On Degree Constrained Shortest Paths

Samir Khuller *, Kwangil Lee **, and Mark Shayman *** †

University of Maryland

Abstract. Traditional shortest path problems play a central role in both the design and use of communication networks and have been studied extensively. In this work, we consider a variant of the shortest path problem. The network has two kinds of edges, "actual" edges and "potential" edges. In addition, each vertex has a degree/interface constraint. We wish to compute a shortest path in the graph that maintains feasibility when we convert the potential edges on the shortest path to actual edges. The central difficulty is when a node has only one free interface, and the unconstrained shortest path chooses two potential edges incident on this node. We first show that this problem can be solved in polynomial time by reducing it to the minimum weighted perfect matching problem. The number of steps taken by this algorithm is $O(|E|^2 \log |E|)$ for the single-source single-destination case. In other words, for each v we compute the shortest path P_v such that converting the potential edges on P_v to actual edges, does not violate any degree constraint. We then develop more efficient algorithms by extending Dijkstra's shortest path algorithm. The number of steps taken by the latter algorithm is $O(|E||V|)$, even for the single-source all destination case.

1 Introduction

The shortest path problem is a central problem in the context of communication networks, and perhaps the most widely studied of all graph problems. In this paper, we study the *degree constrained shortest path problem* that arises in the context of dynamically reconfigurable networks. The objective is to compute shortest paths in the graph, where the edge set has been partitioned into two classes, such that for a specified subset of vertices, the number of edges on the path that are incident to it from one of the classes is constrained to be at most one.

* S. Khuller is with Department of Computer Science, University of Maryland, College Park, MD 20742, USA. Tel: (301) 405-6765, Fax: (301) 314-9658 E-mail: samir@cs.umd.edu. Research supported by NSF grants CCR-0113192 and CCF-0430650.

** K. Lee is with Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA. E-mail: kilee88@yahoo.com

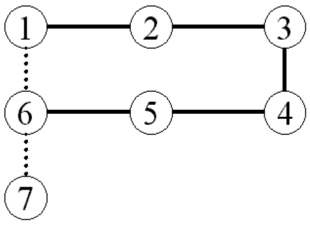
*** M. Shayman is with Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742, USA. Tel: 301-405-3667, Fax: 301-314-9281, E-mail: shayman@glue.umd.edu

† Research partially supported by AFOSR under contract F496200210217

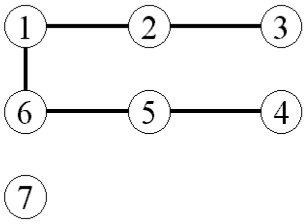
This work is motivated by the following application. Consider a free space optical (FSO) network [14]. Each node has a set of D laser transmitters and D receivers. If nodes i and j are in transmission range of each other, a transmitter from i can be pointed to a receiver at j and a transmitter from j can be pointed to a receiver at i, thereby creating a *bidirectional* optical communication link between i and j. If i and j are within transmission range of each other, we say that a *potential link* exists between them. If there is a potential link between i and j and they each have an unused transmitter/receiver pair, then an *actual link* can be formed between them. We will refer to a transmitter/receiver pair on a node as an *interface*. Thus, a potential link can be converted to an actual link if each of the nodes has an available interface.

We consider a sequential topology control (design) problem for a FSO network. The network initially consists entirely of potential links. Requests arrive for communication between pairs of nodes. Suppose that shortest path routing is used. When a request arrives for communication between nodes v_s and v_t, a current topology exists that consists of the actual links that have thus far been created, along with the remaining potential links. We wish to find a shortest path in this topology consisting of actual and potential links with the property that any potential link on the path can be converted to an actual link. This means that if the path contains a potential link from node i to node j, i and j must each have a free interface. Therefore, when searching for a shortest path, we can delete all potential links that are incident on a node that has no free interfaces. However, deleting these potential links still does not reduce the routing problem to a conventional shortest path problem. This is because if the path contains a pair of consecutive potential links (i, j), (j, k), the intermediate node j must have at least two free interfaces[7].

As an example, suppose the current topology is given in Figure 1(a) where the solid lines are actual links and the dotted lines are potential links. Suppose each node has a total of two interfaces. If a request arrives for a shortest path between nodes 1 and 7, the degree constraint rules out the path $1-6-7$ because node 6 has only one free interface. The degree constrained shortest path from 1 to 7 is $1-2-3-4-5-6-7$.



(a) Example Network



(b) Shortest Path Tree

Fig. 1. Degree Constrained Shortest Path Problem

In addition to showing that the degree constrained shortest path problem cannot be reduced to a conventional shortest path problem by deleting potential links incident on nodes without free interfaces, the example illustrates two other features of this problem. Firstly, the union of the set of constrained shortest paths originating at a node need not form a tree rooted at that node. For all shortest paths from node 1, other than to node 7, we can construct a shortest path tree as shown in Figure 1(b). However, node 7 cannot be added to this tree. Secondly, since the constrained shortest path from 1 to 6 is the single hop path $1-6$ and not $1-2-3-4-5-6$, it follows that a sub-path of a constrained shortest path need not be a (constrained) shortest path.

The problem is formally described as follows. We have a network with two kinds of edges (links), 'actual' and 'potential'. We refer to these edges as *green* and *red* edges respectively. We denote green edges by S and red edges by R. Let $E = S \cup R$, where E is the entire edge set of the graph. We are required to find the shortest path from v_s to v_t. Edge e_{ij} denotes the edge connecting v_i and v_j. The weight of edge e_{ij} is w_{ij}. Green edges S represent actual edges, and red edges R represent potential edges. We denote a path from v_s to v_t by $P_t = \{v_s, v_1, \cdots, v_t\}$ and its length $\mid P_t \mid$. The problem is to find a shortest path P_t^* from a source node v_s to a destination node v_t. However, each vertex has an degree constraint that limits the number of actual edges incident to it by D. If there are D green edges already incident to a vertex, then no red edges incident to it may be chosen, and all such edges can safely be removed from the graph. If the number of green edges is $\leq D-2$ then we can choose up to two red edges incident to this vertex. In this case, a shortest path is essentially unconstrained by this vertex, as it can choose up to two red edges. The main difficulty arises when a vertex already has $D-1$ green edges incident to it. In this case, at most one red edge incident to this vertex may be chosen. Hence, if a shortest path were to pass through this vertex, the shortest path must choose at least one green edge incident to this vertex.

In this paper, we study algorithms for the *Degree Constrained Shortest Path problem*. We propose two algorithms for finding a shortest path with degree constraints. First, we show how to compute a shortest paths between a pair of vertices by employing a perfect matching algorithm. However, in some cases we wish to compute single source shortest paths to all vertices. In this case, the matching based algorithm is slow as we have to run the algorithm for every possible destination vertex. The second algorithm is significantly faster and extends Dijkstra's shortest path algorithm when all edge weights are non-negative. The rest of this paper is organized as follows. In Section 2, we show how to use a perfect matching algorithm to compute the shortest path. The complexity of this algorithm is $O(|E|^2 \log |E|)$ from a source to a single destination. In Section 3 we propose an alternate shortest path algorithm by extending Dijkstra's algorithm. We introduce the degree constrained shortest path algorithm in Section 4. Section 5 analyzes and compares the complexity of these algorithms. Its complexity is $O(|E||V|)$ from a source not only to one destination but to all destinations. Finally, we conclude the paper in Section 6. *We would also like to note that* *even though we describe the results for the version where all the nodes have the same degree constraint of D, it is trivial to extend the algorithm to the case when different nodes have different degree constraints.*

While the shortest path problem with degree constraints has not been studied before, considerable amount of work has been done on the problems of computing bounded degree spanning trees for both weighted and unweighted graphs. In this case, the problems are NP-hard and thus the focus of the work has been on the design of approximation algorithms [4, 10, 3]. In addition, Gabow and Tarjan [6] addressed the question of finding a minimum weight spanning tree with one node having a degree constraint.

2 Perfect Matching Approach

One solution for the degree constrained shortest path problem is based on a reduction to the minimum weight perfect matching problem. We define an instance of a minimum weight perfect matching problem as follows. Each node v has a constraint that at most δ_v red edges can be incident on it from the path. If node v has D green edges incident on it, then $\delta_v = 0$. When node v has $D-1$ green edges, then $\delta_v = 1$; otherwise, $\delta_v = 2$. When $\delta_v = 0$ we can safely delete all red edges incident to v. We now reduce the problem to the problem of computing a minimum weight perfect matching in a new graph G'. For each vertex $x \in V - \{v_s, v_t\}$ we create two nodes x_1 and x_2 in G', and add a zero weight edge between them. We retain v_s and v_t as they are.

For each edge $e_{xy} \in E$ we create two new vertices $v_{e_{xy}}$ and $v_{e'_{xy}}$ (called edge nodes) and add a zero weight edge between them. We also add edges from $v_{e_{xy}}$ to x_1 and x_2, each of these has weight $w_{xy}/2$. When $x = v_s$ or $x = v_t$, we simply add one edge from $v_{e_{xy}}$ to x. We also add edges from $v_{e'_{xy}}$ to y_1 and y_2, with each such edge having weight $w_{xy}/2$. Finally, for any red edges (u, x) and (x, y) with $\delta_x = 1$ we delete the edges from $v_{e'_{ux}}$ to x_1 and from $v_{e_{xy}}$ to x_1.

Theorem 1. *A minimum weight perfect matching in G' will yield a shortest path in G connecting v_s and v_t with the property that for any vertex v on the path, no more than δ_v red edges are incident on v.*

The running time of the minimum weight perfect matching algorithm is $O(|V'|(|E'|+|V'|\log|V'|))$ [5][2]. Since $|V'|$ is $O(|V|+|E|)$ and $|E'|$ is $O(|E|+|V|)$ we get a running time of $O(|E|^2 \log|E|)$ (we assume that $|E| \geq |V|$, otherwise the graph is a forest, and the problem is trivial).

3 Shortest Path Algorithm

In this section, we develop an algorithm for finding degree constrained shortest paths using an approach similar to Dijkstra's shortest path algorithm.

3.1 Overview of the Algorithm

In Dijkstra's algorithm, shortest paths are computed by setting a label at each node. The algorithm divides the nodes into two groups: those which it designates as permanently labeled and those that it designates as temporarily labeled. The distance label d of any permanently labeled node represents the shortest distance from the source to that node. At each iteration, the label of node v is its shortest distance from the source node along a path whose internal nodes are all permanently labeled. The algorithm selects a node v with a minimum temporary label, makes it permanent, and reaches out from that node, i.e., scans all edges of the node v to update the distance labels of adjacent nodes. The algorithm terminates when it has designated all nodes as permanent.

Let Z denote the set of nodes v_x satisfying $\sum_{e_{xy} \in S} e_{xy} = D - 1$. In other words, Z is the set of nodes where there is only one interface available for edges in R. Hence any shortest path passing through $v_x \in Z$ must exit on a green edge if it enters using a red edge. If the shortest path to v_x enters on a green edge, then it can leave on any edge. We start running the algorithm on the input graph with red *and* green edges. However, if a shortest path enters a vertex using a red edge, and the node is in Z then we mark it as a "critical" node. We have to be careful to break ties in favor of using green edges. In other words, when there are multiple shortest paths, we prefer to choose the one that ends with a green edge. So a path terminating with a red edge is used only if it is strictly better than the shortest known path. A shortest path that cuts through this node may not use two red edges in sequence. Hence, we only follow edges in S (green edges) out of this critical vertex. In addition, we create a *shadow* node v'_x for each critical node v_x. The shadow node is a vertex that has directed edges to all the red neighbors of v_x, other than the ones that have been permanently labeled. The shadow node's distance label records the length of a valid (alternate) shortest path to v_x with the constraint that it enters v_x on a green edge. (Note that the length of this path is strictly greater than the distance label of v_x, due to the tie breaking rule mentioned earlier.)

Let C denote the set of critical nodes. When $v_c \in C$, then only edges in S leaving v_c may be used to get a valid shortest path through v_c. To reach a neighbor v_j such that $e_{cj} \in R$, v_c needs an alternate shortest path p^+_{sc}, where parent$(v_c) = v_{p'}, v_{p'} \in p^+_{sc}$ and $e_{p'c} \in S$. We re-define the degree constraint shortest path problem as a shortest path problem which computes shortest valid paths for all nodes in V, and alternate shortest valid paths for nodes in C.

In fact, the edges from v'_c to the red neighbors of v_c are *directed* edges so they cannot be used to obtain a shortest path to v'_c. The corresponding critical node v_c retains its neighbors with green edges, and the red edge to its parent.

The set *ancestor(v_i)* is the set of all vertices on a shortest valid path from v_s to v_i, including the end-nodes of the path. Let V be the set of vertices in the graph, and V' be the set of shadow vertices that were introduced.

Definition 1. *Let $v_i, v_j \in V \cup V'$ and $v_c \in C$ with $(e_{ij} \in S) \vee (e_{ij} \in R \wedge v_i, v_j \notin C)$. A node pair $(v_i, v_j) \in gateway(c)$ if $v_c \in ancestor(v_i)$, and $v_c \notin ancestor(v_j)$.*

This means that there always exists a possible alternate path to a critical node through a gateway node pair since P_j^* (shortest path from source to j) does not include the critical node v_c. If v_i's neighbor v_j is a critical node and it is connected to it with a red edge, then this node pair cannot be a legal gateway node pair. A similar problem occurs when v_i is critical and the edge connecting them is red.

A virtual link $e'_{jc} = (v_j, v'_c)$ is created for inverse sub-path $(p^*_{ci} \cup \{e_{ij}\})^{-1}$, where $(v_i, v_j) \in gateway(c)$ (p^*_{ci} is the portion of the shortest path P_i^*). With shadow nodes and virtual links, we can maintain the data structure uniformly.

When nodes are labeled by the algorithm, it is necessary to check if these nodes form a gateway node pair or not. For this purpose, we define a data structure called CL, the Critical node List. For all nodes, $v_c \in CL(i)$ if $v_c \in ancestor(v_i)$ and $v_c \in C$. So, $CL(i)$ maintains all critical node information along the shortest path from v_s to v_i. By comparing $CL(i)$ and $CL(j)$, we know a node pair (v_i, v_j) is a gateway pair for v_c if ($v_c \in CL(i)$ and $v_c \notin CL(j)$).

3.2 Algorithm

The degree constrained shortest path algorithm using critical node sets is developed in this sub-section. The algorithm itself is slightly involved, and the example in the following section will also be useful in understanding the algorithm. In fact, the algorithm modifies the graph as it processes it. We also assume that the input graph is connected. Step 1 is to initialize the data structure. The difference with Dijkstra's algorithm is that it maintains information on two shortest paths for each vertex according to link type. Step I.1 initializes the data structure for each node. We use *pred* to maintain predecessor (parent node) information in the tree. Step I.2 is for the initialization of the source. The label value is set to 0. Step I.3 is for the initialization of the permanently labeled node set (P) and the queue for the shortest path computation (Q).

Step 2 consists of several sub-steps. First, select a vertex node with minimum label in Q (Step 2.1) and permanently label it (Step 2.2). The label of each node $d[y]$ is chosen as the minimum of the green edge and red edge labels. If the green edge label is less than or equal to the red edge label, we select the path with the green edge and its critical node set (Step 2.3.1). Otherwise, we choose a red edge path (Step 2.4.1). If the path with the red edge is shorter than that with the green edge and the number of green edges incident on v_y is $D-1$ ($v_y \in Z$), then v_y becomes a critical node (Step 2.4.2.1). In Step 2.5, we define CL; its initial value is the same as its parent node. After the node is identified as a critical node, then CL will be changed later (Step 2.6.2).

The decision of whether a node is a critical node or not is made when a node is permanently labeled. When a node v_y is identified as a critical node, a shadow node v'_y is created as shown in Steps 2.6.3 through 2.6.6. In Step 2.6 we also choose the neighbors according to the node type. A critical node has neighbors with green edges and its shadow node has directed edges to the neighbors to which the critical node had red edges, if the neighbor has not been permanently labeled as yet (Step 2.6.7). Otherwise, the node can have neighbors with any type of edges (Step 2.7.1). However, Step 2.7.1 specifies an exception to not add neighbors to which there is a red edge if the neighbor is a critical node. Consider a situation when a non-critical node v_i has a red edge to a critical node v_c. Since v_c is a critical node, v_c is already permanently labeled. Note that v_i cannot be a neighbor of v_c, but a neighbor of v'_c, by definition. Later, when v_i is permanently labeled and is a non-critical node, we do not wish to have v_c in v_i's list. For this reason, we check if a neighbor node with a red edge is a critical node or not. Since v_i is permanently labeled, there is no shorter path in the graph including path through v'_c. All legal neighbor information is maintained by data structure *adj*. Note that the graph we construct is actually a *mixed* graph with directed and undirected edges, even though the input graph was undirected.

Step 2.8 examines the neighbors of a permanently labeled node. It consists of two parts. Step 2.8.3 updates labels for all neighbor nodes in *adj*. This procedure is similar to Dijkstra's algorithm. The only difference is that label update is performed based on link type. Step 2.8.4 is the update for shadow nodes by using procedure *UpdateSNode*. If v_y cannot be a part of the shortest path for its neighbor v_x, then we should check if it can be a gateway node pair or not. If so, we should update labels of all possible shadow nodes. Step P.1 considers only permanently labeled nodes in order to check if it can be a gateway node pair. The reason is that we cannot guarantee that the computed path for shadow nodes through its (temporarily labeled) neighbors would be shortest path since the path for temporarily labeled node could be changed at any time. So, shadow nodes in two different sub-trees are considered at the same time. Steps 2.8.4.1.1 and 2.8.4.1.2 compute the path for shadow nodes for all critical nodes along the path P^*_x and P^*_y. We finally delete v_y from the Q (Step 2.9).

Comments:
$d[x] = \min(d[x][green], d[x][red])$
$d[y'][red] = \infty$ for all shadow nodes $v_{y'}$
Z is the set of nodes with only one free interface
P is the set of permanently labeled nodes
Q is a Priority Queue with vertices and distance labels
C is the set of critical nodes
V' is the set of shadow nodes

I.0 Procedure *Initialize*
I.1 for each $v_x \in V$
I.1.1 $d[x] \leftarrow d[x][green] \leftarrow d[x][red] \leftarrow \infty$
I.1.2 $pred[x] \leftarrow pred[x][green] \leftarrow pred[x][red] \leftarrow null$
I.1.3 $CL[x] \leftarrow \emptyset$
I.1 endfor
I.2 $d[s] \leftarrow d[s][green] \leftarrow d[s][red] \leftarrow 0$
I.3 $P, V', C \leftarrow \emptyset$ $Q \leftarrow V$
I.0 End Procedure

Degree Constrained Shortest Path Algorithm

INPUT : $G = (V, E)$ and source v_s and $E = S \cup R$

OUTPUT: $G' = (V \cup V', E')$ with $pred$ giving shortest path information

1 call $Initialize$
2 while NotEmpty(Q)
2.1 $v_y \leftarrow \arg_{v_x \in Q} \min d[x]$ Exit if $d[y] = \infty$
2.2 $P \leftarrow P \cup \{v_y\}$
2.3 if $d[y][green] \leq d[y][red]$ then // v_y's shortest path enters on a green edge//
2.3.1 $pred[y] \leftarrow pred[y][green]$
2.4 else //v_y's shortest path enters using a red edge //
2.4.1 $pred[y] \leftarrow pred[y][red]$
2.4.2 if $v_y \in Z$ then //v_y has only one free interface //
2.4.2.1 $C \leftarrow C \cup \{v_y\}$
2.4.2 endif
2.3 endif
2.5 if $pred[y]$ is not null then
2.5.1 $CL[y] \leftarrow CL[pred[y]]$ //copy critical list from parent //
2.5 endif
2.6 if $v_y \in C$ then // processing a critical node //
2.6.1 $adj[y] \leftarrow v_x, \forall v_x, e_{xy} \in S$ //add green neighbors //
2.6.2 $CL[y] \leftarrow CL[y] \cup \{v_y\}$
2.6.3 $V' \leftarrow V' \cup \{v'_y\}$ // create a shadow node //
2.6.4 $Q \leftarrow Q \cup \{v'_y\}$
2.6.5 $d[y'] \leftarrow d[y'][green] \leftarrow d[y'][red] \leftarrow \infty$
2.6.6 $pred[y'][green] \leftarrow pred[y'][red] \leftarrow null$
2.6.7 $adj[y'] \leftarrow v_x, \forall v_x, e_{yx} \in R \wedge v_x \notin P$
2.7 else if $v_y \in V$ then // processing a non-critical node //
2.7.1 $adj[y] \leftarrow \{v_x | (e_{yx} \in S) \vee (e_{yx} \in R \wedge v_x \notin C)\}$
2.7.2 $adj[y] \leftarrow adj[y] \cup \{v'_x | (e_{yx} \in R \wedge v_x \in C \wedge v'_x \in P)\}$
2.6 endif
2.8 for $\forall v_x \in adj[y]$
2.8.1 if $e_{yx} \in S$ then index $\leftarrow$ green
2.8.2 else if $e_{yx} \in R$ then index $\leftarrow$ red
2.8.1 endif
2.8.3 if $d[x][index] > d[y] + w_{yx},\ v_x \notin P$ then
2.8.3.1 $d[x][index] \leftarrow d[y] + w_{yx}$
2.8.3.2 $pred[x][index] \leftarrow \{v_y\}$
2.8.3.3 $d[x] \leftarrow \min(d[x][green], d[x][red])$
2.8.4 else
2.8.4.1 if $v_x \in P$ then
2.8.4.1.1 call $UpdateSNode(v_y, v_x)$
2.8.4.1.2 call $UpdateSNode(v_x, v_y)$
2.8.4.1 endif
2.8.1 endif
2.8 endfor
2.9 Delete $[Q, v_y]$
2 endwhile

P.0 Procedure $UpdateSNode(v_m, v_n)$
P.1 for each $v_i \in CL[m] - CL[n]$ and $v_i' \notin P$
P.1.1 if $d[i'][green] > d[n] + w_{nm} + d[m] - d[i]$ then
P.1.1.1 $d[i'][green] \leftarrow d[n] + w_{nm} + d[m] - d[i]$ //encodes a path from v_n to v_i//
P.1.1.2 $pred[i'][green] \leftarrow \{v_n\}$
P.1.1.3 $CL[i'] \leftarrow CL[n]$
P.1.1.4 $d[i'] \leftarrow d[i'][green]$
P.1.1.5 $d[i'][red] \leftarrow \infty$
P.1.1.6 $E' \leftarrow E' \cup \{e'_{ni'}\}$
P.1.1 endif
P.1 endfor
P.0 End Procedure

4 Detailed example

Consider the graph shown in Figure 2. The source vertex is v_s. Our goal is to compute shortest valid paths from the source to all vertices in the graph. We now illustrate how the algorithm computes shortest valid paths and shortest alternate paths (for critical nodes). Suppose that the nodes in $Z = \{v_c, v_f, v_b, v_a\}$, and we can pick at most one red edge incident to any of these nodes in a shortest path.

Initially, we have $Q = \{v_s, v_c, v_b, v_d, v_f, v_a, v_e\}$. The first row of the table shows the distance labels of each vertex in V when we start the while loop in Step 2. In fact, we show the status of the queue and the distance labels each time we start a new iteration of the while loop. In the table, for each node we denote the shortest path lengths ending with a green/red edge as x/y.

Iteration	v_s	v_c	v_b	v_d	v_f	v_a	v_e	v_c'	v_b'	v_f'
1 ($v_y = v_s$)	0/0	∞/∞	∞/∞	∞/∞	∞/∞	∞/∞	∞/∞			
2 ($v_y = v_c$)	0/0	$\infty/5$	$\infty/9$	$7/\infty$	∞/∞	∞/∞	∞/∞			
3 ($v_y = v_d$)	0/0	$\infty/5$	55/9	$7/\infty$	∞/∞	$15/\infty$	∞/∞	∞/∞		
4 ($v_y = v_e$)	0/0	$\infty/5$	55/9	$7/\infty$	∞/∞	$15/\infty$	$8/\infty$	∞/∞		
5 ($v_y = v_b$)	0/0	$\infty/5$	11/9	$7/\infty$	∞/∞	$15/\infty$	$8/\infty$	∞/∞		
6 ($v_y = v_b'$)	0/0	$\infty/5$	11/9	$7/\infty$	∞/∞	$15/\infty$	$8/\infty$	$59/\infty$	$11/\infty$	
7 ($v_y = v_a$)	0/0	$\infty/5$	11/9	$7/\infty$	∞/∞	15/16	$8/\infty$	$59/\infty$	$11/\infty$	
8 ($v_y = v_c'$)	0/0	$\infty/5$	11/9	$7/\infty$	∞/∞	15/16	$8/\infty$	$26/\infty$	$11/\infty$	
9 ($v_y = v_f$)	0/0	$\infty/5$	11/9	$7/\infty$	$\infty/31$	15/16	$8/\infty$	$26/\infty$	$11/\infty$	
10 ($v_y = v_f'$)	0/0	$\infty/5$	11/9	$7/\infty$	$\infty/31$	15/16	$8/\infty$	$26/\infty$	$11/\infty$	∞/∞

1. (Iteration 1): $Q = \{v_s, v_c, v_b, v_d, v_f, v_a, v_e\}$
   $v_y = v_s$. Add v_s to P. Since v_s is not critical, we set $pred[s] = null$ in Step 2.3.1. $CL[v_s] = \emptyset$. We define $adj[s] = \{v_d, v_c, v_b\}$ in Step 2.7.1. In Step 2.8 we now update the distance labels of v_d, v_c, v_b. Since none of these nodes are in P, we do not call procedure $UpdateSNode$. The updated distance labels are shown in the second row.
2. (Iteration 2): $Q = \{v_c, v_b, v_d, v_f, v_a, v_e\}$
   $v_y = v_c$. Add v_c to P. Since v_c is critical (shortest path enters using a red edge, and $v_c \in Z$) we add v_c to C. We define $pred[c] = v_s$. We also define $CL[c] = \{v_c\}$. Since v_c is critical, in Step 2.6 we define $adj[c] = \{v_a, v_b\}$. Note that we do not add

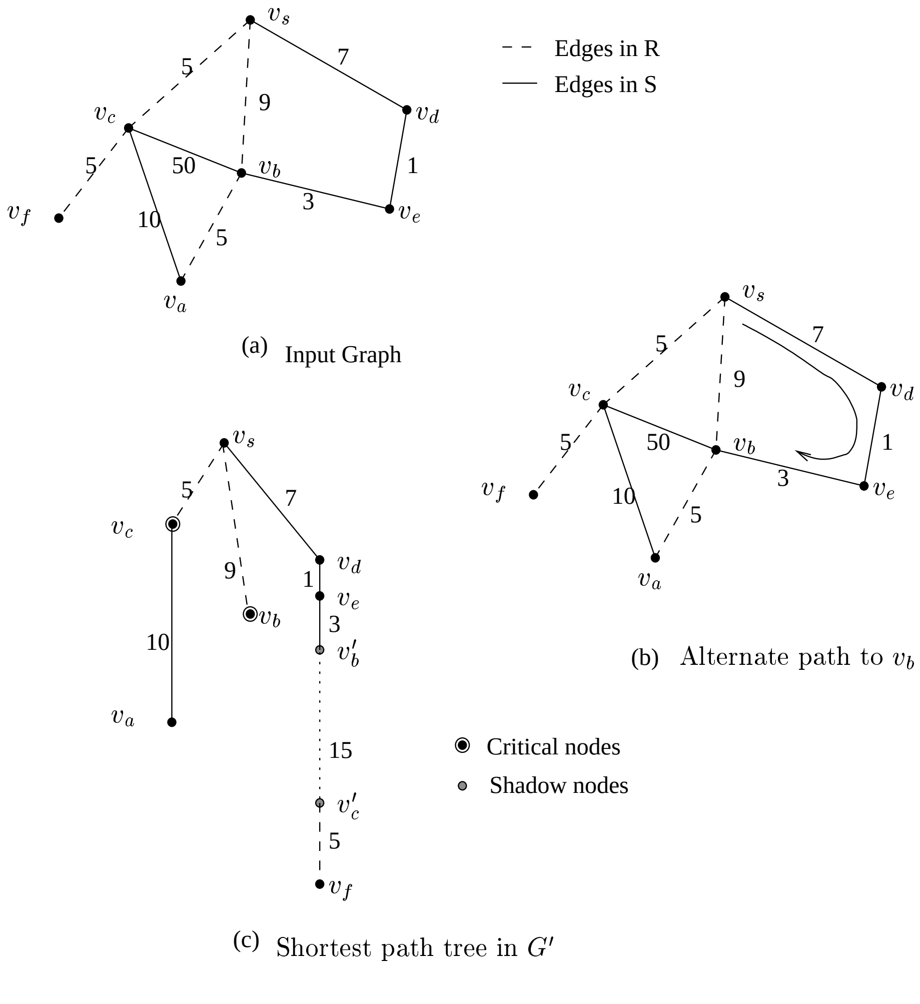


Fig. 2. Example to illustrate algorithm.

v_f to $adj[c]$ since the edge (v_c, v_f) is in R. We also create a shadow node v'_c and add it to Q and V'. We define $adj[c'] = \{v_f\}$. We now update the distance labels of v_a and v_b in Step 2.8. The updated distance labels are shown in the third row.

3. (Iteration 3): $Q = \{v_b, v_d, v_f, v_a, v_e, v'_c\}$
$v_y = v_d$. Add v_d to P. Define $pred[d] = v_s$. $CL[d] = \emptyset$. In Step 2.7.1 we define $adj[d] = \{v_e\}$. We update the distance label of v_e in Step 2.8 The updated distance labels are shown in the fourth row.
4. (Iteration 4): $Q = \{v_b, v_f, v_a, v_e, v'_c\}$
$v_y = v_e$. Add v_e to P. We define $pred[e] = v_d$. $CL[e] = \emptyset$. In Step 2.7.1 we define $adj[e] = \{v_b\}$. We update the distance label of v_b in Step 2.8. The updated distance labels are shown in the fifth row.
5. (Iteration 5): $Q = \{v_b, v_f, v_a, v'_c\}$
$v_y = v_b$. Add v_b to P. Since v_b is critical (shortest path enters using a red edge, and $v_b \in Z$) we add v_b to C. We define $pred[b] = v_s$. $CL[b] = \{v_b\}$. Since v_b is critical, in Step 2.6 we define $adj[b] = \{v_c, v_e\}$. We also create a shadow node v'_b and add it to Q and V'. We define $adj[b'] = \{v_a\}$. In Step 2.8, since both v_c and v_e are in P we call $UpdateSNode(v_b, v_c)$, $UpdateSNode(v_c, v_b)$ and $UpdateSNode(v_b, v_e)$ $UpdateSNode(v_e, v_b)$. Consider what happens when we call $UpdateSNode(v_b, v_c)$, $UpdateSNode(v_c, v_b)$. We update the distance label of v'_b to 55. We also define $pred[b'] = v_c$. At the same time we update the distance label of v'_c to 59 and define $pred[c'] = v_b$. Consider what happens when we call $UpdateSNode(v_b, v_e)$ $UpdateSNode(v_e, v_b)$.
We update the distance label of v'_b to 11 and re-define $pred[b'] = v_e$. The updated distance labels are shown in the sixth row.
6. (Iteration 6): $Q = \{v_f, v_a, v'_c, v'_b\}$
$v_y = v'_b$. Add v'_b to P. We define $pred[b'] = v_e$. $CL[b'] = \emptyset$. Recall that $adj[b'] = \{v_a\}$. (Since this is a shadow node, note that it is not processed in Step 2.6 or Step 2.7). In Step 2.8 we let $v_x = v_a$. However, this does not affect $d[a]$ which has value 15, even though it updates $d[a][red]$. The updated distance labels are shown in the seventh row.
7. (Iteration 7): $Q = \{v_f, v_a, v'_c\}$
$v_y = v_a$. Add v_a to P. Define $pred[a] = v_c$ and $CL[a] = \{v_c\}$. Since $v_a \in V$ and is not critical, in Step 2.7.1 we define $adj[a] = \{v_c\}$. In Step 2.7.2, we add v'_b to $adj[a]$. In Step 2.8, we make calls to $UpdateSNode(v_a, v_c)$, $UpdateSNode(v_c, v_a)$ and $UpdateSNode(v_a, v'_b)$, $UpdateSNode(v'_b, v_a)$. The first two calls do not do anything. The second two calls update $d[c']$ to be $11 + 5 + 15 - 5 = 26$ (Step P.1.1.1). We also define $pred[c'][green] = v'_b$. The updated distance labels are shown in the eighth row.
8. (Iteration 8): $Q = \{v_f, v'_c\}$
$v_y = v'_c$. We add v'_c to P and this node is not critical. We define $pred[c'] = v'_b$. We also define $CL[c'] = \emptyset$. Recall that $adj[c'] = \{v_f\}$. (Since this is a shadow node, note that it is not processed in Step 2.6 or Step 2.7). In Step 2.8 we update $d[f] = 31$. The updated distance labels are shown in the ninth row.
9. (Iteration 9): $Q = \{v_f\}$
$v_y = v_f$. We add v_f to P. This node is identified as critical since it is in Z. We also define $CL[f] = \emptyset$. We create a shadow node v'_f. However, v_f has no green neighbors so $adj[f] = \emptyset$. We add v'_f to V' and $adj[f'] = \emptyset$. The updated distance labels are shown in the tenth row.
10. (Iteration 10): $Q = \{v'_f\}$
We exit the loop since $d[f'] = \infty$.

Due to space limitations, we omit the proof of the algorithm and the complexity analysis completely.

Acknowledgments

We thank Julian Mestre and Azarakhsh Malekian for useful comments on an earlier draft of the paper.

References

1. Ravindra K. Ahuja , Thomas L. Magnanti, James B. Orlin, "Network Flows: Theory, Algorithms and Applications", *Prentice Hall*, 1993.
2. W. Cook, A. Rohe, "Computing Minimum Weight Perfect Matchings", *INFORMS Journal of Computing*, 1998.
3. S. Fekete, S. Khuller, M. Klemmstein, B. Raghavachari and N. Young, "A Network-Flow technique for finding low-weight bounded-degree spanning trees", *Journal of Algorithms*, Vol 24, pp 310–324 (1997).
4. M. Fürer and B. Raghavachari, "Approximating the minimum degree Steiner tree to within one of optimal", *Journal of Algorithms*, Vol 17, pp 409–423 (1994).
5. H. N. Gabow, "Data structures for weighted matching and nearest common ancestors with linking", *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, pages 434–443, 1990.
6. H. N. Gabow and R. E. Tarjan, "Efficient algorithms for a family of matroid intersection problems", *Journal of Algorithms*, Vol 5, pp 80-131 (1984).
7. P. Gurumohan, J. Hui "Topology Design for Free Space Optical Network", *ICCCN '2003*, Oct. 2003
8. Z. Huang, C-C. Shen, C. Srisathapornphat and C. Jaikaeo, "Topology Control for Ad Hoc Networks with Directional Antennas", *ICCCN '2002*, Miami, Florida, October 2002.
9. A. Kashyap, K. Lee, M. Shayman "Rollout Algorithms for Integrated Topology Control and Routing in Wireless Optical Backbone Networks" *Technical Report*, Institute for System Research, University of Maryland, 2003.
10. J. Könemann and R. Ravi, "Primal-dual algorithms come of age: approximating MST's with non-uniform degree bounds", *Proc. of the 35th Annual Symp. on Theory of Computing*, pages 389–395, 2003.
11. S. Koo, G. Sahin, S. Subramaniam, "Dynamic LSP Provisioning in Overlay, Augmented, and Peer Architectures for IP/MPLS over WDM Networks", *IEEE INFOCOM* , Mar. 2004.
12. K. Lee, M. Shayman, "Optical Network Design with Optical Constraints in Multi-hop WDM Mesh Networks", *ICCCN'04*, Oct. 2004.
13. E. Leonardi, M. Mellia, M. A. Marsan, "Algorithms for the Logical Topology Design in WDM All-Optical Networks", *Optical Networks Magazine*, Jan. 2000, pp. 35- 46.
14. N.A.Riza, "Reconfigurable Optical Wireless", *LEOS '99 , Vol.1*, Nov. 1999, pp. 8-11.