Finding Multi-Constrained Multiple Shortest Paths

Gang Feng, *Member, IEEE* and Turgay Korkmaz, *Member, IEEE*

Abstract—Numerous algorithms have been proposed for the well-known multi-constrained shortest path (MCSP) problem, but very few have good practical performance in case of two or more constraints. In this paper, we propose a new Lagrangian relaxation algorithm to solve a generalized version of the MCSP problem where we search for *multiple* shortest paths subject to multiple constraints. As in some related work, our algorithm first identifies the lower and upper bounds, and then tries to close the gap with a path enumeration procedure. However, our algorithm is based on the recognition that the Lagrange multipliers found by existing methods usually do not give the best search direction for minimizing path enumerations even though they can provide near-optimized lower bounds. We provide a solution to meet both of these goals, and incorporate feasibility checks into a state-of-the-art K-shortest paths algorithm to further reduce path enumerations. Through experiments on various networks including the most challenging benchmark instances, we show that our algorithm can solve a significantly larger number of instances to optimality with less computational cost, often by one or two orders of magnitude, when compared with the best known algorithm in the literature.

Index Terms—Multi-constrained shortest path, multi-constrained path, Lagrangian relaxation, *K* shortest paths

1 Introduction

Given a directed graph with each edge associated with a cost and a set of additive weights, the multi-constrained shortest path (MCSP) problem is to find the least-cost path between a pair of nodes such that the sum of each weight is no more than a specified limit. This problem, known to be NP-hard [1], arises in many practical applications including quality-of-service (QoS) routing in communication networks, wire routing on circuit boards, aircraft routing, etc. Accordingly, it has been extensively studied in the literature while being called with different names such as the resource constrained shortest path problem.

The MCSP problem with a single constraint has been well studied, as evidenced by the availability of numerous heuristic algorithms [2], [3], approximation schemes [4] and exact approaches [5], [6], [7]. In contrast, the algorithms solving this problem subject to two or more constraints are quite limited or have poor practical performances. Xue et al. [8] proposed an approximation algorithm, but its average running time seems to be high. Kuipers and Mieghem [9] studied the hardness of this problem for communication networks with certain characteristics. They also summarized in [10] several techniques such as look-ahead and path dominance for developing exact algorithms. Liu and Ramakrishnan [11] proposed an A*prune algorithm, which is essentially a combination of A* search [12] with the look-ahead technique. Li et al. [13] proposed an iterative deepening search method using the *hop count* as the cost function.

A different thread of research is on the Lagrangian relaxation (LRE) approach. The basic idea is to first solve a Lagrangian dual (LD) problem to find a lower bound for the optimal path. With the upper bound given by any feasible path, a successive procedure is then used to close the gap between the bounds. The pioneering work on this approach was due to Handler and Zang [14]. They solved the LD problem with a hull approach [15], and used Yen's K-shortest paths (KSP) algorithm [16] to close the gap. Beasley and Christofitdes [18] were the first to propose a LRE method for the multi-constrained case. They used a subgradient procedure to solve the LD problem, and a tree search procedure for closing the gap. Ziegelmann [15] pointed out that the tree search method is much less efficient than the label setting algorithms [19] and the label correcting approaches [20].

For problems on large networks, it has been recognized that using a preprocessing procedure to reduce the problem size is very helpful. This recognition was originally brought up in [19], but the most successful preprocessing procedure was developed by Dumitrescu and Boland [21]. They reported that their modified label setting algorithm (MLSA) making use of preprocessing is as competitive as the algorithm proposed by Beasley and Christofides [18]. Dumitrescu and Boland did not show how MLSA [21] performs with multi-constrained problems. Zhu and Wilhelm [22] proposed an improved version for multi-constrained case and investigated its performance on small networks. Carlyle et al. [23] have done similar work with performance evaluations on much larger networks. The algorithm proposed by Carlyle et al. is called LRE-PA, which stands for Lagrangian relaxation with preprocessing and aggregation. LRE-PA uses a repeated bisection method [24] to solve the LD

- *G. Feng is with Electrical Engineering Department, University of Wisconsin, Platteville, WI 53818. E-mail: fengg@uwplatt.edu.*
- *T. Korkmaz is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX. E-mail: korkmaz@cs.utsa.edu.*

Manuscript received 27 May 2014; accepted 1 Oct. 2014. Date of publication 2 Nov. 2014; date of current version 12 Aug. 2015.
Recommended for acceptance by S. Rajasekaran.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2366762

problem, and closes the gap using an improved A*prune algorithm, which enforces additional aggregated constraints. Experimental results of hard benchmark instances [23] demonstrated that overall LRE-PA performs significantly better than MLSA.

Most recently, Lozano and Medaglia [7] suggested combining the path dominance check with A*prune. Although such an algorithm could work well for single-constraint problems, it would have to keep a significantly large number of labels at each node in case of two or more constraints, requiring considerable time to check path dominance (as mentioned in [15] and [23]).

In this paper, we propose a new LRE algorithm to further improve the practical performance over the existing MCSP solutions and to solve the generalized version of MCSP problem where we search for *multiple* shortest paths subject to multiple constraints. This generalized version of MCSP problem would arise in many applications. For example, it may appear as a subproblem in other problems such as the constrained minimum Steiner tree problem [25]. For QoS routing in communication networks, having multiple paths satisfying the given QoS requirements would ensure a minimal interruption of service in case the currently used path fails. In fact, the A*prune algorithm [11] was intended for such an application. More recently Shi [26] proposed a solution to this problem in the context of transportation systems. Both in [11] and [26], a parameter K has been used to specify the number of paths to be found. Our algorithm will use a different parameter J to avoid confusion with the parameter K in the KSP algorithm.

Our algorithm follows the fundamental idea of Handler and Zang [14] by using a KSP procedure to close the gap. Unlike their algorithm which only works for the *single-constraint* case, our algorithm is not limited by the number of constraints and can provide significantly better performance than LRE-PA [23] which, from a practical point of view, is the best algorithm to date. Using KSP for gap-closing has been criticized [6] as it causes the proliferation of paths. We show that this was due to the lack of an algorithm to find the best Lagrange multipliers. While the existing LD algorithms can provide near-optimal lower bounds, they return the Lagrange multipliers that usually do not give the best search direction for a KSP based gap-closing algorithm. Since finding such multipliers is different from the conventional LD problem, we formulate our problem with a new objective function. We then provide a procedure to find the Lagrange multipliers that can significantly reduce the number of paths to be enumerated while optimizing the lower bound. This is our main contribution. Our second contribution is to show how to incorporate feasibility checks into a state-of-the-art KSP algorithm [27] to further reduce the number of paths to be stored.

We provide two versions of the proposed algorithm: a basic version that solely relies on the techniques proposed in this paper and an enhanced version that is combined with the preprocessing and constraint aggregation techniques used in related studies. Both versions have been tested with the most challenging benchmark instances as well as random instances on large grid and random networks. Experimental results on the hard MCSP instances indicate that our algorithm is able to solve a significantly larger number of instances to optimality while having much less computational cost, often by one or two orders of magnitude, when compared with LRE-PA. In the case of finding multiple solutions, our algorithm also demonstrates a superior performance.

The remainder of this paper is organized as follows. Section 2 provides the formal definitions of the related problems. In Section 3, we describe an algorithm to find J feasible paths. The new algorithm for the generalized MCSP problem is elaborated in Section 4. The experimental results for the MCSP problem are presented in Section 5, and the results for finding multiple solutions are given in Section 6. Section 7 concludes the paper.

2 Preliminaries

2.1 Definitions and Notation

A network is modeled by a directed graph $G(V, E)$, where V is the set of nodes, E is the set of edges, $n = |V|$ and $m = |E|$. The nodes in V are numbered from 1 to n. An edge e from node i to node j is denoted by $e = (i, j)$, where $i = tail(e)$ and $j = head(e)$ are called the tail and head of edge e. Each edge $(i, j) \in E$ has a nonnegative cost c_{ij} and R nonnegative weights $w^r_{ij}, r \in [1..R]$.

A *path* p in G is a tuple $(e_1, e_2, \ldots, e_{|p|})$ of edges in E such that $head(e_k) = tail(e_{k+1}), \forall k = 1, \ldots, |p| - 1$. If a path does not visit a node more than once, it is called a *simple* path. We denote by $c(p)$ and $w_r(p)$, respectively, the cost and the rth weight of a path p, i.e., $c(p) = \sum_{e \in p} c_e$ and $w_r(p) = \sum_{e \in p} w^r_e, r \in [1..R]$.

Definition 1. *Given a node $s \in V$, a node $t \in V$ and R nonnegative* weight limits *$U_r, r \in [1..R]$, a simple s-t path p is* feasible *if it satisfies R* weight constraints

$$w_r(p) \leq U_r, \forall r \in [1..R]. \tag{1}$$

The MCSP problem is to find a least-cost feasible path from s to t.

Definition 2. *Given a node $s \in V$, a node $t \in V$, R weight limits U_r, $r \in [1..R]$, and an integer $J \geq 1$, the J-feasible-paths (J-FP) problem is to find a subset $P_1 \subseteq P$, where P is the set of all feasible paths and $|P_1| = J$. The J-MCSP problem is to find a subset $P_2 \subseteq P$ such that $|P_2| = J$, and $c(p) \leq c(q)$, $\forall p \in P_2, q \in P \backslash P_2$.*

Throughout the paper, we use KSP$(\hat{l}, K, l_{max})$ to denote a KSP procedure that uses an edge length function $\hat{l} : E \rightarrow \mathbb{R}^+_0$, and finds either the K shortest simple paths, or all the simple paths with the length no more than $l_{max} \in \mathbb{R}^+$, whichever happens first. In addition, we use σ_i to denote the least cost path from node i to node t, and for each $r \in [1..R]$, use τ^r_i to denote the shortest path with respect to (w.r.t.) weight r from node i to node t.

2.2 Integer Programming Formulation

The MCSP problem can be formulated as an integer linear programming (ILP) problem with a set of zero-one decision variables x_{ij}'s [28]. For each $(i, j) \in E$, $x_{ij} = 1$ if edge (i, j) is on a path and $x_{ij} = 0$ otherwise. Let $\mathbf{x} = (x_{ij} | (i, j) \in E)$ be

an m-dimensional column vector composed of x_{ij}'s. Then, a simple s-t path can be mapped to a vector $\mathbf{x}$ satisfying the following equations:

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = 0,\ \forall i \in V\backslash\{s,t\} \tag{2}$$

$$\sum_{j:(s,j)\in E} x_{sj} = 1 \tag{3}$$

$$\sum_{i:(i,t)\in E} x_{it} = 1 \tag{4}$$

$$x_{ij} = 0 \text{ or } 1,\ \forall (i,j) \in E. \tag{5}$$

Eqs. (3) and (4) ensure that one edge leaving s and one edge entering t are on the path. Eq. (2) guarantees that, at any node $i \in V$, an outgoing edge must be on the path if an incoming edge is on the path.

Denote by X the set of vectors $\mathbf{x}$ satisfying Eqs. (2)-(5), and let $\mathbf{c} = (c_{ij}|(i,j) \in E)$ be an m-dimensional row vector of costs, $\mathbf{w}_r = (w^r_{ij}|(i,j) \in E)$ an m-dimensional row vector composed of the rth edge weights, and $\mathbf{U} = (U_r|r = 1,\ldots,R)$ a R-dimensional column vector of the weight limits. Finally, let $\mathbf{W}$ be a $R \times m$ matrix with $\mathbf{w}_r$ being the rth row. Then, the ILP formulation of the MCSP problem can be given as follows:

$$C^* = \min_{\mathbf{x}\in X} \mathbf{cx},$$

subject to $\mathbf{Wx} \leq \mathbf{U}$.

Note that $C^* = \mathbf{cx}^*$ is the cost of an optimal path corresponding to $\mathbf{x}^*$. Let $\lambda = (\lambda_r|r \in [1..R])$ be a row vector of Lagrangian multipliers. Then, for a given $\lambda \geq \mathbf{0}$, a lower bound of C^* can be found by solving the following LRE problem [15]:

$$\mathcal{L}(\lambda) = \min_{\mathbf{x}\in X}(\mathbf{c} + \lambda\mathbf{W})\mathbf{x} - \lambda\mathbf{U}. \tag{6}$$

Since the quantity $\lambda\mathbf{U}$ is a fixed value, the LRE problem can be solved by finding the shortest path w.r.t. the Lagrangianized edge length function

$$l_{ij} = c_{ij} + \sum_{r=1}^{R} \lambda_r w^r_{ij}, \forall (i,j) \in E, \tag{7}$$

followed by computing the lower bound with Eq. (6).

A solution to the LRE problem that results in a larger lower bound often requires less time during the gap-closing stage. Therefore, it is beneficial to maximize the lower bound by solving the LD problem

$$C^*_l = \mathcal{L}(\lambda^*) = \max_{\lambda\geq\mathbf{0}} \mathcal{L}(\lambda).$$

Note that if a feasible path p is found while solving the LD problem, the upper bound of C^* is then identified since $C^* \leq c(p)$ must hold.

3 Finding J Feasible Paths

In this section we describe an algorithm called FindFP to solve the J-FP problem, which is a critical step for solving the J-MCSP problem. The J-FP problem in the single-constraint case ($R = 1$) can be simply solved with a KSP algorithm by setting $K = J$. However, in the case with $R \geq 2$, it becomes NP-hard [1]. So, FindFP will concentrate on that case.

FindFP($G(V,E), s, t, J, w_r, U_r, r \in [1..R]$)
1 $(\lambda, solved, P_1) \leftarrow$ FPDT($G, s, t, J, w_r, U_r, r \in [1..R]$)
2 **if** $solved = TRUE$ **then** go to Step 4
3 $(solved, P_1) \leftarrow$ FPPE($G, s, t, J, w_r, U_r, r \in [1..R], \lambda$)
4 return $(solved, P_1)$

Fig. 1. The FindFP algorithm.

As shown in Fig. 1, FindFP contains two main procedures called FPDT and FPPE. The primary goal of FPDT is to determine a Lagrange multiplier vector λ. While searching for such a λ, FPDT could successfully find J feasible paths. Therefore, FPDT returns three parameters: λ, $solved$, and P_1. If the boolean variable $solved$ is set to $TRUE$, P_1 would contain the set of feasible paths. Otherwise, the λ is passed to the FPPE procedure, which would calculate a Lagrangianized weight defined as

$$\tilde{l}_{ij} = \sum_{r=1}^{R} \lambda_r w^r_{ij}, \forall (i,j) \in E, \tag{8}$$

which is Eq. (7) without the cost. FPPE will then enumerate the shortest paths w.r.t. $\tilde{l}$ until J feasible paths are found or a termination condition is met. FPPE returns two parameters, $solved$ and P_1; Similarly, $solved$ indicates whether the problem has been successfully solved, and if $solved$ is $TRUE$, P_1 would contain the set of feasible paths. In the following, we elaborate the two procedures.

3.1 FPDT: J-FP Direction-Tuning Procedure

Since the λ returned by FPDT will eventually be used for path enumeration, we are interested in finding a λ that minimizes the number of paths to be enumerated. FPDT is precisely designed for such a goal. Before presenting the pseudo-code, we explain the basic principle as below.

In the polyhedral theory [29], a path p is a point $(w_1(p), w_2(p), \ldots, w_R(p))$ in $\mathbb{R}^R$. The R inequalities in Eq. (1) defines a polytope in $\mathbb{R}^R$, which can be called the feasibility region since a point inside the polytope is corresponding to a feasible path. When $R > 3$, enumerating the sequence of paths w.r.t. $\tilde{l}$ (i.e., Eq. (8)) can be viewed as a process of moving a hyperplane along the direction defined by λ, and returning the corresponding path when a point is hit. When $R = 3$, the "hyperplane" becomes a plane, and a line when $R = 2$. In the worst case, the hyperplane has to be moved until the whole feasibility region is scanned. An example for $R = 2$ is shown in Fig. 2, where the rectangle denotes the feasibility region and the hatched triangle area needs to be scanned in the worst case. Clearly, a specific λ value determines a certain area in $\mathbb{R}^R$ that has to be swept through. Our goal is to find such a λ that would allow the algorithm to encounter the minimum number of infeasible paths before J feasible paths are found or the entire feasibility region is swept.

Given a λ and an integer $K > 1$, let P_λ be the set of K shortest paths w.r.t. $\tilde{l}$. If K is sufficiently large, we would know precisely how many paths have to be enumerated before the set of feasible paths are found. On the other hand, with a smaller value for K, we could still extract

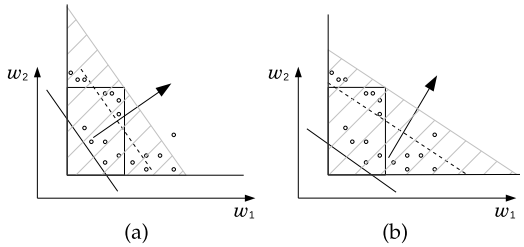


Fig. 2. An example for feasibility-region scanning.

some useful information about the probability that a path is feasible. To compute this probability, let us first introduce the following definition.

Definition 3. *For any $r \in [1..R]$, a path p is said to be* constraint-r feasible *if $w_r(p) \leq U_r$. Note that a path is feasible if it is constrain-r feasible for all $r \in [1..R]$.*

Assuming P_λ has been found using a KSP algorithm, we can calculate h_r, which is defined as *the number of paths in P_λ that are constraint-r feasible*. Then, for a path p found while enumerating along the λ direction, the ratio $\xi_r = h_r/K$ can be interpreted as the probability that p is constraint-r feasible. Consequently, we can compute $\rho = \prod_{r=1}^{R} \xi_r$ as the probability that p is feasible. These concepts can be illustrated using the same example in Fig. 2, where we assume each small circle denotes a path. Assuming $K = 10$ and the search follows the direction shown in Fig. 2a, we would have $h_1 = 8$, $h_2 = 7$ and $\rho = 0.56$. In the case with Fig. 2b, they would be $h_1 = 5$, $h_2 = 10$ and $\rho = 0.5$.

The KSP algorithm returns at most K paths and the probability that a returned path is feasible is given by ρ. Accordingly, using the well-known binomial distribution formulas [30], we can compute the expected number of feasible paths as $K\rho$. Our goal of minimizing the number of infeasible paths can be achieved by maximizing $K\rho$, which can only be done by maximizing ρ.

To maximize ρ, different solutions could exist. We consider the following *normalized least-mean-square-error* objective function:

$$\mathcal{E}(\lambda^*) = \min_{\lambda \geq \mathbf{0}} \frac{1}{R} \sum_{r=1}^{R} \left(\frac{h_r - \bar{h}}{\bar{h}} \right)^2, \tag{9}$$

where $\bar{h}$ ($\leq K$) is a preset integral value.

Maximizing ρ is equivalent to minimize a function $f = 1 - \rho = 1 - \prod_{r=1}^{R} \xi_r$. Note f is a concave function w.r.t. each ξ_r, reaching the minimum when each $\xi_r = 1$. When $\bar{h} = K$, the summation in Eq. (9) essentially becomes $\sum_{r=1}^{R} (-1(1-\xi_r))^2 = \sum_{r=1}^{R} (1-\xi_r)^2$ and it carries the same property as function f. So with $\bar{h} = K$, Eq. (9) ends up maximizing ρ.

The reason for selecting the formulation in Eq. (9) is because this objective function is symmetric w.r.t. the h_r's, and the Purkiss Principle [31] guarantees that the solution to this problem is in favor of a λ that makes each h_r close to $\bar{h}$. The ideal solution would be obtained with $h_r = \bar{h}$. Since we set $\bar{h} = K$, it will force each h_r close to K, resulting in a maximized ρ.

FPDT($G(V,E), s, t, J, w_r, U_r, r \in [1..R]$)
0 Set $\lambda_1 = 1$, $\lambda_r = \frac{U_1 - w_1(\tau_s^1)}{U_r - w_r(\tau_s^r)+1}, r = 2, \cdots, R$
Set $\lambda^* = \lambda$, $K = \max\{1000, 2J\}$, $\mathcal{E}^* = \infty$, $\mu = 2$
Set $y = 0$, $\alpha = 0.05$, $\tilde{\alpha} = 0.1$, $solved = FALSE$, $P_1 = \varnothing$
1 Use Eq. (8) to compute $\tilde{l}$
2 Let $P_\lambda \leftarrow$ KSP($\tilde{l}, K, \lambda\mathbf{U}$)
3 **if** $|P_\lambda| < K$ **or** P_λ has J or more feasible paths **then**
set $solved = TRUE$, $P_1 \leftarrow$ {feasible paths}, go to Step 14
4 Calculate $h_r, \forall r \in [1..R]$
5 Calculate $\mathcal{E}$ by Eq. (9)
6 **if** $\mathcal{E} < \mathcal{E}^*$ **then**
7 Set $\mathcal{E}^* = \mathcal{E}$, $\lambda^* = \lambda$, $h_r^* = h_r, \forall r \in [1..R]$
8 **else**
9 **if** $\mathcal{E} - \mathcal{E}^* > \tilde{\alpha}$ **then**
10 Set $\mu = \mu/2$, $\lambda = \lambda^*$, $h_r = h_r^*, \forall r \in [1..R]$
11 **if** $\mathcal{E} < \alpha$ **then** go to Step 14
12 Update λ by Eq. (10) with $\bar{h} = K$
13 $y = y + 1$; **if** $y < 20$ **then** go to Step 1
14 return $(\lambda^*, solved, P_1)$

Fig. 3. The FPDT procedure.

The problem defined by Eq. (9) can be solved by an iterative procedure. Following a steep gradient descent technique, λ can be updated iteratively, starting from a certain initial value, with the following equation:

$$\lambda_r^+ = \lambda_r - \lambda_r \cdot \mu \frac{h_r - \bar{h}}{\bar{h}}, \forall r \in [1..R], \tag{10}$$

where λ^+ denotes the value in the next iteration and μ is a parameter that controls the aggressiveness of search. We now formally present the FPDT procedure in Fig. 3, and explain some related issues below.

When FPDT calls the KSP procedure, the maximum path length is set to $\lambda\mathbf{U}$ based on the following lemma.

Lemma 1. *Let p denote the current shortest path while enumerating the shortest paths w.r.t. $\tilde{l}$. If $\tilde{l}(p) > \lambda\mathbf{U}$, then p is infeasible.*

Proof. $\tilde{l}(p) > \lambda\mathbf{U}$ means $\sum_{r=1}^{R} \lambda_r (w_r(p) - U_r) > 0$. Thus, there is at least one r such that $w_r(p) > U_r$. □

We choose $K = \max\{1000, 2J\}$ for several reasons. With at least 1,000 paths enumerated, we would have a relatively accurate estimation on the quality of λ. It also allows the detection of trivial instances. If $|P_\lambda| < K$, then there are fewer than K simple paths from s to t, and we can simply pick J feasible paths from P_λ (or all feasible paths from P_λ if the number of feasible paths is less than J). On the other hand, if $|P_\lambda| = K$, it is still possible to find J feasible paths from P_λ because $K \geq 2J$. In either case, FPDT simply returns the set of feasible paths. Since enumerating 1,000 paths could pose a computation burden for some KSP algorithm [16], [17], it is worth mentioning that setting K to a fixed smaller values (e.g., $K = 100$) would not significantly affect the performance of our algorithm (as will be verified through experiments). Moreover, recent research efforts have produced several fast KSP algorithms [27], [34], including some Yen's variants [32].

The initial value of λ (see Step 0) is chosen in hope for a low initial $\mathcal{E}$ value in case the paths are uniformly distributed in the feasibility region. λ^* and $\mathcal{E}^*$ are used to keep the best λ and the best objective value found so far. During a certain iteration, FPDT compares the current objective value

$\mathcal{E}$ with $\mathcal{E}^*$. If $\mathcal{E}$ is smaller, it updates both λ^* and $\mathcal{E}^*$. Otherwise, the difference between $\mathcal{E}$ and $\mathcal{E}^*$ is checked. If the difference is greater than a "cushion" value $\tilde{\alpha} = 0.1$, μ is reduced by half. Then the search starts over from the current best solution but with a finer stepsize. Otherwise, the search continues with the current μ. Using a cushion value is quite critical as it allows the algorithm to search in a larger region of the solution space, and has been used in other optimization areas such as simulated annealing [33].

With FPDT, the condition $\lambda \geq \mathbf{0}$ in Eq. (9) is guaranteed as stated in the following lemma.

Lemma 2. $\lambda \geq \mathbf{0}$ *always holds before FPDT ends.*

Proof. Following Eq. (10), λ_r is adjusted in each iteration by a factor $|\mu(h_r - \bar{h})/\bar{h}|$. Since $\mu \in (0, 2]$, $\bar{h} = K$, and $h_r \leq K, \forall r \in [1..R]$, this lemma is proved. □

Setting $\mu = 2$ initially seems to work very well, although a larger initial value would not invalidate Lemma 2.

FPDT returns to FindFP if the objective value is less than $\alpha = 0.05$ or a maximum number of 20 times have been repeated. These threshold values as well as the cushion value $\tilde{\alpha}$ were chosen based on experimenting on relatively large networks (e.g., 200×200 grids) even though they follow common-sense guesses. It is important to point out that, unlike existing algorithms for the LD problem, the Eq. (10) used for updating λ does not rely on edge weights. This explains why these parameters were easily determined. In fact, if these parameters were chosen from certain intervals (e.g., ± 50 percent of the values specified in the procedure), our algorithm would work equally well with some reasonable trade-offs between the running time and the performance.

Lemma 3. *The worst case time complexity of FPDT is* $O(Jn(m + n \log \log n))$.

Proof. If Gotthilf's algorithm [17] is used, FPDT takes $O(Kn(m + n \log \log n))$ worst-case time in each iteration to rank K shortest paths. Since $K = \max\{1000, 2J\}$ and FPDT repeats a fixed number of times in the worst case, we have the result claimed in this lemma, although a large constant is hidden if $J \ll 1000$. □

3.2 FPPE: J-FP Path-Enumeration Procedure

If FPDT fails to find J feasible paths, FindFP will invoke the FPPE procedure. FPPE basically keeps enumerating the shortest paths w.r.t. Eq. (8) with the λ found by FPDT until J feasible paths are found or the whole feasibility region is scanned. One can simply use KSP here with a very large K until J feasible paths are found. Actually, the enumeration would be also stopped when the current shortest path q has a length $\tilde{l}(q) > \lambda \mathbf{U}$, because all successive paths would be infeasible (Lemma 1). Although an implementation following this simple idea could work on small networks, it might require an excessive amount of memory space when dealing with large-network instances. Accordingly, we consider a more sophisticated path-enumeration algorithm proposed in [27] and extend it by incorporating a feasibility check on each weight constraint to avoid storing too many paths.

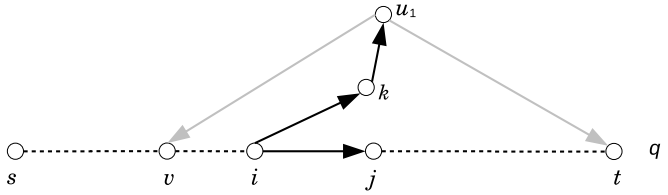


Fig. 4. A path deviating from q at node i.

FPPE starts with the following initialization:

i. Compute $\tilde{l}_{ij}, \forall (i,j) \in E$ by following Eq. (8).
ii. Compute the reverse shortest path tree T rooted at t w.r.t. $\tilde{l}$. Let π_i denote the path on T from node i to node t, $\forall i \in V$.
iii. Compute the *reduced cost function* [34], $f_{ij} = \tilde{l}_{ij} + \tilde{l}(\pi_j) - \tilde{l}(\pi_i), \forall (i,j) \in E$.
iv. For each $i \in V$, sort all outgoing edges of i in increasing order of f. Denote by $next(i,j)$ the outgoing edge of i that follows (i,j) in the sorted order.
v. Let the candidate path queue $Q = \{\pi_s\}$ (containing only the shortest path).

Following the initialization, FPPE enters into an iterative procedure. Suppose at a certain iteration, q is the shortest path in Q. FPPE first removes q from Q and then adds a set of candidate paths that deviates from q. Assuming edge $(i,j) \in q$, a candidate path p that deviates at node i through an edge (i,k) can be written as $p = S_q(s,i) \bowtie (i,k) \bowtie \pi_k$, where $S_q(s,i)$ is the subpath on q from node s to node i, and $\bowtie$ is a concatenation operator. Note that $S_q(s,i)$ can be extracted from q, and π_k has been precomputed. Therefore, p can be stored in Q as a two-tuple $\langle q, (i,k) \rangle$, which takes only $O(1)$ space. On the other hand, a node in π_k could have appeared in $S_q(s,i)$, and in such case p is a non-simple path. In the example given in Fig. 4, if $\pi_k = ((k,u_1),(u_1,t))$, p is a simple path, but p would be non-simple if $\pi_k = ((k,u_1), (u_1,v)) \bowtie S_q(v,t)$, where $v \in S_q(s,i)$.

As Q contains both simple and non-simple paths, the shortest path q removed from Q could be non-simple. Since we are only interested in ranking simple paths, there is only a subset of the edges on q that can be deviated from to generate candidate paths.

Definition 4. *Given a path $q = \langle x, (i,k) \rangle$ in Q, q can be written as the concatenation of three subpaths: $S_x(s,i) \bowtie S_q(i,v) \bowtie S_q(v,t)$ such that either $v = t$ or v is the first node along subpath $S_q(i,t)$ that appears in $S_x(s,i)$. Then, the edges in $S_q(i,v)$ are called the* deviation-qualifying *edges of q.*

Note that initially $Q = \{\pi_s\}$, and π_s can be represented as a two-tuple $\langle NULL, (s,s) \rangle$. This means any path in Q can be stored as a two-tuple taking $O(1)$ space, allowing all paths in Q to be stored using a linked-list data structure [27] to achieve a high space efficiency.

We now formally present FPPE in Fig. 5. During the initialization, a parameter M is used to specify the maximum number of iterations. In each iteration (Steps 1-14), the current shortest path q is first removed from Q. The procedure then attempts to find a candidate path deviating from each deviation-qualifying edge. Step 10 checks the length of a potential candidate path deviating through

FPPE($G(V,E), s, t, J, w_r, U_r, r \in [1..R], \lambda$)
0 Compute $\tilde{l}_{ij}, f_{ij}, \forall (i,j) \in E$, and $\pi_i, \forall i \in V$
Sort the outgoing edges of each node in increasing order of f
Set M to the maximum number of iterations
Set $Q = \{\pi_s\}$, $P_1 = \varnothing$, $y = 0$, $solved = FALSE$
1 **if** $Q = \varnothing$ **then** set $solved = TRUE$ and go to Step 16
else remove the shortest path q from Q
2 **if** q is simple and feasible **then**
3 add q to P_1;
if $|P_1| = J$ **then** set $solved = TRUE$ and go to Step 16
4 Set $j = 1$ and let $(e_1, e_2, \cdots, e_\ell)$ be the subpath containing the deviation-qualifying edges of q
5 **while** $j \leq \ell$ do
6 Let $(u,a) = e_j$
7 **if** $\exists r \in [1..R]$, $w_r(S_q(s,u)) + w_r(\tau_u^r) > U_r$ **then** break the **while** loop
8 **if** $\exists next(u,a)$ **then**
9 Set $(u,a) = next(u,a)$
10 **if** $\tilde{l}(S_q(s,u) \bowtie (u,a) \bowtie \pi_a) > \lambda \mathbf{U}$ **then** go to Step 14
11 **if** $a \in S_q(s,u)$ **then** go to Step 8
12 **if** $\exists r \in [1..R]$, $w_r(S_q(s,u)) + w_{ua}^r + w_r(\tau_a^r) > U_r$ **then** go to Step 8
13 Add path $\langle q, (u,a) \rangle$ to Q
14 $j = j + 1$
15 $y = y + 1$; **if** $y < M$ **then** go to Step 1
16 return $(solved, P_1)$

Fig. 5. The FPPE procedure.

edge (u,a). If this length is greater than $\lambda \mathbf{U}$, it stops checking other outgoing edges of node u (by branching to Step 14) because a path deviating through any successive edge would also violate this condition as the edges were sorted in increasing order of the reduced cost. This guarantees that any candidate path admitted to Q has a length no more than $\lambda \mathbf{U}$. Step 11 ensures the edge through which the path deviates does not immediately go back to a preceding node. Steps 7 and 12 perform feasibility checks on the weight constraints.

We now prove the the correctness of FPPE. Without the feasibility checks on Steps 7 and 12, FPPE follows the same process as the algorithm in [27], which guarantees that a simple path with a length smaller than or equal to $\lambda \mathbf{U}$ is not excluded from Q. Therefore, we need to just prove that the feasibility checks do not reject any feasible path. For this, we first give a recursive definition of *descendant*.

Definition 5. *If p is a path deviating from q, p is called a descendant of q, and so is a path deviating from any descendant of q.*

Lemma 4. *If a path p is excluded from Q due to a feasibility check on Step 7 in FPPE, then all descendants of p found without feasibility checks are infeasible.*

Proof. Suppose FPPE is executed with Steps 7 and 12 skipped, and p is a path deviating from q at node u, meaning p can be written as $p = S_q(s,u) \bowtie S_p(u,t)$. Let Z contain the set of descendants of p. Then, any path in Z would contain $S_q(s,u)$ as a subpath. So, a path $z \in Z$ can be written as $z = S_q(s,u) \bowtie S_z(u,t)$. Because for any $r \in [1..R]$, τ_u^r is the least-w_r path from u to t, $w_r(S_z(u,t)) \geq w_r(\tau_u^r)$ must hold, implying $w_r(z) = w_r(S_q(s,u)) + w_r(S_z(u,t)) \geq w_r(S_q(s,u)) + w_r(\tau_u^r)$. Hence, if p fails a feasibility check on Step 7 for a certain $r \in [1..R]$, we would have $w_r(z) > U_r$, and all paths in Z would be infeasible. □

DT($G(V,E), s, t, J, c, w_r, U_r, r \in [1..R], \delta$)
0 $P_2 = \varnothing$
1 $(solved, P_1) \leftarrow$ FindFP($G, s, t, J, w_r, U_r, r \in [1..R]$)
2 **if** $solved = FALSE$ **then** go to Step 7
3 Let $\dot{p} = \arg\max_{p \in P_1} c(p)$
5 $(\lambda, C_u) \leftarrow$ MCSPDTHi($G, s, t, J, c, w_r, U_r, r \in [1..R], \dot{p}$)
6 $(solved, P_2) \leftarrow$ MCSPPE($G, s, t, J, w_r, U_r, r \in [1..R], \lambda, C_u, \delta$)
7 **if** $solved = TRUE$ **then** return P_2 **else** indicate a failure

Fig. 6. The DT algorithm.

If at a certain iteration a feasibility check on Step 7 failed with a certain $e_j = (u,a) \in q$, then a path p that includes subpath $S_q(s,u)$ would be infeasible, and so would a path that includes subpath $S_q(s,w)$, where $w = tail(e_k), \forall k = j+1, \ldots, \ell$. Due to Lemma 4, all descendants of these paths would be infeasible. Therefore in such case the procedure can stop the current while loop and start the next iteration (Step 7). Step 12 extends the checks on Step 7 by one more edge. The difference is, if a check on Step 12 failed, it needs to go back to Step 8 because there could be another outgoing edge of u that leads to a feasible path.

The above analysis allows us to claim the correctness of FPPE with the following theorem.

Theorem 1. *As the value of M goes to ∞, FPPE will find either J feasible paths or all feasible paths.*

Proof. Because Steps 7 and 12 only exclude infeasible paths from Q, any feasible path can be admitted to Q before M iterations are repeated. With M set to a significant large number, FPPE will return to FindFP (through Step 16) either when Q is empty (see Step 1) or when J feasible paths have been found (see Step 3). If during a certain iteration Q becomes empty, it means there are fewer than J feasible paths from s to t, which have been stored in P_1. □

Practically M can not be too large, so FPPE could fail to find J feasible paths after repeating M iterations. In such case, FPPE returns to FindFP with $solved = FALSE$.

Lemma 5. *The worst case time complexity of FPPE is $O(Mn^2)$ and the space complexity is $O(m + Mn)$.*

Proof. Following the analysis in [27], each iteration takes $O(n^2)$ worst case time. The space complexity is due to the fact that in each iteration no more than n paths are added to the Q and each path takes $O(1)$ space. □

4 A New Algorithm for J-MCSP

The J-MCSP problem needs to find a set P_2 of the least-cost J feasible paths. Our fundamental idea can be described as follows. Suppose $\hat{p} = \text{argmax}_{p \in P_2} c(p)$ and $\check{p} = \text{argmin}_{p \in P_2} c(p)$ are, respectively, the most expensive path and the least-cost path in P_2. Our algorithm first determines the best search direction (i.e., a λ vector), and then enumerates the shortest paths along that direction while gradually closing the gap between the upper bound of $c(\hat{p})$ and the lower bound of $c(\check{p})$.

Fig. 6 shows a high-level description of our algorithm, which is simply called DT. The input parameter δ is for specifying the optimality tolerance. DT first invokes the FindFP algorithm to find a set P_1 of J feasible paths. If

MCSPDTHi($G(V,E), s, t, J, c, w_r, U_r, r \in [1..R], \dot{p}$)
0 Initialize λ^*; Let $\beta = 0.5$, $C_l^* = -\infty$, $C_u^* = c(\dot{p})$, $\Delta = 0.05$
1 $(\lambda^*, C_l^*, C_u^*) \leftarrow$ MCSPDT($\lambda^*, C_l^*, C_u^*, \beta$)
2 **for** ($\beta = 0.5 + \Delta$; $\beta \leq 0.9$; $\beta = \beta + \Delta$) **do**
3 $(\lambda^*, C_l^*, C_u^*) \leftarrow$ MCSPDT($\beta, \lambda^*, C_l^*, C_u^*$)
4 **if** C_l^* is not improved **then** break the **for** loop
5 **for** ($\beta = 0.5 - \Delta$; $\beta \geq 0.1$; $\beta = \beta - \Delta$) **do**
6 $(\lambda^*, C_l^*, C_u^*) \leftarrow$ MCSPDT($\beta, \lambda^*, C_l^*, C_u^*$)
7 **if** C_l^* is not improved **then** break the **for** loop
8 return (λ^*, C_u^*)

Fig. 7. The MCSPDTHi procedure.

FindFP is successful, we further find the most expensive path $\dot{p}$ in P_1. Clearly, $c(\dot{p})$ must be an upper bound of $c(\hat{p})$. Hence, we pass $\dot{p}$ to a procedure called MCSPDTHi, which returns two parameters, λ and C_u. λ gives the best search direction, and C_u is a minimized upper bound of $c(\hat{p})$. Both λ and C_u are then passed to a procedure called MCSPPE. MCSPPE first calculates a Lagrangianized weight l according to Eq. (7), and then enumerates the shortest paths w.r.t. l until a set P_2 with the specified tolerance is found. Below we elaborate the two new procedures.

4.1 MCSPDTHi: J-MCSP Direction-Tuning Procedure

To ensure that MCSPPE solves the J-MCSP problem most efficiently, the λ returned by MCSPDTHi must give the best search direction as well as a maximized lower bound of $c(\check{p})$. To find such a λ, we can not rely on conventional LD solutions because they only try to maximize the lower bound, nor can we use the FPDT procedure because FPDT ignores the cost. MCSPDTHi achieves both goals by following a procedure similar to FPDT but taking the cost into account.

The pseudo-code of MCSPDTHi is given in Fig. 7. The three variables λ^*, C_l^* and C_u^* are, respectively, the current best values of the search direction, the lower bound and the upper bound. (In our implementation, the initial value of λ^* is set to the λ found by FPDT.) MCSPDTHi repetitively calls a lower-level procedure MCSPDT, trying to optimize λ^*, C_l^* and C_u^*. Each time when MCSPDT is called, a different value of the parameter β is provided to ensure it searches for a different region of the solution space.

As illustrated in Fig. 8, MCSPDT follows nearly the same process as FPDT. It basically keeps optimizing the objective function in Eq. (9), and during this process it preserves the λ that gives the best lower bound. Below we highlight the major differences in MCSPDT compared to FPDT.

First, the $\bar{h}$ in the objective function Eq. (9) is set to $\bar{h} = \lfloor \beta K \rfloor$ (Step 1). This $\bar{h}$ value is used to calculate $\mathcal{E}$ (Step 10) as well as to update λ (Step 17). Note that β is passed in from MCSPDTHi and $\beta \in (0, 1)$. Hence, $\bar{h}$ must be a value smaller than K. The significance for ensuring $\bar{h} < K$ will be explained shortly. Second, P_λ, which is set of the K shortest paths, is computed w.r.t. Eq. (7) (Steps 2-3). Since the cost is involved, optimizing the only multiplier for the single-constraint case is no longer a trivial issue. MCSPDT takes care of this case in the same manner as for the multiple-constraint case. Third, once P_λ is found, we calculate a lower bound with $C_l = l(p) - \lambda \mathbf{U}$, where p is the shortest path in P_λ (the first path returned by the KSP procedure). If C_l is greater than the best lower bound C_l^*, the procedure

MCSPDT($G(V,E), s, t, J, c, w_r, U_r, r \in [1..R]; \beta, \lambda^*, C_l^*, C_u^*$)
0 Set $\lambda = \lambda^*$, $\tilde{\lambda}^* = \lambda^*$
Initialize K, $\mathcal{E}^*$, y, α, $\tilde{\alpha}$, and *solved* the same as in FPDT
1 Let $\bar{h} = \lfloor \beta K \rfloor$, $\mu = \frac{\bar{h}}{2(K-\bar{h})}$
2 Use Eq. (7) to compute l
3 Let $P_\lambda \leftarrow$ KSP($l, K, C_u^* + \lambda \mathbf{U}$)
4 Let $C_l = l(p) - \lambda \mathbf{U}$, where $p = \arg\min_{q \in P_\lambda} l(q)$
5 **if** $C_l > C_l^*$ **then** set $\lambda^* = \lambda$, $C_l^* = C_l$
6 **if** P_λ has at least J feasible paths **then**
7 Let $S \leftarrow$ the least-cost J feasible paths in P_λ
8 Set $C_u^* = c(p)$, where $p = \arg\max_{q \in S} c(q)$
9 Calculate $h_r, \forall r \in [1..R]$
10 Calculate $\mathcal{E}$ by Eq. (9) using $\bar{h}$ from Step 1
11 **if** $\mathcal{E} < \mathcal{E}^*$ **then**
12 Set $\mathcal{E}^* = \mathcal{E}$, $\tilde{\lambda}^* = \lambda$, $h_r^* = h_r, \forall r \in [1..R]$
13 **else**
14 **if** $\mathcal{E} - \mathcal{E}^* > \tilde{\alpha}$ **then**
15 Set $\mu = \mu/2$, $\lambda = \tilde{\lambda}^*$, $h_r = h_r^*, \forall r \in [1..R]$
16 **if** $\mathcal{E} < \alpha$ **then** go to Step 19
17 Update λ by Eq. (10) using $\bar{h}$ from Step 1
18 $y = y + 1$; **if** $y < 20$ **then** go to Step 2
19 return $(\lambda^*, C_l^*, C_u^*)$

Fig. 8. The MCSPDT procedure.

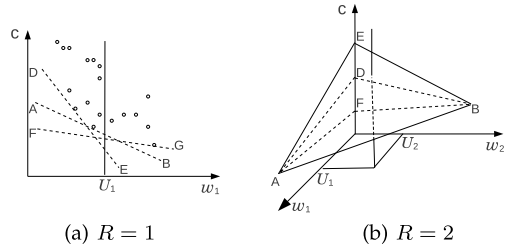
updates C_l^* and lets λ^* keep the corresponding λ. In addition, if there are J or more feasible paths in P_λ, we update the best upper bound C_u^* by utilizing the the least-cost J feasible paths in P_λ (Steps 6-8). Finally, with a proof similar to the one for Lemma 1, we can show that a path with a length greater than $C_u^* + \lambda \mathbf{U}$ would either be infeasible or has a cost greater than C_u^*. Thus, we use $C_u^* + \lambda \mathbf{U}$ as the maximum path length in the KSP procedure.

MCSPDT also guarantees $\lambda \geq \mathbf{0}$ required by Eq. (9).

Lemma 6. $\lambda \geq \mathbf{0}$ *always holds before MCSPDT returns.*

Proof. Since μ monotonically decreases after being initialized at Step 1, $\mu \leq \frac{\bar{h}}{2(K-\bar{h})}$ always holds. When updating λ with Eq. (10), we have $\mu \frac{h_r - \bar{h}}{\bar{h}} \leq \frac{h_r - \bar{h}}{2(K-\bar{h})} \leq \frac{1}{2}$. If in an iteration $h_r > \bar{h}$ for some $r \in [1..R]$, λ_r would be reduced, but by at most 50 percent of its current value. □

Now let us return to the geometric concepts to explain how different values of $\bar{h}$ are related to the lower bounds. We first consider the single-constraint case. With the only multiplier λ_1, the Lagrangianized weight of a path is $c + \lambda_1 w_1$ which, for a certain value of λ_1, corresponds to set of parallel lines in the c-w_1 plane [15]. If we set $\bar{h} = 0.5K$, MCSPDT would try to find a λ_1 such that P_λ contains 50 percent of feasible paths. Assuming the set of lines corresponding to the resulting λ_1 are in parallel with AB in Fig. 9a, the lower bound would be the c value of the point where AB intersects with the limit line (the solid vertical line). In a



(a) $R = 1$ (b) $R = 2$

Fig. 9. The relationship between $\bar{h}$ and the lower bound.

second scenario where we let $\bar{h} = 0.9K$, the P_λ corresponding to an ideal value of λ_1 would have about 90 percent of feasible paths. Since the percentage of feasible paths needs to be higher (compared to the case with $\bar{h} = 0.5K$), the resulting set of lines must be in parallel with AB rotated clockwise by a certain angle, e.g., to be in parallel with DE. Similarly, in case a lower percentage of feasible paths are to be found (e.g., by setting $\bar{h} = 0.2K$), the resulting λ_1 would correspond to a set of lines in parallel with AB rotated counterclockwise by an angle (e.g., to FG). In either case a different lower bound could be found.

Now consider the case with two constraints ($R = 2$). The Lagrangianized weight becomes $c + \lambda_1 w_1 + \lambda_2 w_2$ which, for a given λ, corresponds to a set of parallel planes in the $c{-}w_1{-}w_2$ space. Recall MCSPDT tries to find a λ such that all h_r's are equal to $\bar{h}$. Suppose such ideal solutions can always be found. With $\bar{h} = 0.5K$, MCSPDT would have a λ such that among the paths in P_λ, 50 percent of them are constraint-1 feasible and 50 percent of them are constraint-2 feasible. Let us assume the corresponding set of planes are in parallel with ABD in Fig. 9b. Then, in a second scenario with $\bar{h} = 0.9K$, because a higher percentage of constraint-r feasible paths need to be found, the set of planes corresponding to the resulting λ would be in parallel with ABD rotated along the direction of c by a certain angle (e.g., to ABE). Similarly, in case a lower percentage of constraint-r feasible paths need to be found (e.g., by setting $\bar{h} = 0.2K$), the set of planes would be in parallel with ABD rotated against the direction of c by a certain angle. In any of these cases, a lower bound can be found by looking at the c value of the point that lies on both the limit line $(w_1, w_2) = (U_1, U_2)$ and the first plane that hits a path when moving along the λ direction [15].

With the concept of hyperplane, one can visualize that the procedure works similarly when $R \geq 3$. We can also see that either a too large value or a too small value for $\bar{h}$ would lead to a small lower bound (see Fig. 9a as an example). Hence there is a best value between 0 and K for $\bar{h}$ that gives a near optimized lower bound. If the paths are uniformly distributed in the area swept through, there is a high chance that $0.5K$ is such best value. However, there are clearly exceptions. For the example in Fig. 9a, setting $\bar{h} = 0.5K$ could end up with any of the three lines (or any of three planes in Fig. 9b).

For the above reason, when calling MCSPDT, MCSPDTHi starts with $\beta = 0.5$, and then uses a hill climbing method to search in two directions, one with β set to $0.5 + \Delta$, $0.5 + 2\Delta, \ldots$, and the other with β set to $0.5 - \Delta, 0.5 - 2\Delta, \ldots$, where Δ is a stepsize (set to 0.05 in our implementation). In either direction it stops if the lower bound stops increasing or if Δ is beyond the interval $[0.1, 0.9]$. Although, a binary search can be used, we found this procedure works more efficiently.

Lemma 7. *MCSPDTHi has a worst-case time complexity of* $O(Jn(m + n \log n))$.

Proof. This is because MCSPDTHi calls MCSPDT no more than a fixed number of times, and MCSPDT and FPDT have the same worst-case time complexity. □

MCSPPE($G(V, E), s, t, J, c, w_r, U_r, r \in [1..R], \lambda, C_u, \delta$)
0 Compute $l_{ij}, f_{ij}, \forall (i,j) \in E$, and $\pi_i, \forall i \in V$
Sort the outgoing edges of each node in increasing order of f
Set M to the maximum number of iterations
Add a small value to C_u
Set $Q = \{\pi_s\}$, $P_2 = \varnothing$, $y = 0$, $solved = FALSE$
1 **if** $Q = \varnothing$ **then** set $solved = TRUE$ and go to Step 25
else remove the shortest path q from Q
2 **if** q is simple and feasible and $c(q) < C_u$ **then**
3 **if** $|P_2| < J$ **then**
4 add q to P_2
5 **if** $|P_2| = J$ **then**
6 find $\hat{p} = \arg\max_{p \in P_2} c(p)$; set $C_u = c(\hat{p})$; prune Q
7 **else**
8 Use q to replace the $\hat{p}$ in P_2
9 find $\hat{p} = \arg\max_{p \in P_2} c(p)$; set $C_u = c(\hat{p})$; prune Q
10 **if** $|P_2| = J$ **then**
11 Compute the lower bound $C_l = l(q) - \lambda \mathbf{U}$
12 **if** $(C_u - C_l)/C_u \leq \delta$ **then**
set $solved = TRUE$ and go to Step 25
13 Set $j = 1$ and let $(e_1, e_2, \cdots, e_\ell)$ be the subpath containing the deviation-qualifying edges of q
14 **while** $j \leq \ell$
15 Let $(u, a) = e_j$
16 **if** $c(S_q(s,u)) + c(\sigma_u) \geq C_u$ **or**
$\exists r \in [1..R], w_r(S_q(s,u)) + w_r(\tau_u^r) > U_r$ **then**
break the while loop
17 **if** $\exists next(u,a)$ **then**
18 Set $(u, a) = next(u, a)$
19 **if** $l(S_q(s,u) \bowtie (u,a) \bowtie \pi_a) \geq C_u + \lambda \mathbf{U}$ **then**
go to Step 23
20 **if** $a \in S_q(s,u)$ **then** go to Step 17
21 **if** $c(S_q(s,u)) + c_{ua} + c(\sigma_a) \geq C_u$ **or**
$\exists r \in [1..R], w_r(S_q(s,u)) + w_{ua}^r + w_r(\tau_a^r) > U_r$ **then**
go to Step 17
22 Add path $\langle q, (u,a) \rangle$ to Q
23 $j = j + 1$
24 $y = y + 1$; **if** $y < M$ **then** go to Step 1
25 return $(solved, P_2)$

Fig. 10. The MCSPPE procedure.

4.2 MCSPPE: J-MCSP Path-Enumeration Procedure

Fig. 10 shows the pseudo-code of MCSPPE. With the λ passed in, MCSPPE first computes the weight l according to Eq. (7) and initialize the set P_2 to be empty. It then enumerates the shortest paths w.r.t. l. If a feasible path is found, it adds the path to P_2. However, if P_2 already has J paths, the current most expensive path in P_2 would be replaced. This process continues until no more path is available in the candidate-path queue or the specified optimality tolerance is reached.

During the initialization, after the weight l is computed, the reverse shortest paths $\pi_i, \forall i \in V$ and the reduced cost function f are calculated w.r.t. l. We also slightly increase the upper bound C_u by a small value (e.g., 1 if the costs are integers). Steps 3-9 are used to update C_u and $\hat{p}$, where $\hat{p}$ is the current most expensive path in P_2. Steps 10-12 check if C_u is within δ-optimality of the current lower bound. The remaining steps are similar to those in FPPE except Steps 16, 19 and 21. Steps 16 and 21 include an additional cost-based check which ensures that a path in Q must have a cost less than C_u. Likewise, Step 19 ensures a path in Q must have a length less than $C_u + \lambda \mathbf{U}$.

We now explain why a small value should be added to C_u initially. Because MCSPPE initially sets P_2 to empty, it needs to first find J feasible paths to fill P_2 (see Steps 3-4). On the other hand, MCSPPE requires that all paths admitted to Q have a cost *less than* C_u (see Step 16). Therefore, if C_u is not increased, then in a special case when there are

exactly J feasible paths, MCSPPE could miss some feasible paths (which have been successfully found by FPPE).

When P_2 contains J feasible paths for the first time and after, different operations need to be performed. For the first time when $|P_2| = J$ (Step 6), the most expensive path $\hat{p}$ in P_2 is identified, and C_u is reset to the cost of $\hat{p}$. MCSPPE also prunes Q by removing all paths that have a length greater than or equal to $C_u + \lambda\mathbf{U}$ to save memory space and the running time. After that, if in a successive iteration a path q that has a lower cost than $\hat{p}$ is found, MCSPPE replaces $\hat{p}$ with q (Step 8), finds the *new* most expensive path in P_2, updates C_u and prunes Q similarly.

Like in FPPE, a parameter M is used to specify the maximum number of iterations. We claim the correctness of MCSPPE with a similar theorem.

Theorem 2. *As the value of M goes to ∞, MCSPPE can solve the J-MCSP problem to optimality or δ-optimality.*

Proof. The feasibility checks on Steps 16 and 21 differ from those for FPPE in that a cost-based check is enforced. If a path p is excluded from Q due to a cost-based check, all descendants of p would have a cost no smaller than C_u. If at a certain iteration Q becomes empty (Step 1), the problem is solved to optimality. Otherwise, at a certain time the δ-optimality condition at Step 12 must be met. □

Clearly, MCSPPE and FPPE have the same worst-case time complexity. Due to Lemmas 3, 5 and 7 we can conclude that the complexity of the DT algorithm is $O(Jn(m+n\log\log n)+Mn^2)$. In practice, we usually choose a number for M such that $M \gg J$, making the complexity be $O(Mn^2)$. Compared to FPPE and MCSPPE, other procedures called by DT only use negligible memory space. Thus, the total space complexity of DT is still $O(m+Mn)$.

4.3 Algorithmic Enhancements

Although DT can give a decent performance, its performance can be further improved by enhancing it with the preprocessing [21] and constraint aggregation [23] techniques. Let **DT$^+$** denote the enhanced version of DT with such techniques. We will experimentally evaluate both DT and DT$^+$ in the next two sections.

The preprocessing used in DT$^+$ is a light version of the one proposed in [21] in that only edges that do not lie on a feasible path are removed (the one in [21] also removes nodes). More accurately, the following steps are executed before the graph is passed to DT:

1) For each $v \in V$ and each $r \in [1..R]$, compute the distance ζ_r^v (along the shortest path w.r.t. w_r) from s to v, and the distance η_r^v from v to t.
2) For each edge $(i,j) \in E$, if $\zeta_r^i + w_{ij}^r + \eta_r^j > U_r$ for any $r \in [1..R]$, remove edge (i,j) from G.
3) If at least one edge is removed in Step 2, go back to Step 1. Otherwise, stop.

Note that a similar preprocessing procedure with some additional rules was used by Carlyle et al. [23].

Constraint aggregation [23] is based on the observation that a linear combination of two or more weight constraints in Eq. (1) is also a valid constraint. Thus, an aggregated constraint can be treated in the same manner as for a constraint in Eq. (1). A reverse shortest path tree w.r.t. each aggregated weight needs to be precomputed, and because the number of constraints is increased, some extra computations are needed for the feasibility checks (Steps 7 and 12 in FPPE, and Steps 16 and 21 in MCSPPE). Thus, an aggregated constraint formed carelessly could increase the computation burden without limiting any additional path enumeration. However, it seems no general guidelines can be established for how to make this technique effective since the information that it can rely on usually varies from algorithm to algorithm.

DT$^+$ uses a pairwise aggregation in both FPPE and MCSPPE if $R \geq 2$. With FPPE as an example, assuming the h_r's associated with λ are preserved, we find $h_\kappa = \min\{h_r | r \in [1..R]\}$, and form $R-1$ aggregated constraints as follows:

$$\lambda_i w_i + \lambda_\kappa w_\kappa \leq \lambda_i U_i + \lambda_\kappa U_\kappa, \forall i \in [1..R], i \neq \kappa.$$

The aggregated constraints used in MCSPPE are formed similarly. Following the definition of h_r, constraint κ could most likely be violated when enumerating paths along the chosen direction. Thus, a combination of constraint κ with another constraint would be more effective than other such combinations to detect infeasible paths.

DT$^+$ also uses four cost-based aggregated constraints in MCSPPE for both single and multiple-constraint cases

$$\begin{aligned} &c + 2\lambda\mathbf{w} \leq C_u + 2\lambda\mathbf{U}, \\ &2c + \lambda\mathbf{w} \leq 2C_u + \lambda\mathbf{U}, \\ &c + 4\lambda\mathbf{w} \leq C_u + 4\lambda\mathbf{U}, \text{ and} \\ &4c + \lambda\mathbf{w} \leq 4C_u + \lambda\mathbf{U}, \end{aligned}$$

where $\mathbf{w} = (w_r | r \in [1..R])$ is the vector composed of the R weights. In DT$^+$, each of these constraints is treated as if it is another weight constraint, but each time when C_u is changed, the corresponding upper bounds are updated as well.

5 Experimental Results for MCSP

We dedicate this section to presenting experimental results for the traditional MCSP problem. To solve it, we simply use our algorithm for J-MCSP by letting $J = 1$. All experiments were conducted on a 32-bit Linux system with a 3.4 GHz Core i7 PC, which has a total of 12 GB memory, but a single process is limited to a maximum of 3 GB. The KSP algorithm used in FPDT and MCSPDT was the one proposed in [27].

We compared our algorithm with several other algorithms, but mostly against the LRE-PA proposed in [23]. We implemented LRE-PA by exactly following the procedure described in [23], including their suggested preprocessing and constraint aggregation. Unlike DT and DT$^+$ which would report a failure if memory is insufficient or M iterations have been repeated, LRE-PA could run infinitely and an upper limit of the execution time has to be specified. We set this time limit to 15 minutes for all experiments.

Our testing shows the 15-minute limit would allow MCSPPE to repeat about 20 million iterations for some

TABLE 1
Run Time (Sec.) for Solving One Singly-Constrained Instance to Optimality

	$\gamma = 0.05$			$\gamma = 0.5$		
$a \times b$	DT	DT$^+$	LRE-PA	DT	DT$^+$	LRE-PA
30×100	0.02	0.02	0.12(0.06)	0.008	0.008	0.02(0.03)
100×100	0.03	0.02	0.02(0.08)	0.03	0.03	0.03(0.05)
200×200	0.09	0.12	0.17(0.25)	0.07	0.1	0.1(0.2)
350×200	0.18	0.22	0.21(0.38)	0.17	0.22	0.27(0.27)
450×300	1.01	1.1	0.89(0.99)	0.35	0.44	0.55(0.74)

TABLE 2
The Average and Standard Deviation of Run Time (Sec.) for Solving Singly-Constrained Random Instances to Optimality

		DT		DT$^+$		LRE-PA	
$a \times b$	γ	avg.	s.d.	avg.	s.d.	avg.	s.d.
30×100	0.05	0.02	0	0.02	0	0.01	0
100×100	0.05	0.03	0.04	0.02	0.01	0.02	0
200×200	0.05	0.19	0.41	0.16	0.22	0.24	0.36
350×200	0.05	0.27	0.34	0.24	0.25	0.48	1.3
450×300	0.05	6.7	43.3	1.3	4.3	3.4	18
30×100	0.5	0.02	0.02	0.02	0.02	0.02	0.01
100×100	0.5	0.02	0.02	0.03	0.02	0.04	0.02
200×200	0.5	0.2	0.35	0.22	0.29	0.65	1.1
350×200	0.5	0.24	0.32	0.28	0.26	2.3	12.4
450×300	0.5	1.2	3.4	1.2	3	17.5†	52.3†
30×100	0.95	0.02	0	0.02	0	0.01	0
100×100	0.95	0.02	0	0.02	0	0.02	0
200×200	0.95	0.11	0.1	0.13	0.06	0.22	0.27
350×200	0.95	0.19	0.09	0.25	0.08	0.36	0.24
450×300	0.95	0.82	2	0.99	2.7	2.9	10

(†*LRE-PA failed to solve one instance).*

hardest instances used in our experiments. Since we were interested in knowing how our algorithm performs in an extreme condition, the M (for FPPE and MCSPPE) was set 20 million for all experiments unless explicitly mentioned. To make the comparison fair, a result of our algorithm was treated as a failure if the actual running time exceeds 15 minutes. Considering some applications require much shorter worst-case running time, we will provide some results with M set to much smaller values.

In our experiments we used cyclic/acyclic grid networks and random networks. Due to space limit we only report some typical results on acyclic grid networks. (A complete set of results can be found in an extended technical report [35].) Instances on acyclic grids were used in both [21] and [23], and are known to be hard benchmarks. An acyclic grid in the form $a \times b$ has a rows and b nodes in each row. Each node has edges going right, up, and down to its neighboring nodes if any such connection is possible. An additional node s is connected all nodes on the leftmost column, and an additional node t is connected all nodes on the rightmost column. As a result, an acyclic grid has $n = ab + 2$ and $m \approx 3n$. The edge cost and weights are integers in the interval $[1, 10]$ for vertical edges and in the interval $[80, 100]$ for horizontal edges.

For all experiments, the upper bound of an instance is set to $U_r = w_r(\tau_s^r) + \gamma(w_r(\sigma_s) - w_r(\tau_s^r)), \forall r \in [1..R]$, where $\gamma > 0$ will be referred to as *constraint factor*. Unless explicitly mentioned, all results were obtained with each instance *solved to optimality* ($\delta = 0$).

5.1 Singly-Constrained Problems

Table 1 shows the run time (in seconds) of each algorithm solving one singly-constrained benchmark instance for each grid network. These instances were tested in both [21] and [23]. For LRE-PA, we also provided the run times reported in [23] (in the parentheses), which were achieved on a 3.8 GHz P4 processor.

Table 2 shows the average (avg.) and standard deviation (s.d.) of the running time for each algorithm to solve randomly generated single-constraint instances. Each statistical result was obtained by solving 50 random instances. The 50 instances for each $a \times b$ grid have the same topology as those in Table 1, but the cost and edge weight were random numbers with the random seeds set to 1-50, respectively. Note that a "0" for the avg. or s.d. simply means the time is less than 0.01 seconds. All three algorithm were able to successfully solve all instances except LRE-PA failed one for the 450×300 grid with $\gamma = 0.5$. When the avg. and s.d. values were calculated, only the results of solved instances were used (same for other results presented hereafter). These results indicate that, with the increase of network size the average running time of LRE-PA overall increases faster. DT$^+$ shows a more robust performance than DT and LRE-PA (compare the results for the 450×300 grids).

Here we extend the comparison to two more algorithms: the heuristic algorithm LARAC [2] and the ϵ-approximation algorithm FPTAS-DCLC [8]. These two algorithms were designed for solving the singly-constrained problem only. We tested them with the instances used in Table 2. Due to space limit, we only report the results for the case with $\gamma = 0.5$ as below. Corresponding to the five different sizes of networks, the average running times of LARAC are 0.01, 0.02, 0.13, 0.25 and 0.41 seconds, respectively. Note that the running time of DT is nearly comparable to that of LARAC except for the 450×300 grids. On the other hand, unlike DT solved every instance to optimality, LARAC was not able to find the optimal solution to any instance (because it is a heuristic). The solution of LARAC is about 5-20 percent above the optimal cost. The FPTAS-DCLC was tested with $\epsilon = 0.5$, and it was able to find the optimal solutions of 35 instances for the 30×100 grid with an average running time of 0.78 seconds. However, for the instances of other network sizes, FPTAS-DCLC was not able to solve any of them because it ran out of memory.

5.2 Multi-Constrained Problems

We generated the multi-constrained instances in the same manner as for the single-constraint case, except that more random numbers were generated for each edge. Tables 3 and 4 show the results of some similar experiments used in [23]. We set R to 2-5 and set γ to 0.5 and 0.95, respectively. The "n.s." column gives the number of instances (out of 50) solved successfully.

Table 3 shows that, when $\gamma = 0.5$, the number of instances solved by LRE-PA decreases very quickly with R or the network size. In contrast, both DT and DT$^+$ can solve much

TABLE 3
The Average Running Time (Sec.) and the Number of Instances Solved to Optimality When $R \geq 2$ and $\gamma = 0.5$

		DT		DT$^+$		LRE-PA		
$a \times b$	R	avg.	n.s.	avg.	n.s.	avg.	n.s.	s.r.
	2	0.06	50	0.06	50	1.1	50	19
	3	0.26	50	0.2	50	35.7	46	225
30×100	4	1.2	50	0.65	50	81.2	42	161
	5	5	48	2.9	49	158	25	77
	2	0.08	50	0.08	50	1.2	50	14
	3	0.69	50	0.35	50	65.4	48	186
100×100	4	5	48	1.2	49	70.9	30	64
	5	9.8	44	6.1	48	169	23	22
	2	5.7	49	12	50	42.6	42	39
	3	9	46	10.7	48	204	26	12
200×200	4	16.8	39	28.1	44	256	9	57
	5	55.4	17	65	26	122	5	4.8
	2	1.3	47	1.1	47	80.2	42	74
	3	4.8	45	15.4	48	216	22	92
350×200	4	26.9	43	27	47	279	10	70
	5	83.2	23	76.6	33	269	3	79
	2	9.6	50	5.6	50	95.1	31	73
	3	37.8	40	49.7	42	209	4	88
450×300	4	91.8	35	98.6	39	337	3	28
	5	133	12	233	21	118	3	12

more instances. Moreover, our algorithm takes significantly less time than LRE-PA. To put this in perspective, we report the speed up ratio of DT$^+$ over LRE-PA in the last column entitled by "s.r.", which is computed by dividing the avg. time of LRE-PA by that of DT$^+$ for the instances solved by both algorithms.

Table 4 shows that, with γ increased to 0.95, all three algorithms can solve a decent percentage of instances except that LRE-PA performs clearly worse for the 450×300 grids.

TABLE 4
The Average Running Time (Sec.) and the Number of Instances Solved to Optimality When $R \geq 2$ and $\gamma = 0.95$

		DT		DT$^+$		LRE-PA		
$a \times b$	R	avg.	n.s.	avg.	n.s.	avg.	n.s.	s.r.
	2	0.02	50	0.03	50	0.03	50	1
	3	0.04	50	0.05	50	13.1	50	250
30×100	4	0.05	50	0.05	50	10.3	50	175
	5	0.07	50	0.08	50	15.5	49	194
	2	0.04	50	0.05	50	9.1	50	174
	3	0.08	50	0.09	50	2.4	48	26
100×100	4	0.1	50	0.12	50	1.1	49	9.6
	5	0.14	50	0.16	50	9.6	50	58
	2	0.21	50	0.24	50	1.4	48	5.7
	3	0.65	50	0.76	50	19.8	46	28
200×200	4	2.1	50	0.89	50	32.5	42	50
	5	4.4	50	2.9	50	113	38	118
	2	0.58	50	0.67	50	3.9	49	5.8
	3	0.97	50	1.1	50	32.4	45	31
350×200	4	1.6	50	1.6	50	77.2	39	53
	5	2.1	50	2.1	50	119	38	58
	2	4.2	50	2.5	50	23.1	45	34
	3	6	50	5.2	50	59.3	36	47
450×300	4	5.3	48	5.8	49	96	29	56
	5	24.7	50	11.4	50	153	19	58

TABLE 5
The Average Running Time (Sec.) and the Number of Instances Solved for Finding a Feasible Path by DT, RMCP and SAMCRA ($R = 2$ and $\gamma = 0.5$)

	DT		RMCP		SAMCRA	
$a \times b$	avg.	n.s.	avg.	n.s.	avg.	n.s.
30×100	0.06	50	0.005	0	267.6	50
100×100	0.08	50	0.01	2	805.1	13

The reason is that, with a relatively large γ, the instances with a smaller R value are more likely to be loosely constrained and thus easier to solve (discussed in the next section). The advantage of our algorithm, however, is still quite clear; The s.r. column shows that the speed up of our solution over LRE-PA is in one or two orders of magnitude (similar to those in Table 3).

Here we extend the comparison to two more algorithms: the randomized algorithm RMCP [36] and the SAMCRA algorithm [10]. Both RMCP and SAMCRA were designed to find a *feasible* path subject to multiple constraints. They differ in that RMCP is a heuristic while SAMCRA is an exact algorithm. We implemented RMCP by following the pseudo-code in [36]. The source code of SAMCRA was provided by the original authors (available at [37]). Note that their code can only handle problems with no more than two constraints.

We tested RMCP and SAMCRA with two sets of instances, those on the 30×100 and 100×100 grids with $R = 2$ and $\gamma = 0.5$. The DT algorithm proved that each of these instances has at least one feasible path. Table 5 shows the average running time in seconds and the number of instances that each algorithm was able to find a feasible solution. (The 15-minute time limit was applied to each algorithm.) Note that RMCP ran faster than DT, but RMCP could hardly solve any instance. SAMCRA was able to solve all instances on the 30×100 grid, but it failed to solve most instances on the 100×100 grid. In addition, SAMCRA was slower than DT by several orders of magnitude, even though DT found the *optimal* solution, and SAMCRA only needed to find a *feasible* path. Further testing shows SAMCRA would run out of memory for instances on other larger grids.

5.3 Hard Multi-Constrained Problems

In [21] and [23], the weight limits with $\gamma = 0.05$, 0.5 and 0.95 were called *low*, *medium* and *high*, respectively, to distinguish whether an instance is tightly, moderately, or loosely constrained. We have so far avoided using this categorization since we believe it is appropriate only for the singly-constraint case. When $R = 1$, an instance with $\gamma \geq 0$ must be feasible, and an instance with $\gamma \geq 1$ is trivial because the least-cost path is the optimal solution. However, for an instance with $R \geq 2$, $\gamma \geq 0$ no longer guarantees its feasibility, and $\gamma \geq 1$ does not necessarily mean it is easy to solve. Statistically, as R increases, the minimum γ value (say γ_0) that ensures an instance is feasible must increase as well. (This follows $\rho = \prod_{r=1}^{R} \xi_r$ in Section 3.1, where ξ_r would be a function of γ.) Likewise, the minimum γ value (say γ_1) that ensures the least-cost path is feasible must increase with R.

TABLE 6
The Number of Feasible Instances for a Tuple (a, b, R, γ)

$a \times b$	R	$\gamma \leq 0.2$	0.3	0.4	0.5	0.6	≥ 0.7
30×100	2	0	44	**50**	50	50	50
	4	0	0	5	46	**50**	50
	6	0	0	0	26	49	**50**
	8	0	0	0	9	49	**50**
	10	0	0	0	0	39	**50**
100×100	2	0	44	**50**	50	50	50
	4	0	0	7	49	**50**	50
	6	0	0	0	18	**50**	50
	8	0	0	0	5	46	**50**
	10	0	0	0	3	44	**50**
200×200	2	0	49	**50**	50	50	50
	4	0	0	10	**50**	50	50
	6	0	0	0	38	**50**	50
	8	0	0	0	9	**50**	50
	10	0	0	0	0	**50**	50

Clearly, the hardness of an instance is a bell-shaped function of γ, reaching the maximum for some γ value between γ_0 and γ_1. This means the range of γ values that make the instance harder to solve increase with R as well.

To find the harder instances, with FindFP we analyzed, for a selected set of tuples (a, b, R, γ), the number of instances (out of 50) that have at least one feasible path. The results for the grids up to 200×200 are shown in Table 6 (more results are available in [35]). Based on these results, we give a new categorization of the limit factor. For a given tuple (a, b, R), we call the minimum γ value that guarantees all 50 instances are feasible for the *low* limit factor, denoted by γ_l. The γ_l value for each case is highlighted in Table 6. We then set the *high* limit factor $\gamma_h = \gamma_l + 0.3$ and the medium $\gamma_m = (\gamma_l + \gamma_h)/2$.

Table 7 are the results for instances on the 30×100 grids grouped in the order of low, medium and high weight limits. Instances in the low-limit group are the hardest, as we see LRE-PA failed to solve any instance with $R \geq 8$. In comparison, both DT and DT$^+$ can solve a decent percentage of instances. With medium limits, the number of instances solved by LRE-PA decreases quickly with R, while both DT and DT$^+$ solved nearly all of them. For both the medium- and high-limit groups, the speed-up ratio of DT$^+$ over LRE-PA shows that DT$^+$ is faster than LRE-PA often by two orders of magnitude.

TABLE 7
The Average Running Time (Sec.) and the Number of Instances Solved to Optimality for the Instances on 30×100 Grids with Some Different R and γ Combinations

		DT		DT$^+$		LRE-PA		
R	γ	avg.	n.s.	avg.	n.s.	avg.	n.s.	s.r.
2	0.4	0.06	50	0.06	50	0.43	50	6.8
4	0.6	1.4	49	1.8	50	97.9	45	48
6	0.7	9.1	45	9.7	49	216	22	22
8	0.7	19.8	43	15.1	46	0	0	
10	0.7	26.5	24	41.7	35	0	0	
2	0.55	0.05	50	0.06	50	15.5	47	248
4	0.75	0.46	49	0.26	49	99.4	30	391
6	0.85	2.3	50	1.7	50	46.7	45	26
8	0.85	3.3	50	2.4	50	151	17	372
10	0.85	5.2	49	7	50	354	10	460
2	0.7	0.03	50	0.04	50	36.5	43	892
4	0.9	0.11	50	0.11	50	19.7	47	163
6	1	0.04	50	0.04	50	19.4	50	395
8	1	0.04	50	0.05	50	12.3	48	210
10	1	0.05	50	0.06	50	80.7	49	1200

Table 8 contains the results of DT$^+$ and LRE-PA with hard instances on the other larger grids. The columns entitled by "DT$^+$(1 percent)" are the results of DT$^+$ with a 1 percent optimality tolerance ($\delta = 0.01$). These statistics were preserved in the progress of solving the problem to optimality, and thus did not require to rerun the algorithm [14]. (This feature is not available for an A*-based algorithm like LRE-PA.) The results are easier to read if we compare when an algorithm fails to solve more than 10 instances. For example, in the low-limit case, it happens to DT$^+$ when $R \geq 8$ and the grid size is 200×200 (or larger); However, LRE-PA can solve more than 10 instances only when $R \leq 4$ on the 100×100 grids or when $R = 2$ on the 200×200 grids. The DT$^+$(1 percent) columns indicate that, despite the difficulty, DT$^+$ can solve a decent percentage of instances with a low average running time if a 1 percent optimality tolerance is allowed.

TABLE 8
The Average Running Time (Sec.) and the Number of Instances Solved for the Hard Instances on Other Grids

		Low limit						Medium limit						High limit					
		DT$^+$		LRE-PA		DT$^+$ (1%)		DT$^+$		LRE-PA		DT$^+$ (1%)		DT$^+$		LRE-PA		DT$^+$ (1%)	
$a \times b$	R	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.
100×100	2	0.1	50	22.9	50	0.1	50	0.1	50	0.72	50	0.09	50	0.4	50	25.4	48	0.11	50
	4	0.7	50	129	45	0.25	50	1.1	50	118	34	0.27	50	0.28	50	12.9	41	0.24	50
	6	20.8	43	286	7	1.5	50	8	47	91.9	9	0.6	50	0.55	50	30.9	40	0.31	50
	8	26.9	35	0	0	0.68	49	4.5	48	146	25	0.45	50	0.09	50	19.6	48	0.09	50
	10	42.7	25	743	1	1.2	49	13.5	46	122	15	0.67	50	0.11	50	39	44	0.11	50
200×200	2	1.5	50	40.9	33	0.56	50	2.6	50	80.3	41	0.77	50	13	50	24.3	39	0.54	50
	4	30.2	44	258	9	1.7	48	17.8	39	52.2	4	1.6	49	28.2	46	236	13	1.4	50
	6	61	18	5.1	1	4	48	71.9	30	176	2	2.3	44	54.1	48	254	13	1.9	50
	8	197	3	0	0	7.7	38	153	13	0	0	9.7	45	49.6	41	412	4	3.1	49
	10	0	0	0	0	17.1	23	108	6	0	0	9.3	41	66.8	36	742	1	3.2	49

The "DT$^+$(1 percent)" column shows the results of DT$^+$ with a 1 percent optimality tolerance.

TABLE 9
The Average Running Time (Sec.) and the Number of Instances Solved to Optimality by DT$^+$ with Other M and K Combinations for the Hard Instances on 30×100 Grids

		$M = 1$ million				$M = 100{,}000$			
		$K = 1000$		$K = 100$		$K = 1000$		$K = 100$	
R	γ	avg.	n.s.	avg.	n.s.	avg.	n.s.	avg.	n.s.
2	0.4	0.05	50	0.03	50	0.06	50	0.03	50
4	0.6	0.32	47	0.2	46	0.18	43	0.12	42
6	0.7	1.1	37	1.1	37	0.36	19	0.17	17
8	0.7	2.2	27	1.7	25	0.63	11	0.37	9
10	0.7	2.5	13	2	11	0.49	4	0.27	4
2	0.55	0.05	50	0.02	50	0.06	50	0.02	50
4	0.75	0.25	49	0.16	49	0.22	47	0.09	45
6	0.85	0.58	49	0.35	48	0.24	43	0.14	42
8	0.85	0.91	45	0.73	45	0.42	38	0.19	37
10	0.85	1.2	44	1.1	44	0.57	26	0.32	27
2	0.7	0.03	50	0.02	50	0.04	50	0.02	50
4	0.9	0.11	50	0.05	50	0.11	50	0.05	50
6	1	0.04	50	0.01	50	0.04	50	0.01	50
8	1	0.05	50	0.02	50	0.05	50	0.02	50
10	1	0.06	50	0.02	50	0.06	50	0.02	50

5.4 Results with Other M and K Values

Table 9 shows the results of DT$^+$ for the instances used in Table 7, but with different combinations of M and K values (recall K is for the KSP procedure in FPDT and MCSPDT). Comparing between Tables 7 and 9 shows LRE-PA was able to solve a few more instances than DT$^+$ only when $R = 6$, $\gamma = 0.7$ and $M = 100{,}000$. In the low or medium limit categories, the number of instances solved by DT$^+$ decreases when M decreases, but at a reasonable pace. In the high limit category, DT$^+$ was able to solve all instances even when $M = 100,000$. One should also notice how the avg. times of DT$^+$ change with M and K. Comparing the columns between $K = 1000$ and $K = 100$ in Table 9 shows DT$^+$ could work equally well for a large range of K values.

6 Experimental Results for J-MCSP

To compare our algorithm for the J-MCSP problem against LRE-PA, we first improved LRE-PA [23] as a new algorithm called JA*. JA* shares most subroutines of LRE-PA but differs in that it first finds J feasible paths, and continues to search for the minimum-cost J feasible paths. Compared to the A*prune in [11], JA* is enhanced with preprocessing and constraint aggregation, which ensure JA* works better when solving hard instances on large networks.

Table 10 shows the running time to solve each single instance used in Table 1 with J set to 10, 100 and 1,000, respectively. For the instances on 350×200 or smaller grids, DT or DT$^+$ solved each of them within 2 seconds; In comparison, the running times of JA* have a large dynamic range reaching close to 200 seconds. For instances on 450×300, while all algorithms took more time, the difference between our algorithm and JA* is quite clear.

Table 11 shows the results for experiments with random instances on the 100×100 grids with $\gamma = 0.5$ and $R = 1$, 3, and 5. Recall from Table 3 that the instances with $R \leq 3$ and $\gamma = 0.5$ are relatively loosely constrained, and thus easy to solve if $J = 1$. With J significantly larger, however, they are much harder, as we see the number of instances solved by JA* drops quickly with J when $R = 3$. In comparison, DT$^+$ was able to solve all instances for $R = 1$ or 3. The "Inf." column shows the number of instances that have fewer than J feasible paths. Since instances in such a category are easy to solve, for the $J = 1000$ group JA* was able to solve 21 instances when $R = 5$, but only 2 when $R = 3$. The s.r. column is the speed up ratio of DT$^+$ over JA*.

TABLE 10
Run Time (Sec.) for a Single-Constraint J-MCSP Instance

		$\gamma = 0.05$			$\gamma = 0.5$		
$a \times b$	J	DT	DT$^+$	JA*	DT	DT$^+$	JA*
30×100	10	0.04	0.04	0.35	0.01	0.02	0.31
	100	0.27	0.15	1.3	0.13	0.14	2.1
	1000	0.6	0.54	7.3	0.3	0.4	30
100×100	10	0.05	0.04	0.06	0.03	0.03	0.22
	100	0.13	0.07	0.41	0.09	0.1	1.3
	1000	0.21	0.18	6.4	0.26	0.16	5.4
200×200	10	0.14	0.18	0.32	0.1	0.13	0.19
	100	0.18	0.3	2.3	0.14	0.15	45
	1000	0.4	0.4	18	0.5	0.55	189
350×200	10	0.18	0.23	0.3	1.3	1.4	0.45
	100	0.2	0.26	0.99	1.5	1.5	1.2
	1000	0.4	0.6	8.9	1.8	1.9	5.6
450×300	10	1.1	1.2	2.3	0.5	0.6	5.2
	100	4.3	4	8.2	1.8	2.1	35
	1000	10	7.3	79	1.8	2.8	238

7 Conclusion

We presented a new algorithm that combines the Lagrangian relaxation technique with a path ranking based method to find multiple shortest paths subject to multiple additive constraints. Our algorithm is centered around the recognition that the Lagrangian multipliers used in the path enumeration procedure should not only give a near-optimized lower bound, but more importantly should give the best search direction that would minimize the number of infeasible paths encountered. Although Santos et al. [6] discussed a similar issue for the single-constraint case, no prior work

TABLE 11
The Average Running Time (Sec.) and the Number of Instances Solved to Optimality for the J-MCSP Instances on 100×100 Grids with $\gamma = 0.5$

		DT		DT$^+$		JA*			
J	R	avg.	n.s.	avg.	n.s.	avg.	n.s.	Inf.	s.r.
10	1	0.04	50	0.05	50	0.37	50	0	7
	3	2.6	50	1	50	184	36	0	363
	5	18.5	41	14.8	48	114	22	15	17
100	1	0.11	50	0.11	50	3.3	50	0	29
	3	4.9	49	2.8	50	369	12	0	145
	5	25.5	31	21.8	39	90.7	21	18	10
1000	1	0.26	50	0.26	50	28.1	50	0	107
	3	21.7	47	14.1	50	525	2	0	53
	5	10.9	23	30.2	33	59.4	21	21	7.5

brought up this recognition in the generic multiple-constraint case. The performance of our algorithm was verified through experiments on relatively large networks, including the most challenging instances on grid networks. We showed that our algorithm can solve a significantly larger number of instances to optimality with less computational cost, often by one or two orders of magnitude, when compared with the best known algorithm in the literature.

REFERENCES

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

[2] A. Jüttner, B. Szviatovszki, I. Mëcs, and Z. Rajkö, "Lagrange relaxation based method for the QoS routing problem," in *Proc. IEEE Conf. Comput. Commun.*, 2001, pp. 859–868.

[3] F. Kuipers, P. V. Mieghem, T Korkmaz, and M. Krunz, "An overview of constraint-based path selection algorithm for QoS routing," *IEEE Commun. Mag.*, vol. 40, no. 12, pp. 50–55, Dec. 2002.

[4] S. Chen, M. Song, and S. Sahni, "Two techniques for fast computation of constrained shortest paths," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 105–155, Feb. 2008.

[5] L. D. P. Pugliese and F. Guerriero, "A survey of resource constrained shortest path problems: Exact solution approaches," *Networks*, vol. 62, no. 3, pp. 183–200, Jan. 2013.

[6] L. Santos, J. Coutinho-Rodrigues, and J. R. Current,"An improved solution algorithm for the constrained shortest path problem," *Transport. Res. Part B: Method.*, vol. 41, no. 7, pp. 756–771, 2007.

[7] L. Lozanoand A. L. Medaglia,"On an exact method for the constrained shortest path problem," *Comput. Oper. Res.*, vol. 40, no. 1, pp. 378–384, 2013.

[8] G. Xue, W. Zhang, J. Tang, and K. Thulasiraman, "Polynomial time approximation algorithms for multi-constrained QoS routing," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, pp. 656–669, Jun. 2008.

[9] F.A. Kuipers and P. Van Mieghem, "Conditions that impact the complexity of QoS routing," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 717–730, Aug. 2005.

[10] P. V. Mieghem and F. A. Kuipers, "Concepts of exact quality of service algorithms," *IEEE/ACM Trans. Netw.*, vol. 12, no. 5, pp. 851–864, Oct. 2004.

[11] G. Liu and R. Ramakrishnan, "A*prune: An algorithm for finding K shortest paths subject to multiple constraints," in *Proc. IEEE Conf. Comput. Commun.*, 2001, pp. 743–749.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.

[13] Y. Li, J. Harms, and R. Holte, "Fast exact multiconstrained shortest path algorithms," in *Proc. IEEE Int. Conf. Commun.*, 2006, pp. 123–130.

[14] G. Y. Handler and I. Zang, "A dual algorithm for the constrained shortest path problem," *Networks*, vol. 10, no. 4, pp. 293–309, 1980.

[15] M. Ziegelmann, "Constrained shortest paths and related problems," Ph.D. dissertation, Faculty Natural Sci. Technol., Universität des Saarlandes, Saarbrcken, Germany, 2001.

[16] J. Y. Yen,"Finding the K shortest loopless paths in a network," *Manage. Sci.*, vol. 17, no. 11, pp. 712–716, 1971.

[17] Z. Gotthilf and M. Lewenstein, "Improved algorithms for the k simple shortest paths and the replacement paths problems," *Inf. Process. Lett.*, vol. 109, pp. 352–355, 2009.

[18] J. Beasleyand N. Christofides,"An algorithm for the resource constrained shortest path problem," *Networks*, vol. 19, no. 4, pp. 379–394, 1989.

[19] Y. Aneja, V. Aggarwal, and K. Nair, "Shortest chain subject to side conditions," *Networks*, vol. 13, pp. 295–302, 1983.

[20] M. Desrochers and F. Soumis, "A generalized permanent labeling algorithm for the shortest path problem with time windows," *Inf. Syst. Oper. Res.*, vol. 26, pp. 191–212, 1988.

[21] I. Dumitrescu and N. Boland, "Improved preprocessing, labeling and scaling algorithm for the weight-constrained shortest path problem," *Networks*, vol. 42, pp. 135–153, 2003.

[22] X. Zhu, W. E. Wilhelm,"A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation," *Comput. Oper. Res.*, vol. 39, no. 2, pp. 164–178, 2012.

[23] W. M. Carlyle, J. O. Royset, and R. K. Wood,"Lagrangian relaxation and enumeration for solving constrained shortest-path problems," *Networks*, vol. 52, no. 4, pp. 256–270, 2008.

[24] D. DeWolfe, J. Stevens, and K. Wood, "Setting military reenlistment bonuses," *Naval Res. Logistics*, vol. 40, pp. 143–160, 1993.

[25] D.V. Leggieri, "Multicast problems in telecommunication networks," Ph.D. Dissertation, Dipartimento di Matematica e Fisica, Universitá Del Salento, Lecce, Italy, 2009.

[26] N. Shi, "K constrained shortest path problem," *IEEE Trans. Autom. Sci. Eng.*, vol. 7, no. 1, pp. 15–23, Jan. 2010

[27] G. Feng, "Improving space efficiency with path length prediction for finding k shortest simple paths," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2459–2472, Oct. 2014.

[28] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1993.

[29] A. Schrijver, *Theory of Linear and Integer Programming*. New York, NY, USA: Wiley, 1986.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2001.

[31] W. C. Waterhouse, "Do symmetric problems have symmetric solutions?" *Amer. Math. Monthly*, vol. 90, no. 6, pp. 378–387, 1983

[32] A. Sedeño-Noda, "An efficient time and space K point-to-point shortest simple paths algorithm," *Appl. Math. Comput.*, vol. 218, pp. 10244–10257, 2012.

[33] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi,"Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[34] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, vol. 28, pp. 652–673, 1998.

[35] G. Feng. (2013). Finding multiple shortest paths subject to multiple constraints: A direction tunning algorithm. Dept. of Electrical Engineering, Univ. of Wisconsin, Platteville, Tech. Rep [Online]. Available: http://www.uwplatt.edu/~fengg/

[36] T. Korkmaz and M. Krunz, "A randomized algorithm for finding a path subject to multiple QoS constraints," *Comput. Netw.*, vol. 36, pp. 251–268, 2001.

[37] TOTEM Project: Toolbox for Traffic Engineering Methods [Online]. Available: http://totem.run.montefiore.ulg.ac.be/algos/samcra.html, 2007.

Gang Feng received the BE and ME degrees in electronic engineering from the University of Electronic Science and Technology of China in 1992 and 1995, respectively. Between 1995 and 1998, he was studying in Beijing University of Posts and Telecommunications. He received the PhD degree in electrical engineering from the University of Miami in 2001. After graduation, he joined the faculty of the Electrical Engineering Department in the University of Wisconsin - Platteville, where he currently is a full professor. His research interests include routing algorithms, heuristic and approximation algorithms, wireless sensor networks, neural networks, and evolutionary computation. He is a member of the IEEE.

Turgay Korkmaz received the BSc degree with the first ranking in computer science and engineering at Hacettepe University, Ankara, Turkey, in 1994, and two MSc degrees in computer engineering at Bilkent University, Ankara, and computer and information science at Syracuse University, Syracuse, NY, in 1996 and 1997, respectively. In December 2001, he received the PhD degree from Electrical and Computer Engineering from the University of Arizona, under the supervision of Dr. Marwan Krunz. In January 2002, he joined the University of Texas at San Antonio as an assistant professor of Computer Science Department. He received the tenure in September 2008, and he is currently an associate professor of Computer Science Department. He works in the area of computer networks and network security. Specifically, he is interested in quality-of-services (QoS)-based networking issues in both wireline and wireless networks. He is a member of the IEEE.