this paper, we focus on solving the *delay-constrained* minimum-cost path problem, and present a fast algorithm to find a near-optimal solution. This algorithm, called DCCR (for Delay-Cost-Constrained Routing), is a variant of the k-shortest path algorithm. DCCR uses a new adaptive path weight function together with an additional constraint imposed on the path cost, to restrict the search space. Thus, DCCR can return a near-optimal solution in a very short time. Furthermore, we use a variant of the Lagrangian relaxation method proposed by Handler and Zang [8] to further reduce the search space by using a tighter bound on path cost. This makes our algorithm more accurate and even faster. We call this improved algorithm SSR+DCCR (for Search Space Reduction+DCCR). Through extensive simulations, we confirm that SSR+DCCR performs very well compared to the optimal but very expensive solution.

Keywords: Quality-of-Service (QoS) Routing; Traffic Engineering; Constrained Path Optimization; Simulation.

1 Introduction

The constrained shortest-path problem is encountered in many aspects of routing in integrated-services networks. For example, delay-sensitive applications, such as real-time voice and video, require traffic to be received at the destination within a given period of time. At the same time, it is highly desirable to reduce the path cost as much as possible; this could be monetary cost or the cost of utilizing network resources. Recently, it has also been recognized that the performance of operational networks can be improved by engineering Internet traffic so as it is routed over resource-efficient constrained-based paths [5]. However, this constrained shortest (least-cost) path problem, or in general the multi-constrained optimization path selection problem, is notoriously challenging and has been proved to be NP-hard [17, 11, 7].

In this paper, we study the problem of finding a least-cost communication path subject to an end-to-end delay constraint. This problem can be formulated as a *Delay-Constrained Least-Cost (DCLC)* unicast routing problem, or more generally, a constrained optimization problem. Widyono [19] proposed an optimal solution, namely the Constrained Bellman-Ford (CBF) algorithm, to solve this problem. The CBF algorithm performs a breadth-first search to find the optimal solution, thus its running time might grow exponentially in the worst case. The algorithms in [14] and [16] try to compute the path distributively in order to alleviate the

*This work was supported in part by NSF grants CAREER ANI-0096045, ANI-0095988, and MRI EIA-9871022.
†Corresponding author. Address: Computer Science Department, Boston University, 111 Cummington Street, MCS-271, Boston, MA 02215. Tel: (617) 358-1062, fax: (617) 353-6457, e-mail: matta@cs.bu.edu.

centralized computation overhead, however, paths returned by these algorithms may be costly, and the path setup time may be too long. Reference [12] assumes that delay, delay-jitter and buffer space are functions of the available bandwidth, thus the routing algorithm can take advantage of these relationships to find a path in polynomial time. Some previous studies mainly focus on a related but possibly simpler problem — the *Multiple-Constraints Path (MCP)* problem. The difference between the MCP problem and the DCLC problem is that MCP does not optimize the value of any of the metrics, instead, it only seeks a *feasible* path that satisfies all the constraints. Nevertheless, this problem is still NP-hard if more than one metric is additive and takes real values (or unbounded integer values) [17]. Jaffe [10] proposed a pseudo-polynomial heuristic and a polynomial-time heuristic for solving the MCP problem, given that the metrics have a small range of values. In [2], Chen and Nahrstedt try to reduce the problem's complexity by approximating the real values of link metrics by integer values and then use dynamic integer programming to solve it in polynomial time. However, to accurately find a near-optimal path, this algorithm has to resort to using a high granularity in approximating the values of link metrics, thus it becomes very costly in terms of space and time complexity. In [13], a non-linear function of link cost and delay is proposed to convert the problem into the much simpler single-metric routing problem, and so as to efficiently find a path that is far away from all the metric bounds.

Since the MCP problem seems to be easier than the DCLC problem and the heuristics to the former problem are generally more efficient in terms of execution time, it appears attractive to transform a DCLC problem to a MCP problem. Based on this premise, we propose a heuristic, called DCCR (for Delay-Cost-Constrained Routing), to rapidly generate a near-optimal delay-constrained path in large networks with asymmetric link metrics (delay and cost). This algorithm first *introduces* a cost bound according to the network state, then, it employs the k-shortest path algorithm proposed by Chong *et al.* [3] with a new non-linear weight function of path delay and cost to efficiently search for a path subject to *both* the requested delay constraint and the (introduced) cost constraint. The search space is reduced as paths that now do not satisfy *both* constraints are pruned off. Our weight function is designed to give more priority to lower cost paths. This algorithm is very similar to the TAMCRA algorithm proposed in [13], but we observe that our algorithm is more suitable for solving the DCLC problem since TAMCRA has a different objective, that of solving an MCP problem. Moreover, we also notice that using a tighter cost bound may help increase the accuracy and speed of the algorithm, thus, as an improvement, we further employ the algorithm proposed by Handler and Zang [8] to refine our search space. We show by analysis that the complexity of this algorithm, called SSR+DCCR (for Search Space Reduction+DCCR), is asymptotically in the same order as a regular (unconstrained) single-metric shortest-path algorithm such as Dijkstra's algorithm [4]. Furthermore, through extensive simulations, we confirm that the cost of the path found by our SSR+DCCR algorithm is very close to that of the optimal path generated by the much more computationally expensive CBF algorithm.

The rest of this paper is organized as follows. Section 2 defines the DCLC problem. Section 3 describes our SSR+DCCR algorithm after motivating the design of our path weight function. In Section 4, we analyze the complexity of the algorithm. We compare our SSR+DCCR algorithm with some other heuristics via simulations in Section 5. We conclude the paper in Section 6.

2 Problem Description

The Delay-Constrained Least-Cost (DCLC) problem has been formulated in [14]. For completeness, we briefly restate it here. We represent the network by a directed graph $G = (V, E)$, where V is the set of all vertices (nodes), representing routers or switches, E is the set of edges (links) representing physical or logical connectivity between nodes. Each link is bidirectional, *i.e.*, the existence of a link $e = (u, v)$ from node u to node v implies the existence of another link $e' = (v, u)$ for any $u, v \in V$. Any link $e \in E$ has a cost $c(e) : E \mapsto \mathcal{R}^+$ and a delay $d(e) : E \mapsto \mathcal{R}^+$ associated with it, where $\mathcal{R}^+$ is the set of non-negative real numbers. The function $c(.)$ defines the measure we want to optimize (minimize). The function $d(.)$ defines the measure we want to constrain (bound). Due to the asymmetric nature of computer networks, it is possible that $c(e) \neq c(e')$ and $d(e) \neq d(e')$.

For a given source node $s \in V$ and destination node $d \in V$, $\mathcal{P}(s, d) = P_1, \cdots, P_m$ is the set of all possible paths from s to d. The cost and delay of a path P_i are defined as:

$$C(P_i) = \sum_{e \in P_i} c(e)$$

and

$$D(P_i) = \sum_{e \in P_i} d(e)$$

respectively. A delay constraint Δ_d is specified by the application as a performance guarantee. The DCLC problem thus can be formulated as follows:

Problem 1 (Delay-Constrained Least-Cost (DCLC) Path Problem) *Given a directed network G, a source node s, a destination node d, a non-negative link delay function $d(.)$, a non-negative link cost function $c(.)$ for each link $e \in E$ and a positive delay constraint Δ_d, the constrained minimization problem is to find a path that satisfies the following:*

$$\min_{P_i \in \mathcal{P}'(s,d)} C(P_i) \tag{1}$$

and

$$P_i \in \mathcal{P}'(s,d) \quad \text{iff} \quad D(P_i) \leq \Delta_d \tag{2}$$

where $\mathcal{P}'(s,d) \subseteq \mathcal{P}(s,d)$ is the set of paths from s to d for which the end-to-end delay is bounded by Δ_d.

The DCLC problem can be more generally categorized as a *Constrained Optimization* problem, which involves optimizing one or more variables and imposing constraints on other variables. A variation of this problem, namely the Multi-Constraints Path (MCP) problem, only searches for a feasible solution for which all variables are bounded by the constraints. A special case of the MCP problem is the *Delay-Cost-Constrained (DCC)* problem which can be stated similarly as the DCLC problem except that the objective is to find a path $P_i \in \mathcal{P}'(s,d)$, where

$$P_i \in \mathcal{P}'(s,d) \quad \text{iff} \quad D(P_i) \leq \Delta_d \quad \text{and} \quad C(P_i) \leq \Delta_c \tag{3}$$

where Δ_c is the application specified cost bound.

Both the DCLC and DCC problems are NP-hard [6], however, since DCC does not involve optimization, it appears easier to find an efficient DCC heuristic. Thus, first converting a DCLC problem to a DCC problem may help to efficiently solve the original DCLC problem. This idea is applied in our DCCR algorithm.

3 Our SSR+DCCR Algorithm

3.1 Motivation

We convert the DCLC problem into a DCC problem by defining an appropriate cost bound for DCC so that the solution to the original DCLC problem remains a feasible solution to the new DCC problem. This could be easily achieved by using a sufficiently loose cost bound. In our algorithm, we solve a DCC problem, where we start with the least-delay path as a possible feasible solution. The cost of the least-delay path is selected as the *cost bound*. The goal is then to search for a feasible path of possibly higher delay but of lower cost. If such a path does not exist, then the least-delay path itself must be the optimal (least-cost) feasible path and this is what our algorithm returns. Thus, we can convert the original DCLC problem into the problem of searching for a near-optimal path in the solution space of this new DCC problem. As illustrated in Figure 1, the solution space of DCC is clearly smaller than that of DCLC since there are fewer paths that satisfy *both* delay and cost constraints.

To search the solution space for the least-cost path, we need to examine the feasible paths for the DCC problem (i.e., those paths that satisfy both the requested delay bound and the introduced cost bound). For this purpose, we can use any well-known shortest-path algorithm (e.g. Dijkstra, Bellman-Ford [4]). But since these algorithms only deal with a single metric, we need to define a weight function which combines all features of the link metrics so that by optimizing (minimizing) the weight, we will finally find a solution that minimizes all link metrics simultaneously. A simple way to mix the metrics is to use a linear function, for example, $w(e) = \alpha c(e) + \beta d(e)$ as the new weight for each edge. Indeed, this is the approach taken in [HZ80], where α and β were chosen through an optimization step to maximize the chance of finding the optimal

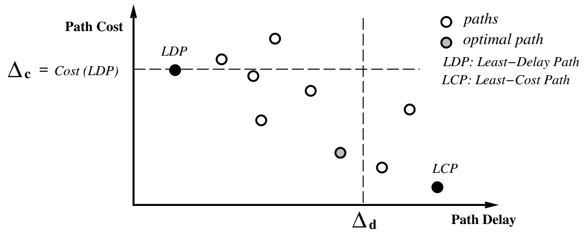


Figure 1: Reduced search space under DCC

path. This approach has the advantage that it is easy to implement since now the multiple constraints on path delay and cost become a single path weight constraint $\Delta = \alpha\Delta_c + \beta\Delta_d$. However, this linear weight function may not reflect the actual quality of a path, *i.e.*, an optimal path according to the new weight function may in fact violate the constraints while a suboptimal path satisfies them. Figure 2 shows why a linear function may not work.

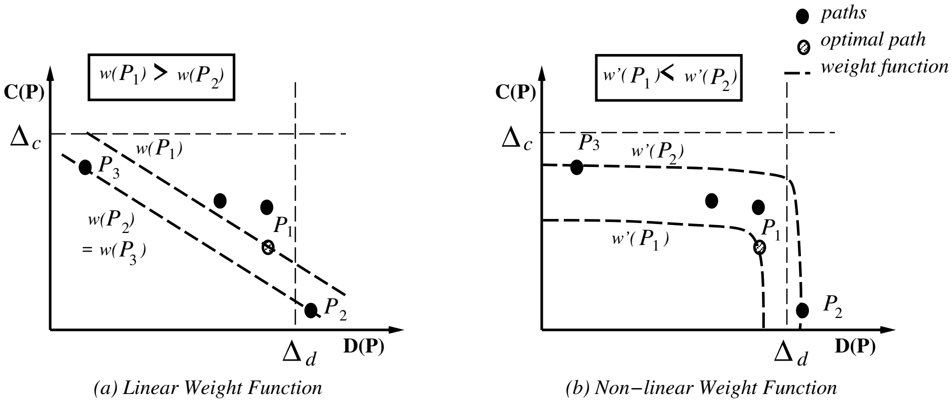


Figure 2: Why a linear function fails?

We can see from Figure 2(a) that although the delay of path P_2 violates the delay bound, it still has the least weight since its cost is relatively low, thus, the search process may miss the actual optimal *and* feasible path P_1. We can also see from the illustration that for the DCLC problem, the feasible path P_3 may be returned by the linear weight based algorithm as it has a relatively low delay, and we miss the actual optimal (least-cost) path P_1. Furthermore, when the number of candidate paths is large, using a linear function may suffer from slow convergence.

Using a non-linear function may help overcome this difficulty. De Neve and Van Mieghem propose to use the concave path weight function $\max(C(P)/\Delta_c, D(P)/\Delta_d)$ in their TAMCRA algorithm [13]. It is shown that with this function, the algorithm can find the shortest path (path whose *both* cost and delay are far from the bounds) with a relatively high success rate. Figure 2(b) illustrates the advantage of using a non-linear weight function over a linear one. Now, the search process would not return P_2 nor P_3 since they have high delay and high cost respectively, resulting in a high weight value.

The problem with defining a non-linear weight function for a link is that now the weight of a path is no longer the sum of the weight of all links on this path, *i.e.*, $W(P) \neq \sum_{e \in P} w(e)$. But since it is easy to record the cumulative delay and cumulative cost of a path, we can easily solve this problem by computing the path weight as a function $\mathcal{F}(.)$ of the delay and cost of the path (rather than as the sum of link weights), *i.e.*, $W(P) = \mathcal{F}(C(P), D(P))$.

A more serious problem is that a non-linear function does not have the *optimal-substructure property* [4], *i.e.* subsections of shortest (least-weight) paths are not necessarily shortest paths themselves. Therefore, a shortest-path algorithm like Dijkstra's will sometimes fail to find the shortest (least-weight) path. Consider the following example shown in Figure 3 assuming a concave (max) weight function is used.

For intermediate node u, path P_2 will be chosen since it has a smaller weight[1], thus the actual feasible

[1] $W(P_1) = max(10/12, 1/12) = 10/12$, $W(P_2) = max(5/12, 5/12) = 5/12$.

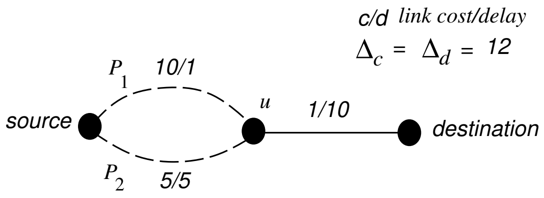


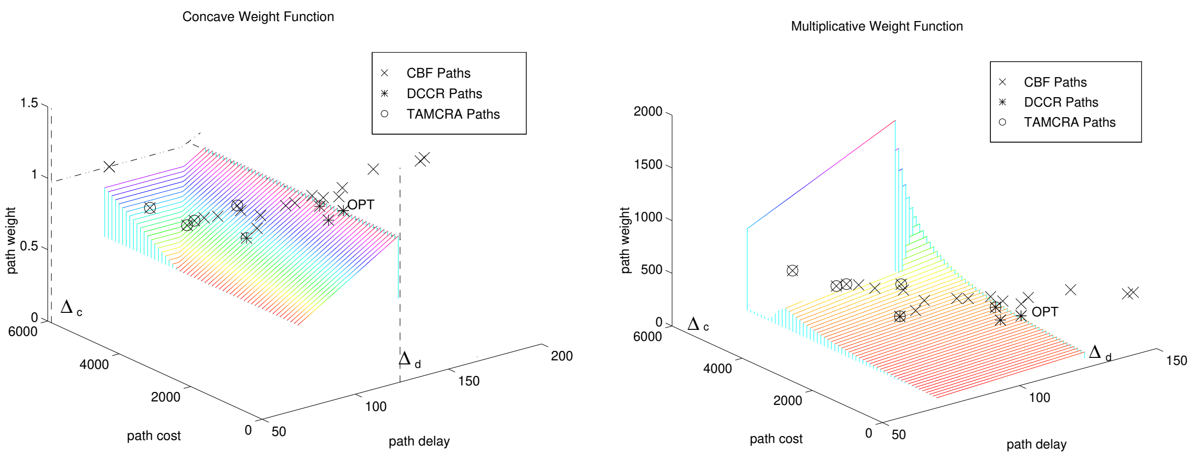
Figure 3: Problem with a non-linear function

path to the destination through P_1, with feasible delay and cost of 11, will be missed. De Neve and Van Mieghem solve this problem by taking advantage of the k-shortest path algorithm proposed by Chong *et al.* [3], which can store k shortest paths in increasing weight order at each node. Thus with an appropriate value of k, the algorithm can almost always find the least-weight and feasible path.

The non-linear (max) weight function in TAMCRA works well so as to find a path that is far from all the bounds. It is not a goal of TAMCRA to optimize any of the metrics. However, since now our objective is to find a path with least cost, this function is no longer suitable since it treats all link measures equally. Instead, we should use a weight function that gives priority to low-cost paths. The weight function used in our algorithm is defined as:

$$W(P_i^u) = \begin{cases} \frac{D(P_i^u)}{1-C(P_i^u)/\Delta_c} & \text{if } D(P_i^u) \leq \Delta_d \text{ and } C(P_i^u) \leq \Delta_c \\ \infty & \text{otherwise} \end{cases} \tag{4}$$

where $P_i^u \in \mathcal{P}(s,u)$ is the i^{th} path from source node s to node u found by the algorithm. With our definition, the path weight has an exponential growth with the path cost, and is only linearly proportional to the path delay. The difference between this new (multiplicative) function and the concave (max) function defined in TAMCRA algorithm can be well illustrated in Figure 4.



(a) (b)

Figure 4: Paths selected by different heuristics and their weights: (a) concave path weight function used by TAMCRA, and (b) multiplicative path weight function used by our DCCR algorithm

In Figure 4, 'x's represent the paths visited by CBF (the optimal algorithm), 'o's represent paths visited by the TAMCRA algorithm, and '*'s represent paths visited by our DCCR algorithm. We can see that with the concave (max) function, it is almost impossible for TAMCRA to find the optimal (least-cost) path denoted by OPT, whose delay is close to the delay bound. On the other hand, with our multiplicative function used in our DCCR algorithm, paths that are close to the optimal one will have a better chance to be visited. Figure 5 shows these paths in a two-dimensional (path cost/delay) view (HZ_1 paths will be explained later in Section 3.3). The path marked with '~' is the final solution returned by the TAMCRA algorithm, while '-' is the final solution returned by our algorithm. We notice that in this example, the

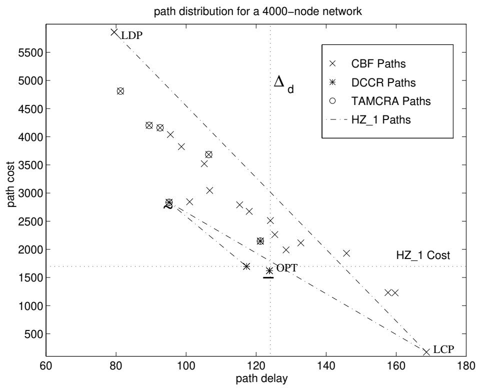


Figure 5: Path distribution in cost-delay plane, network size = 4000, node degree 4, negative correlation between cost and delay

solution of DCCR is exactly the optimal path, while the cost of the TAMCRA path is much higher than that of the optimal solution. The effect of the weight function is reflected in the difference in areas that TAMCRA ('o' paths) and DCCR ('*' paths) visited. Clearly, the area visited by our DCCR algorithm is closer to the optimal solution.

Figure 6 shows an example to summarize the idea behind DCCR. Path weights are computed using equation (4). The least-delay path from source S to destination D is (S, C, E, B, D). The cost of that least-delay path is taken as an additional cost bound of 6. DCCR returns path (S, A, B, D) as it has lower weight *and* cost, while reducing the search space by pruning off other *infeasible* paths.

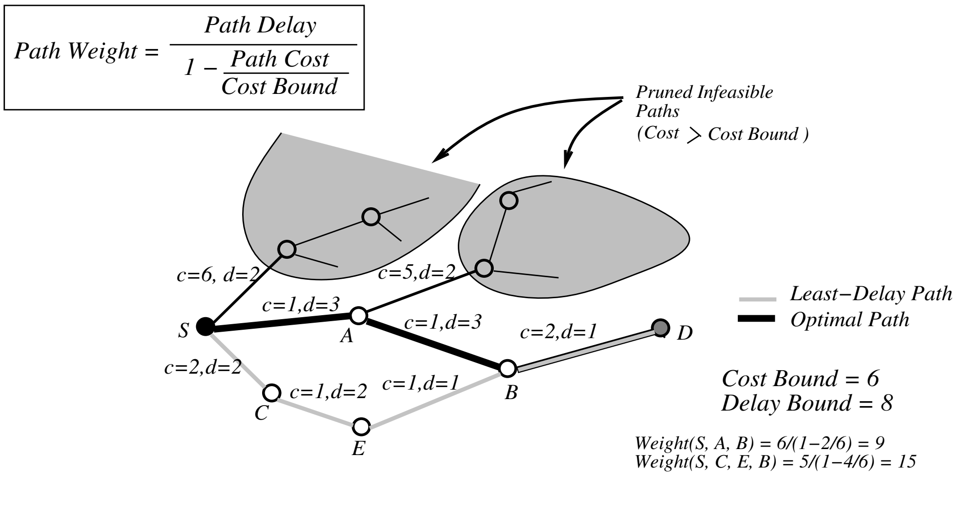


Figure 6: DCCR example

3.2 Algorithm Description

We now describe in detail how our DCCR algorithm works. DCCR takes the same greedy strategy as in Dijkstra's algorithm, but uses a non-linear weight function in searching for the best solution. However, as mentioned earlier, a non-linear weight function does not have the *optimal-substructure property*, thus only recording one best path from the source node to each node may lead to failure in finding the optimal path. The DCCR algorithm solves this problem by applying Chong's k-shortest paths algorithm [3], which records k shortest paths, listed in increasing weight order, for each node. Therefore, out of the k shortest paths, we can pick up and return the path with the lowest cost in the final stage as the best feasible solution. This way, we have more candidate paths to every node, which increases the chance of finding an optimal feasible path (cf. Figure 3).

The k-shortest paths algorithm (see Appendix A for pseudo-code) is basically an extension of the Dijkstra's algorithm. The basic idea of this algorithm is to maintain a k-element array for each node to record the currently k best paths from the source to this node. [3] proves that at most k shortest paths need to be maintained at each node to find k shortest paths from a source to a destination node. The algorithm uses a heap to store the nodes that have not yet been visited k times. Each element of the heap has fields n_id, wgt and idx, where n_id identifies the node and idx locates an element of the array $ND(n\_id, .)$ of the k-shortest paths associated with the node. The heap operations are based on node weight wgt. The relaxation step is almost the same as in Dijkstra's algorithm, except that when an unvisited node's weight is updated, the corresponding element in the heap also needs to be updated. More specifically, if the weight of current path is less than the weight of one of the k paths recorded, then the recorded path with the maximum weight is replaced by the new path (lines $17-23$ in the pseudo-code).

Since the original k-shortest paths algorithm may return a path that contains loops, we use the same non-dominated strategy as in [13]. A path p is said to be *dominated* by another path p' if and only if $D(p) > D(p')$ and $C(p) > C(p')$. We do not allow the algorithm to visit a dominated path in our relaxation step (line 18). Since delay and cost are additive metrics, a path that contains a loop will always be dominated by the corresponding loop-free subpath. Thus, the final candidate solutions will not contain any paths with loops.

3.3 Improvement to DCCR algorithm

Recall that our DCCR algorithm restricts the search space by only examining paths that satisfy the requested delay bound *as well as* a cost bound. The cost bound is taken to be the cost of the least-delay path. This is reasonable since if there is no path with lower cost than that of the least-delay path, then the least-delay path itself is the optimal path, and this is the path returned by DCCR. However, this cost bound may be too loose, especially when the relationship between cost and delay is negative, *i.e.*, the lower the delay, the higher the cost and vice versa. Since we set the weight of all infeasible paths to be infinity, it is easy to see that if we use a tighter cost bound, the number of possible feasible solutions decreases, and thus the opportunity that our algorithm finds the optimal (least-cost) solution increases. Another advantage of using a tighter bound is that since the success rate becomes higher, only a small value of k would work well, thus the speed of the algorithm is also enhanced.

To search for a tighter cost bound, we use another heuristic to the DCLC problem as a prelude to our DCCR algorithm. This heuristic, namely the HZ_1 algorithm, was proposed by Handler and Zang [8]. It still uses a linear function of the link delay and cost to compute link weight. A salient feature of it is that it adjusts the weights given to cost and delay in the weight function according to the quality of the current path, thus it iteratively approaches the optimal (least-cost) solution.

Figure 7 illustrates how the HZ_1 algorithm works (the pseudocode is given in Appendix B.) The algorithm starts from two paths: the least-delay path (LDP) and the least-cost path (LCP), computed using any shortest-path algorithm (Dijkstra's algorithm in our simulations) with the weight function being link delay and link cost, respectively (lines $1-2$). If LCP is a feasible (delay-bounded) path, then it is the optimal solution and the algorithm stops. If it is not feasible, then at each iteration, the algorithm maintains two solutions (paths), the current best feasible (delay-bounded) path LDP and the current best infeasible path LCP. It then defines two parameters α and β (line 7) to construct a new linear path weight function $W(p) = \alpha * D(p) + \beta * C(p)$ for each path p, which is represented by the dashed lines in Figure 7. Using this new linear function of link cost and delay, the algorithm tries to find a new path LWP with least weight so as to reduce both path cost and delay. If successful (that is, when $W(LWP) < \gamma$ where γ is the current least path weight) and LWP is feasible (i.e. $D(LWP) \leq \Delta_d$ as case 1 of Figure 7), LWP replaces LDP to become the best feasible path, thus the weight given to link cost increases in the next round (i.e. lower cost paths are given more preference). If LWP is infeasible (case 2), LWP replaces LCP in the next iteration, thus the weight given to link delay increases (i.e. lower delay paths are given more preference). The algorithm stops when no more progress can be made (that is, when $W(LWP) = \gamma$, lines $8-12$) and returns the best feasible path out of LWP and LDP as the near-optimal solution.

An example of how HZ_1 works is also shown in Figure 5. All HZ_1 paths are linked by a dashed dotted line in the order in which they have been visited. We can see that the cost of the HZ_1 path is getting closer to that of the optimal path at each iteration. The path found by the HZ_1 algorithm may still not be the optimal path due to the inherent weakness of the linear weight function (cf. Figure 2). But its cost is close enough to the optimal cost to be effectively used as a tight cost bound for DCCR. We denote this improved algorithm by SSR+DCCR since using a tighter cost bound is a mechanism for Search Space Reduction. Note

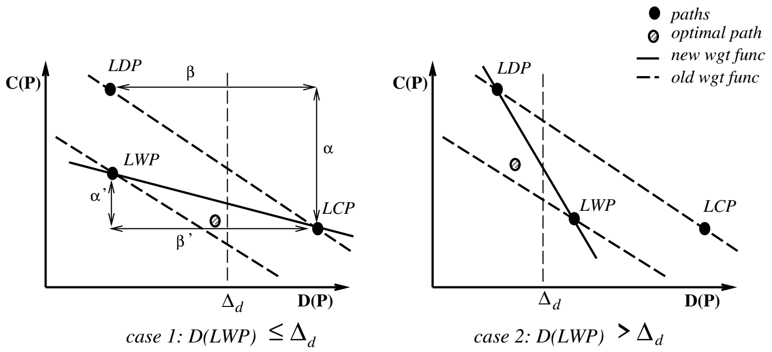


Figure 7: HZ_1 algorithm

that HZ_1 has unbounded time complexity, whereas SSR+DCCR's time complexity is bounded since HZ_1 is only used as a prelude to DCCR with a *very small* number of iterations.

3.4 Comparison to HZ_k Algorithm

Handler and Zang [8] proposed an algorithm which uses HZ_1 as the initial step, then uses a k-shortest path algorithm in the second step. We henceforth refer to this complete Handler and Zang's algorithm as HZ_k. Unlike our SSR+DCCR algorithm, HZ_k uses a *linear* path weight function in its second step. Due to the inherit weakness of the linear function, HZ_k might take a significant amount of time (in other words, significantly high value of k would be required) to find a close-to-optimal solution. This weakness was indeed illustrated in the conclusion part of [8] by an example. Briefly speaking, if the linear path weight function returned by HZ_1 is a "bad" function, k has to be very large (close to the number of paths in the network) to approach the optimal solution. However, our method overcomes this problem by using a non-linear function (as illustrated by Figure 2) which gives higher preference to low cost paths. In other words, since the HZ_k algorithm does not change the "optimality" of the weight function in its second stage, the convergence to the optimal solution is much slower than that of our SSR+DCCR algorithm. Thus, our algorithm can approach the optimal solution with a much smaller k in the worst case.

4 Algorithm Analysis

4.1 Correctness of SSR+DCCR

Theorem 1 *There exists k such that SSR+DCCR always returns a delay-constrained path for a given source s and destination d, if such a path exists.*

Proof: If no feasible path exists, *i.e.*, the delay of each path that connects s and d is greater than the delay bound, then the minimum path weight computed at node d will have a weight of infinity (cf. equation (4)). Thus no relaxation step will be made at d, which means SSR+DCCR returns no path when the search is over.

Now we prove by contradiction that SSR+DCCR returns a path if one or more feasible paths exist. If there are one or more feasible paths, the only possible reason for SSR+DCCR to return no path is if the algorithm finds no feasible path leading to an intermediate node along the feasible path from s to d. In other words, let $P_i^d = \{s, v_1, v_2, \cdots, v_m, d\} \in \mathcal{P}(s,d)$ be a feasible path from s to d, and v_1 to v_m are intermediate nodes, we would have the following two conditions satisfied:

$$\exists P_i^d \;\; s.t. \;\; D(P_i^d) \leq \Delta_d \tag{5}$$

and

$$\exists v_j \in P_i^d \;\; s.t. \;\; \forall P_l^{v_j} \in \mathcal{P}(s, v_j), \quad D(P_l^{v_j}) > \Delta_d \tag{6}$$

However, since delay is an additive metric, and we had assumed that link delay is non-negative, it is not possible that the subpath of a feasible path is not feasible. This contradiction shows that SSR+DCCR can always find a feasible solution if there is one. □

Theorem 2 *The final path returned by SSR+DCCR for a given source s and destination d is loop-free.*

Proof: Since the algorithm does not visit dominated paths, a path that contains a loop is never recorded and thus the final k shortest paths recorded at node d are loop-free, and so is the final path returned. □

4.2 Complexity Analysis

Lemma 1 *The time complexity of the original DCCR algorithm is $O(k|E|\log(k|V|) + k^2|E| + t(\mathcal{A}))$, where $\mathcal{A}$ is any single-metric shortest-path algorithm and $t(\mathcal{A})$ is the time complexity of $\mathcal{A}$.*

Proof: The task of extracting a minimum element from a binary heap takes $O(\log(k|V|))$. Since the algorithm considers a maximum of $k|V|$ paths, thus we need to extract $k|V|$ elements at worst, which gives $O(k|V|\log(k|V|))$. At most $k|E|$ edges can be used in the relaxation process. Each relaxation step includes extracting the maximum element from the k-array of the neighbor node ($O(k)$), one heap search operation ($O(\log(k|V|))$), and one heap replacement or insertion operation ($O(\log(k|V|))$). Thus the total time spent for each relaxation step is $O(k|E|(\log(k|V|) + k)) = O(k|E|\log(k|V|) + k^2|E|)$. We also need to compute a cost bound for DCCR by running a shortest-path algorithm $\mathcal{A}$ to find the cost of the least-delay path in $t(\mathcal{A})$ time. If $\mathcal{A}$ is the Dijkstra's algorithm and the priority queue is implemented as a binary heap, then the total time complexity of the DCCR algorithm is $O(k|E|\log(k|V|) + k^2|E| + |E|\log|V|)$ for $|E| > |V|$. □

Lemma 2 *The time complexity of the HZ_1 algorithm is $O(m(G)(|E| + t(\mathcal{A})) + 2t(\mathcal{A}))$, where $m(G)$ is the total number of executed iterations, $\mathcal{A}$ is any single-metric shortest-path algorithm and $t(\mathcal{A})$ is the time complexity of $\mathcal{A}$.*

Proof: The outer loop executes $m(G)$ times, and at each iteration, we need to compute the weight of at most $|E|$ edges, and also compute the least-weight path in $t(\mathcal{A})$ time. Thus, the total time is $O(m(G)(|E| + t(\mathcal{A})))$. We also need to run $\mathcal{A}$ before the loop to find the least-delay path and least-cost path, which takes $2t(\mathcal{A})$. If $\mathcal{A}$ is the Dijkstra's algorithm and the priority queue is implemented as a binary heap, then the time complexity becomes $O(m(G)(|E| + |E|\log|V|) + 2|E|\log|V|) = O((m(G) + 2)|E|\log|V|)$. □

Lemma 3 *The time complexity of the improved SSR+DCCR algorithm is $O((m(G)+2)|E|\log|V|)+O(k|E|\log(k|V|)+k^2|E|)$.*

Proof: Follows directly from Lemma 1 and Lemma 2. □

Note that $m(G)$ depends on the configuration of the graph, thus the complexity of the HZ_1 algorithm is not known. However, as found in [8, 9], this number is relatively small since the heuristic converges very fast (50% gain at each iteration). And since the main purpose of using the HZ_1 heuristic in our algorithm is just to provide a tighter cost bound, we add an upper bound on the number of iterations in our implementation of the HZ_1 algorithm. Now the time complexity of the improved SSR+DCCR algorithm becomes $O(m|E|\log|V|) + (O(k|E|\log(k|V|) + k^2|E|)$, where m is the upper bound. With a small k, we can argue that the time complexity of the improved SSR+DCCR algorithm is only $k + m$ times that of the regular (unconstrained) Dijkstra's algorithm. Actually, applying the HZ_1 heuristic helps to reduce the value of k as our simulations indicate in the next section.

5 Simulation Model and Results

5.1 Simulation Model

We built a discrete-event simulator to investigate the performance of different algorithms in a realistic communication environment. We use the same graph generation process as in [15] where the positions of the nodes lie in a rectangular area. A random generator based on Waxman's generator [18] is used to create links interconnecting the nodes. Some modifications are added to ensure that the generated network is connected and the probability of existence of a short link is larger than that of a longer link. We fixed the position of the source node s and the destination node d such that the *Manhattan distance* between s and d is the longest possible distance in the graph. The average node degree is set to 4, which is approximately what the situation is in current networks.

The link delay function consists of the propagation delay T_p, the transmission delay T_t and the queuing delay T_q. Since we are considering high-speed links, transmission delay is assumed negligible. Denote by $\tau = T_q/T_p$ the ratio between the queuing delay and propagation delay; this parameter reflects how busy the communication link is. Thus, the link delay is defined as:

$$d(e) = (1 + \tau) \times T_p$$

In our simulation model, we let τ be uniformly distributed in $[0, T]$, where T is a parameter that reflects the maximum queueing delay allowed at each switch. Also, the larger the value of T is, the more likely the generated network is asymmetric. We set T to be 10.0 in our experiments.

The way to generate the link cost can affect the difficulty in finding the optimal path. If there is a *positive correlation* between link cost and delay, *i.e.* the higher the link delay is, the more costly the link is, then it is enough to just use a single-metric shortest-path algorithm since faster paths are also likely to be cheaper. Thus, in our simulation model, we consider the most difficult situation where a *negative correlation* exists between cost and delay, for example, to model the case that high-speed (low-delay) links cost more. We define link cost as:

$$c(e) = M/(c + d(e))$$

where M and c are parameters chosen so as to adjust the value of $c(e)$ within a reasonable range. We choose $M = 1000$ and $c = 1$ in our simulations, and $d(e)$ varies from 0.1 to 20.

Since the tightness of the delay bound might affect the performance of the algorithms under investigation, we choose the delay bound based on the configuration of the graph. Each time a new graph is generated, we first use Dijkstra's algorithm to find the least-delay path and least-cost path, then compute the delay of these two paths, denoted by $D(LDP)$ and $D(LCP)$ respectively. We then define the delay bound Δ_d to be:

$$\Delta_d = D(LDP) + \rho(D(LCP) - D(LDP))$$

where $\rho \in [0, 1]$ is called the *delay bound ratio* [9] and reflects the tightness of the delay bound. In most of our experiments, ρ is set to 0.5.

We assume a link-state type routing, where the routing nodes have complete knowledge of the state of the entire network, and the state information is accurate (up-to-date). The network size is set to 200, 500, 1000, and 2000. 500 executions on different networks are conducted for each experiment, and 95% confidence intervals were computed for all performance measures. We choose $k = 3$ and $m = 5$ for all network sizes, where k is the number of shortest (least-weight) paths maintained from the source to each node, and m is the number of HZ_1 iterations executed to compute a tight cost bound for DCCR. Note that k and m are much smaller than the network size, but we will see shortly that even such a small value is enough to get good performance.

5.2 Performance Metrics

Two performance metrics are used to measure the inefficiency (inaccuracy) and speed of the heuristics. As mentioned earlier, the CBF algorithm provides the optimal solution in terms of path cost. Thus we define the *inefficiency* of an algorithm $\mathcal{A}$ as the path cost difference relative to the cost of the CBF path:

$$inefficiency_{\mathcal{A}} = \frac{(C(P_{\mathcal{A}}) - C(P_{CBF}))}{C(P_{CBF})}$$

We also measure the *actual execution time* of each investigated algorithm. The experiments were conducted on a SUN Ultra 10 workstation.

5.3 Simulation Results

Figure 10 shows the performance measures of different heuristics for different network sizes. Confidence intervals are also shown[2]. We can see that with negative correlation between link cost and delay, the least-delay path can cost as high as three times (200% more than) that of the optimal path. HZ_1 returns a

[2] We use log scale for the cost axis due to the huge difference between the algorithms under investigation.

path that costs about 4-5% higher than the optimal path. Increasing k to 3 in the HZ_k algorithm does not significantly reduce the excess cost. DCCR, which only takes advantage of the non-linear function, does not have much advantage over its counterpart HZ_1; they both achieve about the same cost level. The improved SSR+DCCR algorithm, however, as a combination of HZ_1 and DCCR, shows a very attractive cost performance; *the relative excess cost of SSR+DCCR always remains under 1%.* We can also see that the relative order and the scale of cost difference does not change much with the network size. Considering that we set $k = 3$ for all network sizes, we can then argue that *k can be kept small even for a very large network.* As for the execution time, Figure 10(b) shows the data for all heuristics except the CBF algorithm. The LDP (Dijkstra's least-delay path algorithm) runs the fastest. Then comes the HZ_1 algorithm, which implies that practically, HZ_1 can converge very fast to the final solution even though an analytical bound does not exist. The speed of DCCR is a little bit slower than HZ_1 since DCCR uses a non-linear path weight function and requires a k-shortest path algorithm. Not surprisingly, HZ_3 slows down the process by a factor of 3. What is surprising is that the improved SSR+DCCR algorithm, *i.e.* the combination of HZ_1 and DCCR, runs in almost the same speed as the original DCCR algorithm, which implies *a more efficient search under SSR+DCCR.* The speed of both algorithms using a non-linear path weight function, DCCR and SSR+DCCR, is *only about 4 times* that of the LDP algorithm. This result confirms the complexity analysis in Section 4.2 since we use $k = 3$ in this experiment.

We also compared the speed of the optimal CBF solution and the SSR+DCCR algorithm in Figure 11. It is clear that the CBF algorithm has an exponential growth with the network size in terms of its execution time, as opposed to the polynomial growth of the SSR+DCCR algorithm. The difference in execution time, in a 2000-node network, can be as high as 2 orders of magnitude.

To see the role k plays in the performance of the heuristics that use the k-shortest path method, we conduct another experiment whose results are shown in Figure 12. Increasing k's value results in more candidate paths being examined, thus both heuristics, namely HZ_k and SSR+DCCR, return a cheaper path as k becomes larger. However, for HZ_k, this performance improvement is offset by a large increase in execution time. Since in SSR+DCCR, we already use the HZ_1 heuristic to find a tighter cost bound, there aren't many feasible paths left in the solution space, thus, a small k is enough to find a good (low-cost) path. Thus, *SSR+DCCR requires less space and time.* On the contrary, to achieve the same performance level (e.g., less than 1%), HZ_k requires a much larger k value (greater than 8 in this example), which implies a much longer execution time (about 4 times higher). We can also observe from this figure that, due to the strict admission policy of the non-linear function, the number of candidate paths is relatively small for our SSR+DCCR algorithm, thus, increasing k does not significantly increase the execution time.

Figure 13 shows the effect of the delay bound on the performance. We can see that the relative excess cost of HZ_1, HZ_k, and DCCR, is increasing as the delay bound gets looser. This is because a looser bound will enlarge the solution space, thus the capability of these algorithms becomes limited by either the weakness of the linear weight function (cf. Figure 2) or by the fixed value of k (cf. Figure 12). On the contrary, *the performance of SSR+DCCR is less sensitive to the delay bound.* As analyzed earlier, this is because the cost bound given by the HZ_1 heuristic is already tight enough to restrict the number of feasible paths.

All the above experiments assume that the link cost and link delay are negatively correlated. This assumption is valid for some networks, and increases the difficulty in finding the optimal path. We also note that in some cases, the link cost may not have any relationship with the link delay. Thus, in the next experiment, we assume that link cost is a random number, and is not correlated to link delay. Figure 14 shows the performance of all the investigated algorithms for different network sizes. The relative order of the heuristics remains the same, *i.e.* SSR+DCCR performs the best, followed by DCCR, HZ_3, then HZ_1. LDP performs the worst. The difference here is that the performance gaps are smaller. However, we still confirm that *SSR+DCCR produces paths whose cost is very close to the optimal cost at competitive speed.*

6 Conclusions

An efficient algorithm for obtaining a Delay-Constrained Least-Cost (DCLC) path is presented in this paper. This algorithm uses a non-linear path weight function and applies a k-shortest path heuristic to make the path search more accurate and faster. To further enhance the accuracy and speed of the algorithm, we also propose to use another DCLC heuristic that uses a linear path weight function as a prelude to further reduce the solution/search space. Results from extensive simulations show that even under the most difficult situation, *i.e.*, when link cost and link delay are negatively correlated, our improved SSR+DCCR algorithm always returns very quickly a feasible path whose cost is very close to that of the optimal one, which could only be found using a computationally prohibitive search method.

Our SSR+DCCR algorithm could be applied in multicast routing protocols to build a low-cost multicast tree. Since it is common that the membership of a multicast group is dynamic, and the network state (link delays and costs) is also dynamic, it is very hard, if not impossible, to maintain all the time an optimal cost multicast tree that also satisfies given performance (e.g. delay) constraints. One possible solution to this problem is to, whenever a new group member joins or an existing member becomes out-of-bound, add or replace the old path with a new delay-bounded path. Thus, reducing the cost of this delay-bounded path can further reduce the cost of the whole tree. We will investigate this approach in our future work. We are also investigating using our algorithms for traffic engineering in Diffserv environments [1].

References

[1] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[2] S. Chen and K. Nahrstedt. On Finding Multi-constrained Paths. In Proc. *ICC '98*, Atlanta, Georgia, 1998.

[3] E.I. Chong, S. Maddila, and S. Morley. On Finding Single-Source Single-Destination k Shortest Paths. In Proc. *International Conference on Computing and Information (ICCI) '95*, pages 40–47, Ontario, Canada, July 1995.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1990.

[5] Internet Traffic Engineering. IETF Working Group. ftp://ftpext.eng.us.uu.net/tewg.

[6] M.R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[7] R. Guerin and A. Orda. QoS-based Routing in Networks with Inaccurate Information: Theory and Algorithms. In Proc. *IEEE INFOCOM'97*, 1997.

[8] G.Y. Handler and I. Zang. A Dual Algorithm for the Constrained Shortest Path Problem. *Networks*, 10:293–310, 1980.

[9] S.P. Hong, H. Lee, and B.H. Park. An Efficient Multicast Routing Algorithm for Delay-Sensitive Applications with Dynamic Membership. In Proc. *INFOCOM'98*, San Francisco, CA, March 1998.

[10] J.Jaffe. Algorithms for Finding Paths with Multiple Constraints. *Networks*, 14:95–116, Spring 1984.

[11] D.H. Lorenz and A. Orda. QoS Routing in Networks with Uncertain Parameters. In Proc. *IEEE INFOCOM '98*, San Francisco, CA, March 1998.

[12] Q. Ma and P. Steenkiste. Quality-of-Service Routing for Traffic with Performance Guarantees. In Proc. *IFIP Fifth International Workshop on Quality of Service*, pages 115–126, Columbia University, New York, May 1997.

[13] H. De Neve and Van Mieghem. A Multiple Quality of Service Routing Algorithm for PNNI. In Proc. *IEEE ATM '98 Workshop*, pages 306–314, Fairfax, Virginia, May 1998.

[14] H.F. Salama, D.S. Reeves, and Y. Viniotis. A Distributed Algorithm for Delay-Constrained Unicast Routing. In Proc. *IEEE INFOCOM'97*, Japan, April 1997.

[15] H.F. Salama, D.S. Reeves, and Y. Viniotis. Evaluation of Multicast Routing Algorithms for Real-Time Communication on High-Speed Networks. *IEEE J. Select. Areas Commun.*, 15:332–346, April 1997.

[16] Q. Sun and H. Langendorfer. A New Distributed Routing Algorithm with End-to-End Delay Guarantee. In Proc. *IFIP Fifth International Workshop on Quality of Service*, Columbia University, New York, May 1997.

[17] Z. Wang and J. Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE J. Select. Areas Commun.*, 14(7):1188–1234, Sepetember 1996.

[18] B.M. Waxman. Routing of Multipoint Connections. *IEEE J. Select. Areas Commun.*, pages 1617–1622, December 1988.

[19] R. Widyono. The Design and Evaluation of Routing Algorithms for Real-Time Channels. Technical Report ICSI TR-94-024, International Computer Science Institute, U.C. Berkeley, June 1994.

A The DCCR algorithm

RoutingDCCR(G(V,E), s, d, Δ_d, Δ_c, D, C, k)
/* Each node u has k records, $(D, C, W, \pi_{nd}, \pi_{idx}, mark)$, which is stored in $ND(u, idx)$,
where π_{nd} points to the predecessor node on that path and π_{idx} points to the predecessor's record of that path.
A min-heap MH is maintained in increasing weight order, each heap item has the form (n_id, wgt, idx) */
Set $C_{best} \leftarrow \infty$, $\mathcal{P} \leftarrow nil$
InitializeSingleSource(G, s, ND, MH, k)
HeapInsert$(MH, (s, 0, 1))$ /* Searching start from s */
while $MH \neq \emptyset$
  $(u, wgt_u, idx_u) \leftarrow$ **HeapExtractMin**(MH)
  $ND(u, idx_u)$.mark = VISITED
  if $u = d$ /* found a path p */
    $C(p) \leftarrow \sum_{l \in p} c(l)$ /* trace back this new path p and compute its cost */
    $\mathcal{P} \leftarrow p \bigcup \mathcal{P}$
    if $C(p) < C_{best}$
      $C_{best} \leftarrow C(p)$, $p_{best} \leftarrow p$.
  if $u = d$ and $|\mathcal{P}| = k$ /* Tried k shortest paths */
    Return p_{best}
  for each vertex $v \in Adj[u]$ /* relaxation */
    $(W(v), D(v), C(v)) \leftarrow$ **ComputeWeight**(u, idx_u, v)
    $(idx_v, w_{max}) \leftarrow$ **FindMax**(ND, v) /* Find path to v with max weight */
    if $W(v) < w_{max}$ and path idx_v is **not** dominated
      $ND(v, idx_v) \leftarrow (D(v), C(v), W(v), u, idx_u,$ UNVISITED)
      $i \leftarrow$ **HeapSearch**(MH, v, idx_v)
      if $i \neq nil$ /* Update heap records */
        HeapReplace$(MH, i, (v, W(v), idx_v))$
      else **HeapInsert**$(MH, (v, W(v), idx_v))$

InitializeSingleSource(G, s, ND, MH, k)
for each node $u \in G$
  for $i \leftarrow 1$ to k **do**
    $ND(u, i).(D, C, W, \pi_{nd}, \pi_{idx}, mark) \leftarrow (\infty, \infty, \infty, nil, nil,$ UNVISITED)
$ND(s, 1).W = 0$

ComputeWeight(u, idx, v)
1. $D(v) \leftarrow ND(u, idx).D + d(u, v)$, $C(v) \leftarrow ND(u, idx).C + c(u, v)$
2. Compute $W(v)$ as defined in Equation (4)
3. Return $(W(v), D(v), C(v))$

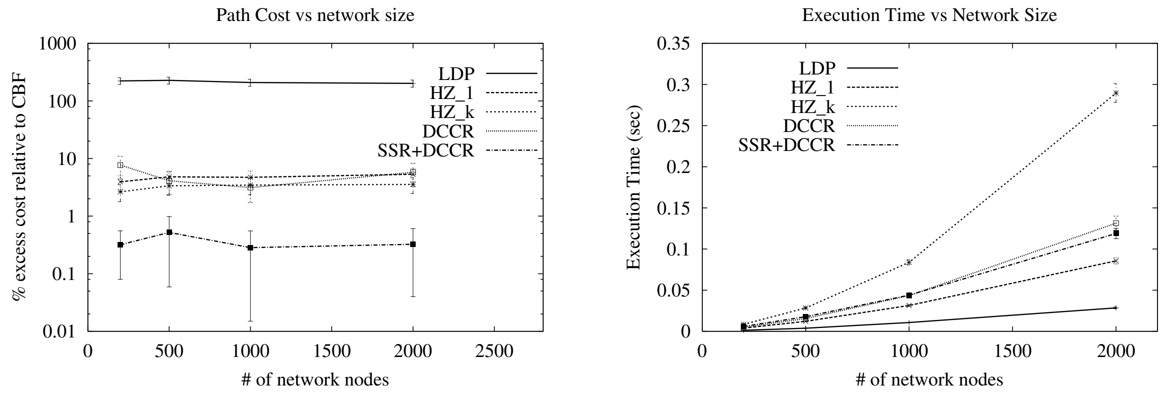
FindMax(ND, u)
1. **Return** $(idx, ND(u, idx).W)$ where idx is the index of the path with maximum weight
and $ND(u, idx).mark$ = UNVISITED

Figure 8: The DCCR algorithm

B The HZ_1 algorithm

INPUT
$G(V, E)$ = graph, s = source node,
d = destination node,
Δ = application specified delay bound,
D = link delay function,
C = link cost function.
DJK = Shortest Path Algorithm (e.g. Dijkstra's algorithm).
OUTPUT
A delay bounded path from source s to destination d.
RoutingHZ(G(V,E), s, d, Δ, D, C, **DJK**)
1. Call $\mathbf{DJK}(G, s, d, D)$ to compute the least-delay path, store it in LDP.
2. Call $\mathbf{DJK}(G, s, d, C)$ to compute the least-cost path, store it in LCP.
3. **if** $D(LDP) > \Delta$
4. **Return** $FAILED$.
5. **if** $D(LCP) <= \Delta$
6. **Return** LCP. /* LCP is a feasible path */
7. Set $\alpha \leftarrow C(LDP) - C(LCP)$, $\beta \leftarrow D(LCP) - D(LDP)$
$\gamma \leftarrow D(LCP) * C(LDP) - D(LDP) * C(LCP)$.
Compute $w(e) \leftarrow \alpha * d(e) + \beta * c(e)$ for each $e \in E$.
Call $\mathbf{DJK}(G, s, d, W)$ to compute the least weight path, store it in LWP.
8. **if** $W(LWP) = \gamma$
9. **if** $D(LWP) \leq \Delta$
10. **Return** LWP
11. **else** /* $D(LWP) > \Delta$ */
12. **Return** LDP
13. **if** $W(LWP) < \gamma$
14. **if** $D(LWP) \leq \Delta$
15. $LDP \leftarrow LWP$.
16. **else** /* $D(LWP) > \Delta$ */
17. $LCP \leftarrow LWP$.
18. Go to step 7.

Figure 9: The HZ_1 algorithm



(a) (b)

Figure 10: Path cost and execution time versus network size: average node degree=4.0, $\rho = 0.5$, $T = 10.0$, negative correlation between cost and delay

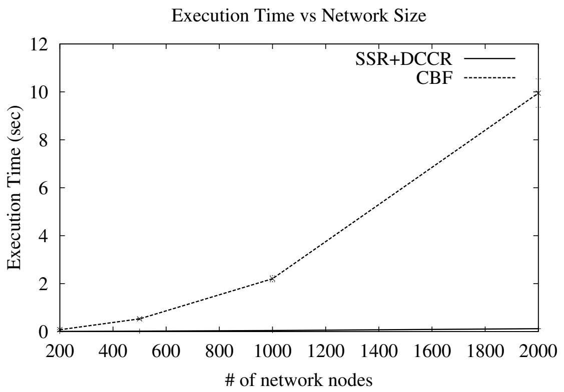
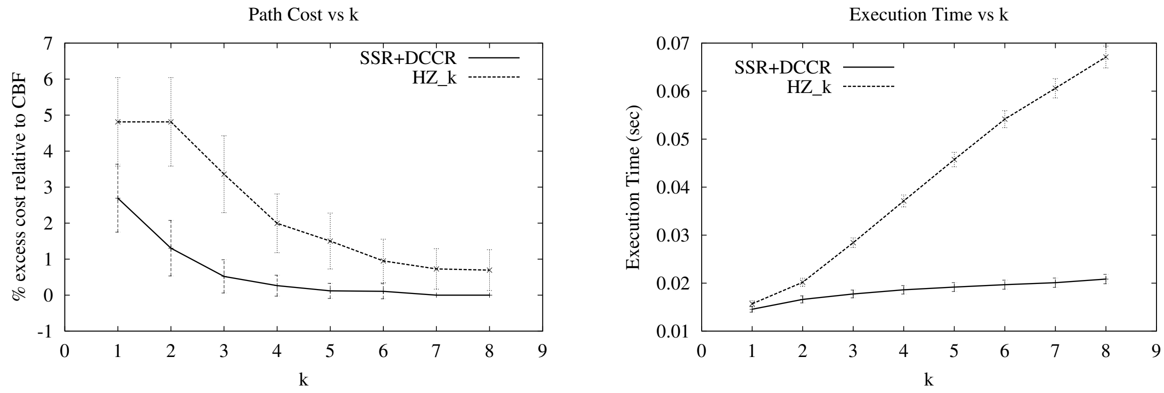


Figure 11: Execution time of CBF and SSR+DCCR, same configuration as in Figure 10



(a) (b)

Figure 12: Path cost and execution time versus k: network size = 500, average node degree = 4.0, $\rho = 0.5$, $T = 10.0$, negative correlation between cost and delay

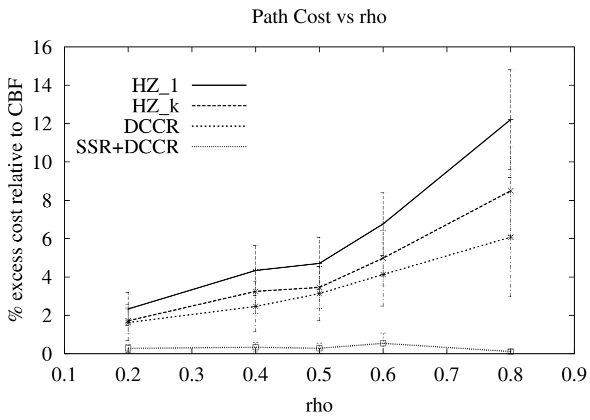


Figure 13: Path cost versus ρ: network size = 1000 nodes, average node degree = 4.0, T = 10.0, negative correlation between cost and delay

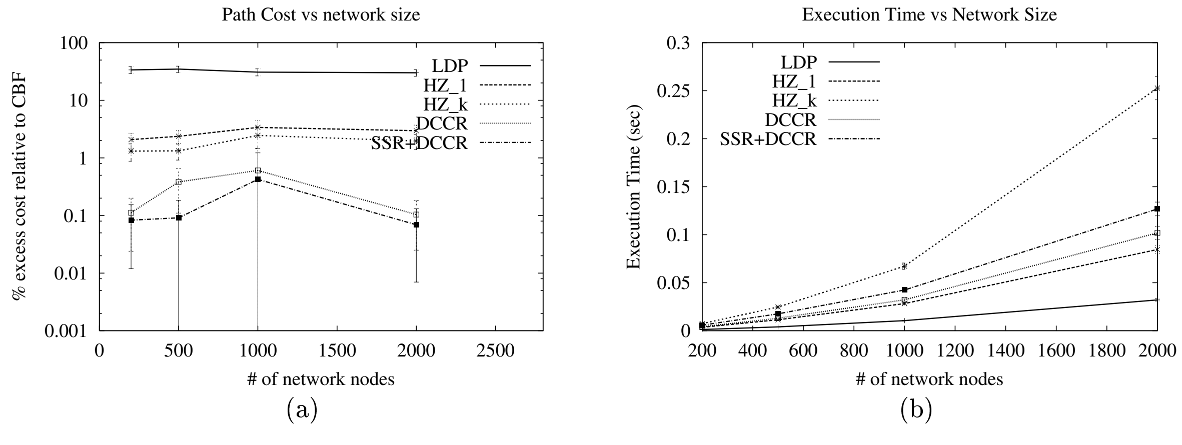


Figure 14: Path cost and execution time versus network size: average node degree=4.0, $\rho = 0.5$, $T = 10.0$, cost and delay are independent