The Improved Sliding Shortest Path Algorithm

Ramesh Bhandari
Laboratory for Telecommunications Sciences
College Park, Maryland 20740
rbhandari@ieee.org

Abstract

Given an undirected weighted graph, and a pair of vertices, s and t, connected by the shortest path, and an edge pq not lying on the shortest path, what are the minimal edge weight changes required within the given graph to cause the shortest path between s and t to pass through edge pq? This is the type of problem often faced by network administrators in the telecommunication world. In this paper, we provide an improvement to an existing algorithm called the *Sliding Shortest Path Algorithm* to solve such a problem; the approach taken is one of including negative edge weight changes, not considered in the previous version. Allowing for negative edge weight changes then augments the parameter-search space, leading to fewer edge weight changes required to achieve the objective of rerouting network traffic over the path that includes edge pq. The algorithm easily extends to the case of constraining the shortest path to include a given vertex p, instead of an edge pq, by simply collapsing the edge pq into a single vertex p.

Keywords: shortest path, algorithm, weighted graph, undirected, sliding, constrained, rerouting, network, minimal, edge weight changes, optimization

1 Introduction

In telecommunication networks, it is often desired to change the link weights to adjust the traffic flow on the links to alleviate congestion and increase the throughput [1-4]. The traffic flow is assumed to occur within the network along the shortest paths, computed from the current values of link weights. Assignment of link weights and adjustment of their values also occur in the design of networks, where the given traffic demands (traffic for different source-destination pairs) are routed along certain given paths, which then become the paths of least cost upon appropriate link weight assignments [5-8]. The latter

CONGRESSUS NUMERANTIUM **203** (2010), pp. 175-192

type of problem is called the "The Inverse Shortest Path Problem". In all of the above problems, several demands are taken into account at the same time.

In a recent paper [9], we addressed the novel problem of rerouting a single demand (single source-destination pair traffic) via link weight changes, and provided an algorithm, called the *Sliding Shortest Path (SSP) algorithm*, to achieve rerouting. The traffic for the single demand was rerouted through a given link or a node, not already on the shortest path. The requirement within the problem was to make link weight changes on as few links as possible to minimize the convergence time of routes in the changed network operating under an *Open Shortest Path First (OSPF)* routing protocol [10], and further to make the link weight changes as small as possible to reduce the possibility of other (source-destination) paths changing within the network as well. The link weight changes considered were all positive increments.

In this paper, we consider the same basic problem, but provide an improved version of the earlier *SSP* algorithm by considering negative changes to link weights as well. Allowing for negative weight changes enlarges the parameter-search space, and thus improves the solution by enabling less number of links to be changed. Since our solution to this problem is an improved form of an existing algorithm, we first revisit the original problem that led to the *SSP* algorithm. In Section 2, we define and show the original problem to be NP-hard, and subsequently provide the full development of the heuristics, starting with the *SSP* algorithm with edge cuts (Section 3), which is a precursor to and serves as the foundation for the subsequent development of the SSP Algorithm (with finite positive increments) in Section 4 and the improved version in Section 6. Section 5 provides an analysis of the efficiency of the algorithms developed in Sections 3 and 4.

2 Problem Definition

Let $G=(V,E)$ denote an undirected, weighted graph, representing a bidirectional network (e.g., a single autonomous system); V is the set of vertices (or nodes), and E is the set of weighted edges (or links); weights are positive integers (> 0); we assume loops and multiple edges are absent in the graph; routing of traffic demands from one point to another within the network takes place along single shortest paths. We also make the assumption that graph G is bi-connected (or 2-connected), i.e., there always exists a simple path, which connects a given pair of vertices, s and t, and includes a given arc pq [11]. A simple path refers to a path in which a given vertex is not visited more than once. Unless otherwise stated, in what follows, the term *path* refers to a simple path.

Let $SP(s,t)$ denote the shortest path between a given pair of vertices s and t in the given graph $G = (V,E)$; let w_i (a positive integer) denote the weight of edge $i \in E$. Let Γ_a denote a set of edges in the given graph $G = (V,E)$, whose weights are increased to alter the given shortest path $SP(s,t)$. Let x_i, $i = 1,\ldots,\Gamma_a$ denote the corresponding weight increments. The problem to solve can be stated as:

Given an undirected, weighted graph $G = (V,E)$, and a pair of vertices $s, t \in V$, and an edge $pq \in E$,

$$minimize\ |\Gamma_a| \qquad (1)$$

subject to arc pq (or arc qp) $\in SP(s,t)$,

$$minimize\ x_i,\ i = 1,\ldots, M \qquad (0 < x_i \leq \infty) \qquad (2)$$

subject to arc pq (or arc qp) $\in SP(s,t)$; $M = |\Gamma_M|$, the result obtained in Eq. (1) above.

The primary objective (Eq. (1)) is to identify the minimum cardinality link set whose weights should be incremented to alter the given route to include edge pq; the secondary problem (Eq. (2)) is a set of multi-objective functions to minimize the increments of the weights of the edges found in Eq. (1).

We now state and prove an important theorem:

Theorem 1: The problem in Eq. (1) is equivalent to the following problem:

$$minimize\ |\Gamma_c| \qquad (3)$$

subject to arc pq (or arc qp) $\in SP(s,t)$,

where Γ_c is interpreted to mean a set of edges $\in E$, which, when cut, alters $SP(s,t)$ in the desired manner.

Proof: Let INF denote a large number, simulating "infinity" (e.g., a number > the sum of the weights of all edges in the graph). Let Γ_M denote the smallest set of edges corresponding to the primary objective (Eq. (1)) solution. Let Γ_N denote the solution of the equivalent problem, Eq. (3). Let w_i, $i = 1,\ldots, |\Gamma_M|$, denote the weights of the edges $\in \Gamma_M$. Let each w_i be incremented in a minimal fashion such that rerouting over the constraint edge *pq* is accomplished. Let w_i' denote the new weight assignments. Let Γ_p denote the set of edges of the desired shortest path *P* (which contains the constraint edge *pq*). Then $\Gamma_p \cap \Gamma_M = \emptyset$; otherwise, the desired shortest path can be made shorter by further decrementing w_j', where edge $j \in \Gamma_p \cap \Gamma_M$, which is a contradiction because w_j' is already the smallest new weight determined with minimal increment to the original weight; and therefore cannot be decreased, i.e., edge j ($\in \Gamma_M$) $\notin \Gamma_p$.

Because edge $j \notin \Gamma_p$, incrementing w_j' further does not alter the new shortest path *P*. Therefore, $\forall\, k$ $(1 \le k \le |\Gamma_M|)$, for which $w_k' \neq$ INF, increment $w_k' \ni w_k' =$ INF. In other words, Γ_M is also the minimum cardinality solution set where rerouting is achieved with weights incremented to INF. Therefore, $|\Gamma_M| = |\Gamma_N|$. If the solution to the problem is unique, $\Gamma_M = \Gamma_N$. If the solution to the problem is not unique, Γ_M need not be the same as Γ_N, but $|\Gamma_M| = |\Gamma_N|$; otherwise there is a contradiction. ***End of Proof (Theorem 1).***

Theorem 2: The problem, Eq. (3), is NP- hard.

Proof: Consider the corresponding decision problem (*Problem A*):

Input: A graph $G = (V, E)$ with non-negative edge weights, two specified vertices, s and t in *V*, and a positive integer *m* and an edge $pq \in E$.

Output: "Yes", if there are *m* edges, whose removal makes the shortest path from *s* to *t* pass through edge *pq*.

It is well-known [12] that the following decision problem (*Problem B*) is NP-complete:

Input: A graph $G = (V, E)$ with non-negative edge weights, two specified vertices, *s* and *t* in *V*, and a positive integer *m* and a threshold *h*.

Output: "Yes", if there are *m* edges, whose removal makes the length of the shortest path from *s* to *t* at least *h*.

We reduce *Problem B* to *Problem A* through the following polynomial-time graph transformation: from the given graph *G* for *Problem B*, create a graph *G'* by adding two new vertices *p* and *q* and three new edges: *pq*, *sp* and *tq*. Make sure that the *s-t* path through the edge *pq* has path length[1] equal to *h*. Now if there are *m* edges whose removal forces the *s-t* path to include edge *pq* in graph *G'*, it follows that there are *m* edges whose removal forces the *s-t* path in graph *G* to become at least *h*. That is, *Problem A* is NP-complete, from which it further follows that problem, Eq. (3), is NP-hard. ***End of Proof (Theorem 2).***

Alternatively, to prove Theorem 2, one can start with the optimization counterpart of *Problem B* (finding the minimum cardinality set of edges so that the s-t shortest path is at least of length *h*), which is NP-hard [12], and make a reduction to a similar optimization counterpart (problem, Eq. (3)) by exactly the same graph transformation as above, as was also done recently and independently in [13].

Because of the NP-hard nature of the problem, Eq. (3), we have constructed heuristics, which we describe in the next few sections.

3 The Sliding Shortest Path Algorithm (Using Edge Cuts)

In a given, undirected, weighted graph $G = (V, E)$, this algorithm determines (in accordance with a certain cutting criterion) the minimal cardinality set of edges, which, when cut, force the shortest path between vertices *s* and *t* to include a given constraint edge *pq*. It is assumed that i)) a simple path, connecting vertices *s* and *t*, and including edge *pq*, exists; ii) the shortest path between vertices *s* and *t*, and including edge *pq*, is unique.

Let Γ_{min} denote the desired minimal cardinality set of edges. Let $SP(s,t)$ denote the shortest path between vertices *s* and *t*. Then the following steps determine Γ_{min}:

1) Assign $\Gamma_{min} = \varnothing$.
2) If the initial shortest path between vertices *s* and *t* includes edge *pq*, terminate; otherwise, go to Step 3.
3) Compute the shortest pair of vertex-disjoint paths [14, 15], one path connecting *s* to one end of the constraint edge *pq* (call it *SP1*), and the other connecting *t* to the other end of the constraint edge *pq* (call it *SP2*); the vertex disjoint path algorithms compute paths *SP1* and *SP2* simultaneously and automatically determine whether *SP1* is a connection from *s* to *p* or *s* to *q*; if *SP1* turns out to be a connection from *s* to *p*, *SP2* is a connection from *t* to *q*, and vice versa; denote their lengths by $l(SP1)$ and $l(SP2)$, respectively; length of a path is defined as the sum of the weights of the edges comprising the path; $l(P_f) = l(SP1)$

[1] If there is already an s-t path of length equal to h in G, then set $h = h - \varepsilon$ in G' to break the tie, where ε is an infinitesimally small number.

$+ w_{pq} + l(SP2)$, where P_f denotes the desired $SP(s,t)$ path, which includes edge pq; this path is composed of path $SP1$, edge pq, and path $SP2$; w_{pq} is the weight of the edge pq. $l(P_f)$ is determined in the original graph and is a value fixed throughout the algorithm.

4) Initialize $i = 1$.
5) Assign $\Gamma(i) = \varnothing$, where $\Gamma(i)$ denotes the ith set of edges, whose weights are changed.
6) Find the first edge of $SP(s,t)$, which does not overlap with $SP1$ *(cut-edge selection criterion)*; cut this edge from the graph; denote this edge by γ.
7) Set $\Gamma(i) = \Gamma(i) \cup \gamma$; if $i = 2$ and $|\Gamma(i)| > |\Gamma_{min}|$, terminate, and go to Step 12.
8) Compute new $SP(s,t)$ in the modified graph.
9) If the new path does not contain edge pq, go to Step 6; otherwise, check for another path (not containing pq) whose length is equal to $l(P_f)$:
    a. Increment w_{pq} by 1.
    b. Compute new $SP(s,t)$.
    c. If $l(SP(s,t)) > l(P_f)$, decrement w_{pq} by 1 and go to Step 10; if $l(SP(s,t)) = l(P_f)$, decrement w_{pq} by 1, and go to Step 6.
10) Set $\Gamma_{min} = \Gamma(i)$; set $i = i + 1$.
11) If $i < 3$, repeat Steps 5-9 in the original graph, replacing $SP(s,t)$ with $SP(t,s)$, and $SP1$ with $SP2$; otherwise, terminate; $SP(t,s)$ denotes the shortest path from t to s, which is taken to be $SP(s\ t)$ in the reverse order.
12) If $|\Gamma(1)| < |\Gamma(2)|$, set $\Gamma_{min} = \Gamma(1)$; if $|\Gamma(2)| < |\Gamma(1)|$, set $\Gamma_{min} = \Gamma(2)$; if $|\Gamma(1)| = |\Gamma(2)|$, set $\Gamma_{min} = \Gamma(1)$ or $\Gamma(2)$.

If the $SP(s,t)$ path already contains edge pq, the algorithm terminates, returning $\Gamma_{min} = \varnothing$; otherwise it performs two runs of an iterative process. The iterative process consists of trimming the graph (cutting one edge at a time) until the shortest path between s and t "slides" over the given constraint edge pq. In the first run of the iterative process (Steps 1-9), an edge to cut in a given iteration (Step 6) is determined by comparing the s to t shortest path $SP(s,t)$ with path $SP1$ and finding the first edge of $SP(st)$, which does not overlap with $SP1$. This edge is then removed from the graph and $SP(s,t)$ is recomputed in the modified graph; the process of comparing and cutting the first non-overlapping edge and re-computing the shortest path is repeated until $SP(s,t)$ passes through the constraint edge pq (Steps 6–9). When this happens, the first run of the iterative process terminates. Note that in Step 9, a check is made to ensure there is no other path of the same length as the desired path, P_f.

In the second run of the iterative process (i=2), *SP2* acts as the reference path and the first non-overlapping edge of *SP(t,s)* is cut in each iteration. The two runs of the iterative process of the algorithm yield two cut-sets, Γ(1) and Γ(2), which can be different. In Step 12, the desired set Γ is identified with the one, which has less links. If there is a tie, Γ is set equal to either of the two. An early termination rule applies in the second iterative process (Step 7) if its solution is worse than the one from the first iterative process.

The iterative process in each run converges without ever disconnecting the source vertex s from the termination vertex t (see Theorem 4 below). In the first run (i=1), the final shortest path from s to t that passes over edge pq is found to be given by

SP(s,t) = *SP1* + arc pq (or qp) + SP2',

where *SP2'* = *SP2* in the reverse order (see Theorem 3 below); the two choices for the *SP(s,t)* solution are due to the fact that edge pq can be traversed in the direction p to q or q to p.

In a similar way, when the iterative process is repeated for i=2, the final shortest path is given by

SP(t,s) = *SP2* + arc qp (or pq) + *SP1'*,

where *SP1'* = *SP1* in the reverse order.

Below we provide proofs:

Theorem 3: In a given graph $G = (V, E)$, the shortest path from s to t over the constraint edge pq (if it exists) is comprised of the edge pq (traversed along the arc pq or arc qp) and the shortest pair of vertex-disjoint paths, one path connecting s to p (or q) and the other connecting q (or p) to t.

Proof: Let us assume that a path from s to t passing through the constraint edge pq exists. The existence of a path from s to t that passes over the constraint edge pq necessarily implies that it is made up of either a path from s to p, arc pq, and a path from q to t, or a path from s to q, arc qp, and a path from p to t. Since any computed shortest path between a pair of vertices in a given graph has to be a simple path, the paths s to p and q to t (or alternatively s to q and p to t) must necessarily be vertex-disjoint, i.e., have no vertices in common.

Furthermore, because the computed path over edge pq is a shortest path, the paths s to p (or q) and q (or p) to t must necessarily comprise the shortest pair of vertex-disjoint paths; shortest means that the sum of the individual lengths of the two paths is the smallest among all possible pairs of vertex-disjoint paths. ***End of Proof (Theorem 3).***

For computation of shortest pair of vertex-disjoint paths, see Refs. [14, 15].

Theorem 4: In the given algorithm, the process of cutting one edge at a time until the shortest path from *s* to *t* slides over edge *pq* does not disconnect *t* from *s*, i.e., the algorithm always converges to a feasible solution.

***Proof*:** During each iteration of the iterative process (e.g., in the first run of the algorithm), when an edge is cut,

1) *Path SP1 remains intact:* The edge that is cut off is the first edge of *SP(s,t)* that does not overlap with *SP1*. Therefore, it does not belong to the set of edges that comprise *SP1*. Consequently, *SP1* remains intact.
2) *Path SP2' remains intact*: The cut edge emanates from *SP1*. It also cannot belong to the set of edges that comprise *SP2'*, because *SP2* is vertex-disjoint from *SP1*, i.e., *SP2'* is at least one edge apart from *SP1*. Therefore, like *SP1*, *SP2'* remains intact.
3) *Edge pq is not cut*: When the shortest path is first found to contain edge *pq* (the first non-overlapping edge of path *SP(s,t)*), the algorithm either terminates or the shortest path *SP(s,t)* changes to one that does not include edge *pq*. Therefore, edge *pq* is never cut.

Because *SP1* and *SP2'*, computed at the outset, remain unaffected during the iterative process, and edge *pq* is never cut, a path always exists from *s* to *t* via edge *pq*. Because a path over edge *pq* from *s* to *t* is always available during the iterative process, the algorithm will always terminate. ***End of Proof (Theorem 4).***

Theorem 5: Path *SP(s,t)* after termination of the algorithm comprises *SP1*, SP2', and edge *pq* (arc *pq* or *qp*)

Proof: At termination, *SP(s,t)* passes through edge *pq*. By Theorem 3, this final *SP(s,t)* must comprise the shortest pair of vertex-disjoint paths in the final truncated graph and edge *pq* (arc *pq* or *qp*). Because paths *SP1* and SP2' exist in this (final) truncated graph (see proof, Theorem 4) and *SP1* ad SP2' comprise the shortest pair of vertex-disjoint paths in the original graph, *SP1* and SP2' must necessarily be the shortest pair of vertex-disjoint paths in this final truncated graph (which is a sub-graph of the original graph). ***End of Proof (Theorem 5).***

In a similar way, proof is constructed for the case when *SP(t,s)* is compared with *SP2*. The cut-set obtained for this case can be different.

Example

Refer to Figure 1. Assume s = A and t = H, and p=B and q=C. *SP(s,t)* = ADFGH of length = 1 + 2 + 1 + 3 = 7; it does not include the constraint edge pq. *SP1* = ADFB and *SP2* = HC comprise the shortest pair of vertex-disjoint paths, one connecting A to B and the other connecting H to C. $l(SP1)$= 1 + 2 + 3 = 6, $l(SP2)$ = 6; w_{pq} = 2. $l(P_f) = l(SP1) + w_{pq} + l(SP2)$ = 14. The first part of the algorithm uses path *SP1* as the reference path. *SP(s,t)* deviates from *SP1* at vertex F and the first non-overlapping edge of *SP(s,t)* is edge FG. Upon deleting this edge from the graph, the recomputed *SP(s,t)* is ADFBGH. This path deviates from *SP1* at vertex B again, and the first non-overlapping edge is BG. Upon deleting this edge, we find that the new, computed *SP(s,t)* path is ADFH; here the first edge of ADFH that deviates from *SP1* is FH. Upon deleting this edge and recomputing *SP(s,t)*, we find *SP(s,t)* in the modified graph is ADFBCH, which traverses the desired edge BC.

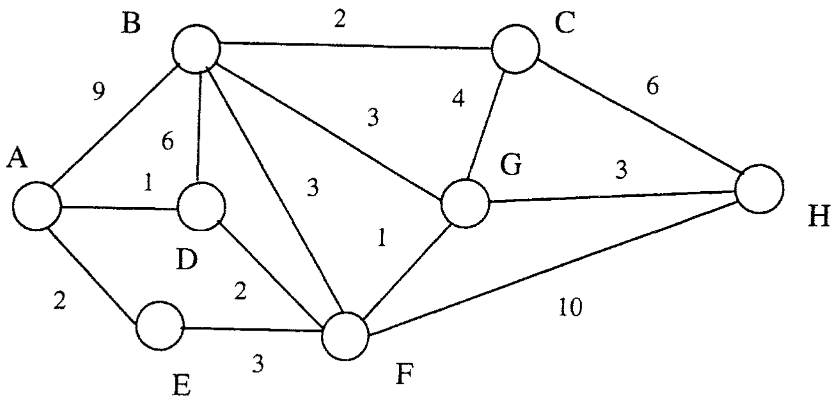

Figure 1 A graph of 8 vertices and 12 edges with their weights indicated.

At this point (Step 9), we check to see if there is another path of the same length, so we change w_{BC} to w_{BC} = 2 + 1 = 3, and recompute the shortest path, which turns out to be the same: ADFBCH. However, $l((SP(s,t))$ (= 15) > $l(P_f)$ (=14), which implies that the first part of the algorithm corresponding to the shortest path from s to t (*SP(st))*, terminates; there is no other path of the same length as the length of the desired shortest path. Reset w_{BC} to its original value of 2. $\Gamma(1)$ =

{FG, BG, FH}, which comprises the three edges that were cut to force *SP(s,t)* to include edge BC.

Using *SP(t,s)* = HGFDA and *SP2* = HC as the comparison path in the second part of the algorithm, the first non-overlapping edge to cut is HG. This leads to *SP(t,s)* = HFDA in the truncated graph. This path deviates from *SP2* at vertex H, so the first non-overlapping edge HF is cut. At this point, there are two possibilities for the new shortest path: i) HCGFDA ii) HCBFDA. These are of equal length (=13). Let us suppose that the shortest path algorithm (e.g., the Dijkstra algorithm) selects path HCBFDA, which includes the desired edge BC.

We now invoke Step 9 to test if there is another path of the same length. Changing w_{BC} to 2 +1 =3 leads to the shortest path being HCGFDA of length = 14; i.e., $l(SP(s,t))$ (= 14) = $l(P_f)$ (=14). We decrement w_{BC} back to w_{BC} = 2, and apply Step 6, which yields edge CG as yet another edge to be cut. The new computed, *SP(t,s),* after the cut is now HCBFDA, the desired path. Invoking Step 9 at this point yields a path with $l(SP(s,t))$ (= 15) > $l(P_f)$ (=14). The second run of the algorithm terminates at this point. $\Gamma(2)$ = {GH, FH, CG}. $|\Gamma(2)|$ = $|\Gamma(1)|$ =3; either solution may be selected, although the tie may be broken by additional criteria such as the number of shortest paths between other pairs of vertices impacted, and so forth.

4 Sliding Shortest Path Algorithm (Using Finite Weight Increments)

This algorithm is identical to the previous algorithm, except that, instead of cutting the first non-overlapping edge (see Step 6 of the algorithm in Section 3), one increments its weight:

$$x_j = l(P_f) - l(P_j) + e,$$

where x_j is the weight increment for the first non-overlapping edge encountered in the *j*th iteration of Steps 6-9 of the algorithm; $l(P_j)$ is the length of the corresponding shortest path (*SP(s,t)* or *SP(t,s),* as the case may be); e is an infinitesimally small positive number. For the integral weights, e = 1. The increment defined above is the minimal amount needed to make path P_j greater (in length) than the desired path P_f; as a result, the latter becomes the shortest path in the final (modified) graph.

In Figure 1, for the case of p = B and q = C, changing l_{FG} from 1 to 9, l_{BG} from 3 to 6, and l_{FH} from 10 to 12 (in accordance with the above formula) changes *SP(s,t)* from ADFGH to ADFBCH in the first run of the algorithm; changing l_{GH}

from 3 to 11, l_{FH} from 10 to 12, and l_{CG} from 4 to 5 (in accordance with the above formula) changes *SP(t,s)* from HGFDA to HCBFDA in the second run of the algorithm.

5 Efficiency (or Time-Complexity)

In Steps 1-9 of the algorithm described in Section 3, Step 2 is $O(|V|^2)$, the efficiency of a shortest path algorithm, like the Dijkstra algorithm [16]. Step 3, which computes the shortest pair of vertex-disjoint paths is also $O(|V|^2)$. Steps 6-9 are repeated as many times as the edges are cut or modified in weights. Step 6, which compares two paths is of $O(|V|)$. Step 8 (computation of shortest paths) is $O(|V|^2)$, and Step 9 is at most of $O(|V|^2)$. The time-complexity of the Dikstra algorithm, which is of $O(|V|^2)$, dominates in Steps 6-9 and the overall efficiency of the Sliding Shortest Path algorithm (denoted by K) is determined by the efficiency of the Dijkstra algorithm times the number of times (denoted by η) it has to be run , i.e, K is $O(\eta|V|^2)$.

The worst-case scenario corresponds to the case when almost all the edges within the given graph (excluding those that comprise paths *SP1*, *SP2* and edge *pq*) are cut or assigned new weights. As $|V|$ increases, this number η is of $O(|E|)$. Below we provide the algorithm efficiency for the various graph types:

1) Dense Graphs: $|E|$ is $O(|V|^2) \Rightarrow$ K (algorithm efficiency) is $O(|V|^4)$ (worst-case scenario).

2) Sparse Graphs (almost tree-like): $|E|$ is $O(|V|) \Rightarrow$ K (algorithm efficiency) is $O(|V|^3)$ (worst-case scenario).

Telecommunication networks are sparse and on the average, the number of edges to cut is $O(\kappa)$, where κ is the edge-connectivity, i.e., the maximum number of edge-disjoint paths that exist between the given vertices s and t (or equivalently, the minimum number of edge cuts that disconnect s from t). The order of magnitude of κ, for sparse networks, is therefore unity, leading to K being of $O(|V|^2)$, on the average. Note that, because the algorithm efficiency is dependent upon the efficiency of the Dijkstra algorithm, the algorithm efficiency is further improved with more efficient implementations of the Dijkstra algorithm (see, e.g., Ref. [16]).

6 The Sliding Shortest Path Algorithm (Finite Weight Changes (Positive and/or Negative))

When negative weight changes are permitted, they are restricted by the fact that the lowest allowed weight on any edge is unity. Consequently, in Eq. (2), the lower bound of 0 is replaced by $-w_i$, where w_i is the weight of the edge i. Because x_i's can be negative, the problem in Eq. (1) is no longer reducible to the problem in Eq. (3), which separated earlier from Eq. (2) as an independent problem. The objective in Eq. (2) also changes now to one of minimizing the absolute value of x_i. Eq. (1) and Eq. (2) are very much interrelated, making the solution of the combination of Eq. (1) with Eq. (2) (in its new form) even more intractable. Therefore, we resort again to heuristics and give below a version of a heuristic, which is based on an extension of the previous algorithms:

In a given, undirected, weighted graph $G = (V, E)$, this heuristic determines a minimal cardinality set of edges (primary objective), with minimal edge weight changes (positive or negative) such that the shortest path between vertices s and t now includes a given constraint edge pq. It is assumed that i) a simple path, connecting vertices s and t, and including edge pq, exists; ii) the shortest path between vertices s and t, and including edge pq, is unique; iii) the modified graph after weight changes has non-zero positive weights.

Allowing for multiple solutions, let $\Gamma_{\min}(j)$, $j = 1,\ldots n$, denote the n possible solutions (of the same cardinality) for the required set of edges. Let $SP(s,t)$ denote the shortest path between vertices s and t. Then the following steps determine $\Gamma_{\min}(j)$ and the individual edge weight changes:

1) Initialize n=1; if the initial shortest path $SP(s,t)$ between vertices s and t contains edge pq, terminate with $n = 1$ and $\Gamma_{\min}(1) = \varnothing$; otherwise, go to Step 2.
2) Compute the shortest pair of vertex-disjoint paths, one path connecting s to one end of the constraint edge pq (call it $SP1$), and the other connecting t to the other end of the constraint edge pq (call it $SP2$); the vertex disjoint path algorithms compute paths $SP1$ and $SP2$ simultaneously and automatically determine whether $SP1$ is a connection from s to p or s to q; if $SP1$ turns out to be a connection from s to p, $SP2$ is a connection from t to q, and vice versa; denote their lengths by $l(SP1)$ and $l(SP2)$, respectively; the length of a path is defined as the sum of the weights of the edges comprising the path; $l(P_f) = l(SP1) + w_{pq} + l(SP2)$, where P_f denotes the desired $SP(s,t)$ path, which includes edge pq; path P_f is composed of path $SP1$, edge pq, and path $SP2$; w_{pq} is the weight of the edge pq. $l(P_f)$ is determined in the original graph and is a value fixed throughout the algorithm.

3) Initialize $i = 1$.
4) Assign $|\Gamma_{min}(1)| = \infty$.
5) Assign $\Gamma = \varnothing$.
6) Define $\delta = l(P_f) - l(SP(s,t)) + 1$, where $l(SP(s,t))$ denotes the length of the current shortest path, $SP(s,t)$.
7) Determine $\{P_f\} \cap \{SP(s,t)\}$, the set of edges common to path P_f and the current shortest path $SP(s,t)$. Define set $I = \{P_f\} - \{P_f\} \cap \{SP(s,t)\}$. Denote by σ the sum of the weights of the edges $\in I$. If $\sigma - |I| < \delta$, go to Step 13.
8) Find a set (denoted by Γ_m) of minimal cardinality formed from the set I, such that the edge weight decrements within the set yield a path just shorter than $SP(s,t)$; a total decrement equal to δ (determined in Step 6) is required. We determine Γ_m by first selecting the edge $\alpha \in I$, with the largest weight and allowing for the maximum possible negative edge weight change of $-w_\alpha + 1$, where w_α denotes the weight of edge α, and then selecting the next largest weight edge from I, decrementing its weight also as much as possible (in the same manner), and continuing until the decrements add up to δ in this edge weight decremental process. Let $l'(P_f)$ denote the new length of path P_f at the end of this decremental process; $l'(P_f) = l(P_f) - \delta$. Assign $\Gamma_t = \Gamma_m$. Compute $SP(s,t)$. If $l(SP(s,t)) = l'(P_f)$, go to Step 10; otherwise, go to Step 9a.
9) a) Find the first edge of $SP(s,t)$, which does not overlap with path SP1; denote this edge by ρ; denotes its weight by w_ρ.
   b) Set $\Gamma_t = \Gamma_t \cup \rho$; set $w_\rho = w_\rho + \delta'$ (calculated as in Step 6, but with $l(P_f)$ replaced with $l'(P_f)$).
   c) Compute $SP(s,t)$ in the modified graph.
   d) If $l(SP(s,t)) \neq l'(P_f)$, go to Step 9a; otherwise, go to Step 10.
10) Check for another path (not containing edge pq) whose length is equal to $l'(P_f)$:
    a. Increment w_{pq} by 1 and compute $SP(s,t)$.
    b. If $l(SP(s,t)) > l'(P_f)$, decrement w_{pq} by 1, go to Step 11; if $l(SP(s,t)) = l'(P_f)$, decrement w_{pq} by 1 and go to Step 9a.
11) Define $\Gamma_C = \Gamma \cup \Gamma_t$. If $|\Gamma_C| < |\Gamma_{min}(1)|$, delete all previous solutions and set $\Gamma_{min}(1) = \Gamma_C$; if $|\Gamma_C| = |\Gamma_{min}(1)|$, set $n = n + 1$ and $\Gamma_{min}(n) = \Gamma_C$.
12) Reset any link weights changed in Steps 8 and 9 to their previous values, and recompute $SP(s,t)$ and δ (defined in Step 6).
13) Find the first edge of $SP(s,t)$, which does not overlap with path SP1; denote this edge by γ; denotes its weight by w_γ.
14) Set $\Gamma = \Gamma \cup \gamma$; set $w_\gamma = w_\gamma + \delta$; if $|\Gamma| > |\Gamma_{min}(1)|$, go to 17.
15) Compute $SP(s,t)$ in the modified graph.
16) If the new path does not contain edge pq and $|\Gamma| < |\Gamma_{min}(1)|$, go to Step 6; if the new path does not contain edge pq and $|\Gamma| = |\Gamma_{min}(1)|$, go to

Step 17. If $l(SP(s,t)) = l(P_f)$, check for another path (not containing pq) whose length is equal to $l(P_f)$:

a. Increment w_{pq} by 1 and compute new $SP(s,t)$.

b. If $l(SP(s,t)) > l(P_f)$, decrement w_{pq} by 1. If $|\Gamma| < |\Gamma_{min}(1)|$, delete all previous solutions and set $\Gamma_{min}(1) = \Gamma$; otherwise, set $n = n + 1$, set $\Gamma_{min}(n) = \Gamma$ and go to Step 17; if $l(SP(s,t)) = l(P_f)$, decrement w_{pq} by 1, compute δ (defined in Step 6) and go to Step 13.

17) $i = i + 1$.

18) If $i < 3$, reset the edge weights to the weights in the original graph, replace $SP(s,t)$ with $SP(t,s)$, and $SP1$ with $SP2$ ($SP(t,s)$ denotes the shortest path from t to s, which is taken to be $SP(s, t)$ in the reverse order), set $\Gamma = \varnothing$, compute δ, and go to Step 13; otherwise, terminate.

Here the parameter-search space is expanded to include negative weight changes as discussed earlier. A solution with negative weight changes is explored for in the beginning via Steps 7 and 8. The idea is to make the desired path P_f shortest by decrementing the weights of the edges of the non-overlapping part of the desired final path: $\{P_f\} - \{P_f\} \cap \{SP(s,t)\}$. The process of decrementing the weights starts with the edge with the largest weight, and proceeds in a descending order in order to minimize the number of edges over which the path length differential δ (Step 6) is spread. That is, the edge weight decrements, when they occur, reduce the length of the desired shortest path P_f by δ. While the path P_f, in this process, is made shorter than the shortest path, $SP(s,t)$, there is no guarantee that P_f is the shortest path in this modified graph because some of the s-t paths that were shorter than path P_f before the weight changes could remain shorter than P_f after these weight changes (which are all negative). However, Steps 9 and 10, which are based on the concepts of the algorithms in Sections 3 and 4, ensure that path P_f remains the shortest path in the modified graph.

A new solution (which is a mix of positive and negative weight changes) is sought for in each subsequent iteration, starting with Step 12. Steps 13-16 are identical to the earlier algorithm (Section 4). A new shortest path is calculated (Step 15) after assigning a positive weight increment. Steps 6-11 are repeated, with the new solution replacing the old solutions, if it is better (reduced cardinality). If the new solution is equally good (solution with the same cardinality), it is added to the set of previous solutions (Step 11).

Each run of the algorithm terminates whenever $|\Gamma| > |\Gamma_{min}(1)|$ (Step 14), or when $|\Gamma| = |\Gamma_{min}(1)|$ and $SP(s,t)$ does not include edge pq (Step 16).

Applying the algorithm to the example of Figure 1, $l(SP(s,t)) = 1 + 2 + 1 + 3 = 7$, $\delta = 8$. $\{P_f\} - \{P_f\} \cap \{SP(s,t)\}$ = {BF, BC, CH}. Steps 8 -10 yield a solution: $\Gamma_{min}(1)$ = {CH(-5), BF(-2), BC(-1)} in Step 11; the numbers in parentheses indicate weight changes, which are negative here. Steps 12, 13, and 14 (which define the start of a new iteration) lead to an increment of 8 in the value of w_{FG}. Recomputed $SP(s,t)$ = ADFBGH (Step 15), which yields a new δ value of 3 (Step 6). Steps 7 and 8 then yield a decrement of w_{CZ} by 3, leading to a solution: {FG(+8), CH(-3)}, which replaces the earlier result for $\Gamma_{min}(1)$ (Step 11). In the next iteration (Steps 12, 13 , and 14), Γ = {FG (+8), BG(+3)}. The recomputed path $SP(s,t)$ = ADFGH (Step 15) does not include edge pq and $|\Gamma| = |\Gamma_{min}(1)|$. So the first run of the algorithm using $SP1$ as the reference path terminates (Step 16), and a new run, using $SP2$ as the reference path, begins via Steps 17 and 18.

Steps 13 and 14 yield Γ = {GH(+8)}; the recomputed $SP(t,s)$ = HFDA (Step 15). Steps 6-8 give $\delta = 2$ and Γ_m = {CH(-2)}; $\Gamma_t = \Gamma_m$ = {CH(-2)}. $l'(P_f) = l(P_f) - \delta = 14 - 2 = 12$. Recomputed $SP(t,s)$ = HCGFDA or HCBFDA; $l(SP(t,s)) = 12$ (= $l'(P_f)$). Step 10a leads to $w_{pq} = 2 + 1 = 3$, and the recomputed $SP(t,s)$ = HCGFDA of length = 12 (=($l'(P_f)$); $w_{pq} = 3 - 1 = 2$. Step 9 gives $\rho = CG$, and $\Gamma_t = \Gamma_t \cup \rho$ = {CH(-2), CG(+1)}. Step 10 gives $l(SP(t,s)) > l'(P_f)$. Step 11 yields $\Gamma_C = \Gamma \cup \Gamma_t$ = {GH(+8), CH(-2), CG(+1)}. $|\Gamma_C| > |\Gamma_{min}(1)|$, so it is ignored. In the next iteration, Steps 12, 13, and 14 give Γ = {GH(+8), FH(+2)}. Step 15 can yield $SP(t,s)$ as HCBFDA or HCGFDA. Suppose $SP(t,s)$ = HCBFDA, which includes edge BC, i.e., $l(SP(t,s)) = l(P_f)$). After incrementing w_{pq} by 1 (Step 16a), $SP(t,s)$ = HCGFDA; $l(SP(t,s)) = l(P_f)$). After decrementing w_{pq} by 1 (Step 16b), Steps 13 and 14 yield Γ = {GH(+8), FH(+2), CG(+1)}. $|\Gamma| > |\Gamma_{min}(1)|$, so the algorithm terminates via Steps 17 and 18, with a single solution: $\Gamma_{min}(1)$ = {FG(+8), CH(-3)}.

The above solution gives fewer links whose weights should be changed as compared to the solutions obtained from the application of the algorithm (positive increments only) given in Section 4. Therefore, it is an improved solution. Note that any solution obtained from the application of this algorithm that includes only positive increments will exactly be the same as the solution obtained from the algorithm in Section 4, as the latter is incorporated into the former.

Efficiency: This algorithm has basically the same number of iterations as the algorithms in Sections 3 and 4, with the difference, however, that, in each iteration, we have additional steps (Steps 6-11), which are needed to expand the parameter search space to include negative values for weight changes. Step 7 is at most $O(|V|^2)$ and Step 8 is $O(|V|)$, but Step 9 is an iterative process (very akin to the algorithms in Sections 3 and 4), nested within the main algorithm.

Denoting by η_1 and η_2 the number of times the shortest path algorithm such as the Dijkstra algorithm has to be run in the main algorithm and within the nested iterative process, the efficiency of the algorithm above is $\eta_1 \eta_2$ $O(|V|^2)$. It is worse than the efficiency of previous versions (Section 5), but still is of polynomial-time efficiency. For sparse telecommunication graphs, η_1 and η_2 are each of order unity, making the algorithm $O(|V|^2)$.

7 Summary and Discussion

In this paper, we have presented an improved version of the earlier heuristics, called the *Sliding Shortest Path Algorithms*, to solve the problem of determining the minimal number of edges whose weights should be changed and by how much minimally. To give a background and to lay the foundation for the improved version, a full development of the earlier versions is given, along with proofs (not given earlier) and efficiency computations, to put them on a solid footing. The aim of these algorithms is to alter the route of a given flow to include a specified edge *pq*; the case of the route including a specified vertex is obtained simply by collapsing the edge *pq* into a single vertex *p*.

The *Sliding Shortest Path Algorithm (using edge cuts)* identifies a set of edges in accordance with a certain edge-cutting criterion to change the shortest path to include a given edge *pq*, while the algorithm in Section 4 (using finite weight increments) is exactly the same algorithm, except that, instead of cutting an edge, it calculates the minimal edge weight increment. Therefore, the set of edges to make increments on is the same set of edges that are cut to achieve the desired rerouting. The improved version in Section 6 expands the parameter search space to include negative edge-weight changes, which are bounded by the requirement that the minimum weight permissible on any edge in the graph is unity.

Because the solution parameter space has been expanded, any solution obtained from the improved version can only get better, as demonstrated by the example of Figure 1. The solution, in general, is an admixture of positive and negative increments; the extreme case is a solution comprising solely negative increments or positive increments. When the solution consists of positive increments only, the solution coincides with the solution obtained using the algorithm in Section 4 (its predecessor). Like the predecessor, the algorithm always converges, finding a feasible solution in polynomial time. Although slightly more involved than its predecessor, the algorithm is still simple, easy to implement, and fast. For sparse networks, it is expected to basically mimic the performance of a shortest path algorithm like the Dijkstra algorithm. The algorithm has been thoroughly tested and its MATLAB implementation has been successfully created and tested on larger graphs.

Note that all versions of the Sliding Shortest Path Algorithm (Sections 3, 4, and 6) accomplish the goal of rerouting the traffic flow of interest over a specific edge or vertex. Algorithm in Section 3, by virtue of cutting edges, is expected to cause maximum chaos in a network; chaos here is measured by the number of other shortest paths affected, i.e., shortest paths for other pairs of vertices in the network, and thus other flows in the network. Algorithms in Sections 4 and 6 cause less chaos as compared to the algorithm in Section 3, but the improved version (section 6) is to be preferred because of the less number of edges to be changed, which implies reduced network convergence time when the algorithm is applied to a real-life network operating under the OSPF protocol. A detailed statistical analysis of the performance of these algorithms as applied to different types/sizes of graphs would be part of any future work.

It is also important to remark that the algorithms can be extended easily when the assumption of uniqueness of the desired (constrained) shortest path P_f between vertices s and t is dropped. Non-uniqueness implies there are multiple *SP1* paths and/or multiple *SP2* paths. This just means that the final t to s path could be different from the final s to t path, although in each case the edge pq would be traversed, as desired.

Furthermore, it should be pointed that the solution to the problem posed in this paper may be further improved by varying the lengths of the reference paths *SP1* and SP2, i.e., by not restricting them to be the shortest pair of vertex-disjoint paths.

Acknowledgments: The proof of *Theorem 2* is based on a suggestion by Prof. Samir Khuller, whose earlier work on the NP-hardness of a related problem (most vital arcs problem) was brought to the author's attention by Prof. Leonard Schulman. The author is also thankful to Sam Weyerman for helpful discussions on NP-completeness and to Ann Cox for her comments.

References

[1] B. Fortz and M. Thorup, *Internet Traffic Engineering by Optimizing OSPF Weights*, Proc. of the 19th IEEE INFOCOM, Tel-Aviv, Israel (2000) pp. 519-528.

[2] B. Fortz and M. Thorup, *Optimizing OSPF/IS-IS Weights in a Changing World*, IEEE Journal on selected areas in communication, **20** (2002) pp.756-767.

[3] B. Fortz, J. Rexford, and M. Thorup, *Traffic Engineering with Traditional IP Routing Protocols, IEEE Communications Magazine*, **40** (2002) pp.118-124.

[4] M. Ericsson, M.G.C. Resende, and P. M. Pardalos, *A Genetic Algorithm for the Weight Setting Problem in OSPF Routing,* J. Combinatorial Optimization, **6** (2002) 299-333.

[5] W. Ben-Ameur and E. Gourdin, "Internet Routing and Related Topology Issues", SIAM Journal of Discrete Mathematics, **17**(1) (2003) pp.18-49.

[6] A. Bley, *Inapproximability Results for the Inverse Shortest Paths Problem with Integer Lengths and Unique Shortest paths,* ZIB Report 05-04

[7] A. Bley, *Finding Small Administrative Lengths for Shortest Path Routing*, Proc. of 2nd International Network Optimization Conference, Lisbon, Portugal (2005) pp.121-128.

[8] P. Nilsson, *On the Inverse Shortest Path Algorithm*, Proc. of the 17th Nordic Teletraffic Seminar (NTS 17), Fornebu, Norway (2004).

[9] R. Bhandari, *The Sliding Shortest Path Algorithms*, Proc. of the 8th Cologne-Twente Workshop on Graphs and Combinatorial Optimization, Paris, France (2009) pp. 95-101.

[10] J. Doyle, *Routing TCP/IP, Volume I*, Cisco Press (2001).

[11] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[12] A. Bar-Noy, S. Khuller, and B. Schieber, *The Complexity of Finding Most Vital Arcs and Nodes,* Technical Report CS-TR-3539, Institute for Advanced Studies, University of Maryland, College Park, MD (1995).

[13] B. Engels and G. Pardella, *A Note on the Complexity of Sliding Shortest Paths,* University of Cologne preprint: zaik2009-594 (unpublished).

[14] R. Bhandari, *Survivable Networks: Algorithms for Diverse Routing*, Kluwer Academic Publishers, 1998.

[15] J. W. Suurballe, *Disjoint Paths in a Network,* Networks **4** (1974) pp.125-145.

[16] M. Gondran and M. Minoux, *Graphs and Algorithms,* John Wiley, 1990.