

# Efficient Update of Shortest Path Algorithms for Network Routing

Bin Xiao, Qingfeng ZhuGe, Edwin H.-M. Sha  
Department of Computer Science  
University of Texas at Dallas  
Richardson, Texas 75083

## Abstract

In this paper, we present a new efficient update of shortest path algorithms for network routing, which make use of the information of the previously computed SPT. Besides efficiency, our algorithm maintain the most routing stability by making minimum changes to the topology of an existing SPT. In order to get the new SPT in the basic algorithm, we introduce a weight-change graph  $G^*$  according to original graph  $G$ . Because of the nonnegative edge in  $G^*$ , we can extend our idea about dynamic update of shortest path tree to a graph with negative edges. To the authors' knowledge, our algorithms are the most efficient algorithms known in the literature.

**Index Terms**—Routing, shortest path tree, OSPF.

## 1 Introduction

With OSPF routing protocols, each link is associated with a cost (weight) and routers exchange link state information so that each router in a routing area (e.g. an OSPF area) has a complete description of the network topology. Using the link costs in a definite area, every router computes a path with minimum cost from itself to every other router, producing a shortest path tree (SPT). When the topology in a local area changes (e.g., a link fails, recovers, or changes its routing cost), each router recomputes its SPT. In most of today's commercial routers, this re-computation is done by deleting the current SPT and recomputing it from scratch by using the well-known Dijkstra algorithm [1].

However, the changes in the link states do not bring too much difference to the topology of the new SPT compared with the old one. Actually, in most cases the new SPT does not change at all. Static algorithms that recompute the SPT from scratch are really inefficient because they do not take advantage of available information about the old SPT. This re-computation

can consume a lot of CPU time and preventing other critical routing functions from being executed. By re-computing a new SPT from scratch, a router may unnecessarily choose a different route of the same minimum distance to forward its packets. That may lead to undesirable fluctuation of traffic load on a given route [2, 3, 4].

In this paper, we present a new search method that eliminates the redundancy between searches at successive iterations of the planning algorithm. Our search method is founded on the ability to efficiently maintain a single-source shortest paths tree embedded in the connectivity graph, subject to the dynamic modifications that result from changes in the link states. These dynamic algorithm use information of the outdated SPT and update only the part of the SPT that is affected by the change. Our design consider two optimization constraints. One is to minimize the computational complexity required to update an SPT. The other is to maintain routing stability by making minimal changes to the topology of an existing SPT. The purpose of our work is restricted to dynamic SPT algorithms that can be used in link-state protocols. The final results show that our algorithms are much better than any previous one. In Section 6, we can see our algorithms have made an half improvement compared with [5] using examples in Section 4 with Table 1. Furthermore, our algorithms can also apply to a graph with negative weight edges to dynamic update shortest path tree.

The remainder of the paper is organized as follows. In Section 2 we introduce graph-theoretic definitions and notations. In Section 3, we describe our algorithm for computing an SPT and prove it. Some examples have been given in Section 4. Section 5 analyzes theoretical bounds on the asymptotic computational complexity of this algorithm. Finally we discuss how our solution improves the efficiency by comparing previous results in Section 6 and present some conclusions about our work in Section 7.

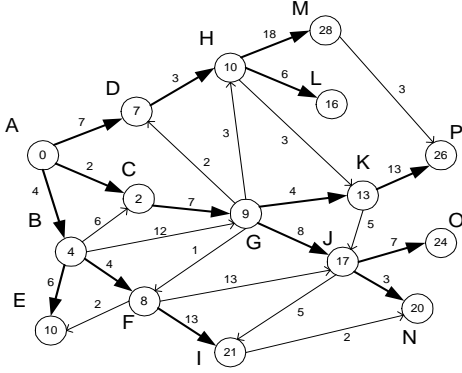


Figure 1: A graph with nodes and edges

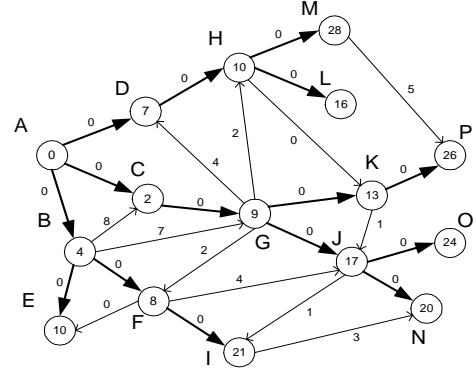


Figure 2: A weight-change graph  $G^*$

## 2 Definition and Notations

### 2.1 Original Graph $G$

We now define some notations to be used in the rest of the paper. Let  $G = (V, E, w)$  denote a directed graph where  $V$  is the set of nodes and  $E$  is the set of edges in the graph. Graph  $G$  contains no negative-weight cycle. Let  $S(G) \in V$  denote the root or source node of  $G$ . An example for a graph is in Figure 1.

We use  $w(e)$  to denote the weight of edge  $e$  for each directed edge  $e \in E$ . If an edge  $e$  is  $i \rightarrow j$ , node  $i$  and node  $j$  denote respectively the source node and the end node of  $e$ . Let  $E(e)$  be the end node of edge  $e$  while  $S(e)$  be the source node. The length or distance of a directed path is the sum of weights of the edges on the path.

A rooted tree  $T$  is a subgraph of  $G$  such that  $S(G)$  is in  $T$  and every node in  $T$  can be reached from  $S(G)$  through a unique directed path using only edges in  $T$ . If an edge  $e \in T$  is  $i \rightarrow j$ , we say node  $i$  is a parent node of  $j$ . We define a node  $i \in T$  with the following attributes:  $P(i)$  is the parent node of  $i$  and  $D(i)$  is the distance attribute of  $i$ . Because  $T$  is a tree, invoking  $P(i)$  recursively determines a unique path from  $S(G)$  to any node in  $T$ .

The descendants of a node  $i$  in  $T$  are all nodes that are reachable by  $i$ . We use  $des(i)$  to denote a subset that includes  $i$  and all descendants of  $i$  in tree  $T$ .

### 2.2 Weight-change graph $G^*$

Here we introduce a weight-change graph  $G^*$ . This graph will help us to understand our algorithms for its good property. The basic algorithm is based on this graph  $G^*$ . If we have a graph  $G = (V, E, w)$  and its

shortest path tree  $T$ , we can get a weight-change graph  $G^*$ .

**DEFINITION 2.1.** For a graph  $G = (V, E, w)$ , let its weight-change graph be  $G^* = (V, E, w^*)$ . For each edge  $e \in E$  is  $i \rightarrow j$  in graph  $G$ , there is one weight-change edge  $e \in E$  from  $i$  to  $j$  and  $w^*(e) = w(e) + D(i) - D(j)$  in graph  $G^*$ .

By the definition of 2.1, we can draw the weight-change graph  $G^*$  in Figure 2, which is corresponding to graph in Figure 1. Both graphs have the same shortest path tree (SPT is shown with bold lines in Figure 1 and Figure 2). During the dynamic algorithms in our basic algorithm, all computations are based on graph  $G^*$  with temporary edge weight and SPT till we find the final SPT. Graph  $G$  and  $G^*$  have the same distance and parent for every node in the SPT.

If we have a node set  $Q$ , we define some edge sets, which have some relationship with  $Q$ . *Source\_part* $\{Q\}$  is an edge set, which includes all edges whose source nodes are in  $Q$  while end nodes are not in. On the contrary, *End\_part* $\{Q\}$  is an edge set, which includes all edges whose end nodes are in  $Q$  while source nodes are not in.

When the weight of one edge  $e$   $i \rightarrow j$  increases or decreases, we use  $w'$  to denote the new weight. Because of this change, if the shortest distance (or parent) from  $S(G)$  to node  $j$  is different from original one  $D(j)$  (or  $P(j)$ ), we should have new values for node  $j$  when algorithms end.

### 3 Algorithms

#### 3.1 Basic Algorithm

We first present a *basic algorithm* that can recompute a SPT from a source node  $S(G)$  to every other node in the graph when the weight of one edge changes. This algorithm deals with weight-change graph  $G^*$ . Suppose we have a static shortest path tree  $T$  for the original graph  $G$ , which can be derived by using a static Dijkstra algorithm. In this tree  $T$ , for each node  $v$ , we have the information of its parent  $P(v)$  and its distance  $D(v)$  from source node  $S(G)$ . During this algorithm,  $T$  and  $P$  update once some nodes changed. When the algorithm ends, we get a new  $T$  and  $P$  for new SPT because of one edge's change.

Before we give the basic algorithm, we present a function  $SELECT\_MIN(B, T)$ .  $B$  is an edge set of graph  $G^*$ .  $T$  is the shortest path tree of graph  $G$ . We try to find the minimum weight edge from edge set  $B$ . If there are several edges whose weights are same, we try to find one node which is the nearest to the source node  $S(G)$  of tree  $T$ .

The basic algorithm is as follows, which includes two parts. One part is for the case when one edge becomes larger. The other part is for the case when one edge becomes smaller. To make time complexity smaller, we separate them to two different cases. Before we execute the basic algorithm, we should have the old SPT and weight-change graph  $G^*$  by Definition 2.1.

**Step1** : wait until the weight  $w^*(e)$  of one edge  $e : i \rightarrow j$  changed to be a new value  $w'^*(e)$ ,

1. if  $w'^*(e) > w^*(e)$  and  $e \in T$  then  $d = w'^*(e) - w^*(e)$ ,

Go to Step 2. /\* case 1: one edge becomes larger \*/

2. else if  $w'^*(e) < 0$  then  $d = w'^*(e)$

Go to Step 3. /\* case 2: one edge becomes smaller \*/,

3. else go to Step 1

**Step2** : /\* when one weight increases \*/

1. Initialize

$Q \leftarrow \text{des}(j)$ ,

$B = \{e | e \in \text{End\_part}\{Q\}, \text{ and } w^*(e) < d\}$ ,

2. if  $B = \emptyset$  then /\* change left nodes \*/

$\forall v \in Q, D(v) \leftarrow D(v) + d$ ,

Update the weight of edges connected to nodes in  $Q$ ,

Go to Step 1.

3. else  $e_1 \leftarrow \text{SELECT\_MIN}(B, T)$ ,  $P(E(e_1)) = S(e_1)$ ,

$Q_1 = \{v | v \in \text{des}(E(e_1)) \text{ and } v \in Q\}$ ,

$\forall v \in Q_1, D(v) = D(v) + w^*(e_1)$ ,

Update the weight of edges connected to nodes in  $Q_1$ ,

$Q = Q - Q_1$ ,  $w(e_1) = 0$  /\* update  $Q$  \*/,

Update  $B$  by adding edges belonging to

$\text{Source\_part}\{Q_1\}$  with weight smaller than  $d$ , and removing edges belonging to  $\text{End\_part}\{Q_1\}$ .

Go to 2.

**Step3** : /\* when one weight decreases \*/,

1. Initialize /\* descendants of node  $j$  update once \*/

$Q_j \leftarrow \text{des}(j)$ ,  $P(j) = i$ ,

$\forall v \in Q_j, D(v) = D(v) - d$ ,

$Q = \emptyset, B_1 = \emptyset$  /\*  $Q$  is a updated node set \*/,

Update all edges connected to nodes in  $Q_j$ ,

$B = \{e | e \in \text{Source\_part}\{Q_j\}, \text{ and } w^*(e) < 0\}$ ,

2. If  $B = \emptyset$  then go to Step 1.

3. else  $e_1 \leftarrow \text{SELECT\_MIN}(B, T)$ ,  $P(E(e_1)) = S(e_1)$

$Q_1 = \{v | v \in \text{des}(E(e_1)) \text{ and } v \notin Q\}$ ,

$\forall v \in Q_1, D(v) = D(v) + w^*(e_1)$ ,

Update the weight of edges connected to nodes in  $Q_1$ ,

$Q = Q + Q_1, w^*(e_1) = 0$  /\* update  $Q$  \*/,

Update  $B_1$  by adding edges belonging to  $\text{Source\_part}\{Q_1\}$  with weight below 0,

Update  $B$  by removing edges belonging to  $\text{End\_part}\{Q_1\}$ ,

If  $B \neq \emptyset$ , then go to 3,

Else  $B = B_1, B_1 = \emptyset$ ,

Go to 2.

To make it much easier for understanding of this algorithm, we give a brief description of how it is executed. At step 1, when the weight of one edge changes, the basic algorithm decides whether the old SPT need to be changed. Only in case 1 and case 2, we need to go to step 2 and step 3 respectively to get a new tree. Otherwise we still wait for another weight change, while remaining the same shortest path tree.

Step 2 deals with the weight of one edge increased. First we initialize a node set  $Q$  and an edge set  $B$ . All nodes that might have a new shortest distance from  $S(G)$  (or a new parent node) are in  $Q$ . All edges in  $B$  are potential edges that could make the shortest distance of nodes in  $Q$  increased less. At this time, we just consider edges whose weight is smaller than  $d$ . Only from these edges nodes in  $Q$  can achieve a distance smaller than the original one plus  $d$ , which is just follow the old SPT. For every update step, we select the minimum weight edge  $e_1$  from  $B$ , which can bring the minimum increase to nodes in  $Q_1$ . Then we update the new distances for these nodes by adding  $w^*(e_1)$ . Also we need to change the weight of edges connected with nodes in  $Q_1$  in graph  $G^*$ . In the end of one update step,  $B$  and  $Q$  should be updated accordingly. This process will not end until  $B = \emptyset$ .

Step 3 deals with the weight of one edge decrease. During the initialization, all nodes in  $\text{des}(j)$  are updated once. Edges in  $B$  are potential elements that could make the shortest distance of nodes outside the

subtree  $\text{des}(j)$  to be smaller. Because we don't know the definite nodes need updated, that nodes have different shortest path from the old SPT are included in node set  $Q$ . Firstly, we update nodes that are neighbor to subtree  $\text{des}(j)$ . we call these nodes  $Q_1$ . All edges from node set  $Q_1$  forms edges set  $B_1$ . We update these edges by adding  $w^*(e_1) - d$  to them. All those negative values are in edge set  $B_1$ . After we execute all edges in  $B$ , we move  $B_1$  to  $B$ ,  $\emptyset$  to  $B_1$  and  $0$  to  $d$ . If  $B = \emptyset$ , the update process ends. Otherwise we do update work from the nearer nodes from subtree  $\text{des}(j)$  to further nodes. The update process is similar to step 2.

### 3.2 Multiple link weight changes

When there are several link weight changes occurring at one time, it is always possible to run the algorithm sequentially for each weight changed. However, some optimization can be achieved by updating several of these changes together. Even though a separate initialization needs to be done for every weight change for *step 2* and *step 3* in the algorithms. First of all, we initialize all the weight increments.  $Q$  includes all nodes that are potential ones whose distances (or parents) need to be changed. Then the update loop body is the same as one weight increases and produce a temporary graph  $G_1^*$ . Second, we update graph  $G_1^*$  for all weight decrements. As before, in the initialization of *Step 3*, we update all descendants for each decreased edge and have one edge set  $B$ . The rest of the algorithm in *Step 3* is executed normally.

### 3.3 Second algorithm

We will show there is still redundancy when we use the basic algorithm for computation of a new SPT. For example, if edge  $(C,G)$  decreases from 0 to -5, which is 7 to 2 in graph  $G$ , the weight of edge  $(G,D)$  changes twice during computation of basic algorithm. One is updating node  $G$  and its connecting edges, which changes its weight to -1. The other is updating node  $D$  and its connecting edges, which changes its weight to 0 for a final result. The redundancy to change edge weight twice is caused by maintaining graph  $G^*$  simultaneously with changed nodes.

To reduce the redundancy discussed above, we introduce the second algorithm. The second one has the same idea as in the basic algorithm. However all computations are based on original graph  $G$ . We do not use graph  $G^*$  any more. Because of the limitation of pages, we do not give a detailed illustration for the second algorithm. The essential idea for the second

algorithm is that all computations are based on graph  $G$  and forming the minimum weight edges  $B$  by Definition 2.1 and do not update the edge weight in graph  $G$ . The other part is the same to the basic algorithm.

## 4 Examples

We will illustrate with a number of figures how the basic algorithms work on a simple network. In these figures, the solid thick arrows between nodes represent the directed edges that are in the current SPT. The numbers in parenthesis illustrate the order of nodes updated.

### 4.1 One edge weight becomes larger

When the weight of an edge  $(C, G)$  changes larger, from 7 to 17 in Figure 1, which is from 0 to 10 in weight-change graph  $G^*$  in Figure 2, a new graph with new shortest path tree is in Figure 3. The result in Figure 3 shows the process for the basic algorithm in *step 2*.

The dotted curve includes the set of nodes  $Q$  that are descendants of node  $G$  in Figure 3, i.e.  $\{G, K, P, J, O, N\}$ . These are the nodes that might have their distance attributes increased as a result of the link change. The nodes outside  $Q$  will be unaffected by the link change. During the first updating step,  $d = 10$ ,  $B = \{(M, P), (H, K), (B, G), (F, J), (I, N)\}$ , the minimum weight edge in  $B$  is  $e_1 : (H, K)$ . For all direct descendants of node  $E(e_1)$ , which are nodes  $K$  and  $P$ , we add  $w(e_1)$  to their *Distance* from  $S(G)$  and make node  $S(e_1)$  as the parent of node  $E(e_1)$  in the new shortest path tree. For all edges connected to nodes  $\{H, K\}$  except in SPT, we update their weight in  $G^*$ . At the end of the first update process, we update  $Q$  by removing nodes  $\{H, K\}$  from it. Also we need to update  $B$  by adding edge  $(K, J)$ , removing edges  $(H, K)$  and  $(M, P)$  from it.

The second update process is done by selecting the minimum weight edge  $(K, J)$  from edge set  $B$  and having the same process described as in the first update step. The last update result is in Figure 3, in which we select the only one edge  $(B, G)$  from edge set  $B$  and update node  $G$ .

### 4.2 One edge weight becomes smaller

When the weight of edge  $(C, G)$  becomes smaller, from 7 to 2, a new graph with new shortest path tree is shown in Figure 4. The result shows the process for the basic algorithm in *step 3*.

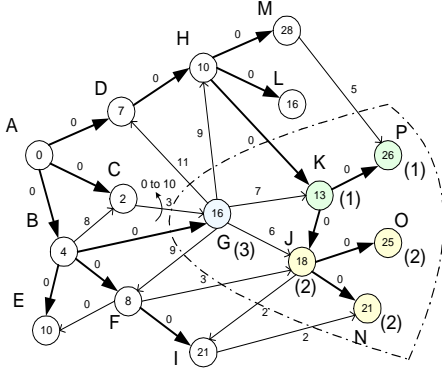


Figure 3: The final result for  $G^*$  with edge  $(C, G)$  changes from 0 to 10

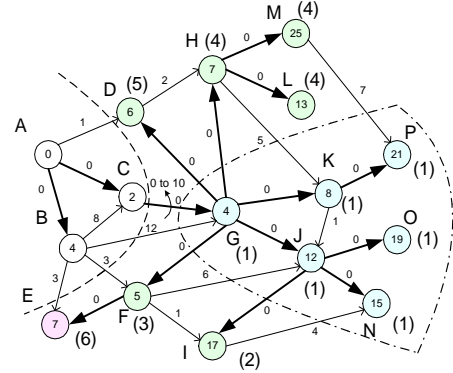


Figure 4: The result for  $G^*$  with the weight of edge  $(C, G)$  changes from 0 to -5

In Figure 4, all nodes which are right to the left dotted line are potential nodes to have a new shortest distances or parent nodes in new SPT. Nodes in the dotted circle can be updated at during the initialization of *step 3*. Also we need to change all edges connected to  $Q_j$  in graph  $G^*$ .  $Q$  is a update node set.  $B$  is an edge set from which we select one minimum weight edge, which is  $(G, D), (G, H), (G, F), (J, I)$  with  $w((G, D)) = -1, w((G, H)) = -3, w((G, F)) = -3, w((J, I)) = -4$  after the update of nodes  $des(G)$ .

The second update process selects one minimum weight edge from node set  $B$ , and does the same procedure as in *step 2*. We add all updated nodes to  $Q$  and count all negative edges to  $B_1$ . This process will not end until  $B = \emptyset$ .  $B_1$  should contain  $(F, E)$  with  $w((F, E)) = -3$  after this update finished. The third update process moves  $B_1$  to  $B$ , does the same procedure as in the second update process. The whole algorithm would not end until  $B_1 = \emptyset$ .

### 4.3 Basic algorithm for a graph with negative weight edges

Our algorithms can also apply to a graph with negative weight edges. In this part we give an example with negative weight edges. However the graph should have no negative-weight cycles reachable from the source node. The original graph is in Figure 5(a). What we want is the new SPT when edge  $(V, U)$  changes its weight from -2 to 4. The weight-change graph is in Figure 5(b) with the weight of edge  $(V, U)$  changes from 0 to 6.

First with static Bellman-Ford we can get the old SPT in *Step 1* in the basic algorithm. The following procedure is the same as the above examples. Edges

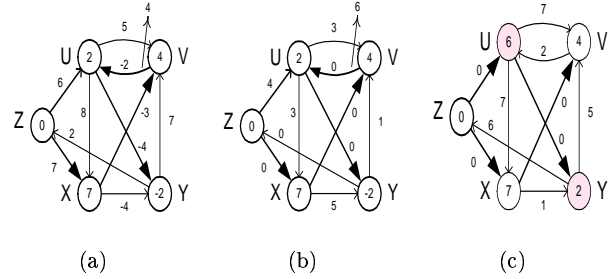


Figure 5: (a)  $G$  with edge  $(V, U)$  changes from -2 to 4 ; (b)  $G^*$  with edge  $(V, U)$  changes from 0 to 6 ; (c) Final result using the basic algorithm

$(Z, U)$  and  $(X, Y)$  forms edge set  $B$ . The minimum weight edge from  $B$  is  $(Z, U)$  with weight value 4. Then the new distances for all descendants of node  $U$  with 4 increased. Thus we update all nodes and the new SPT is in Figure 5(c) (the bold edge is SPT).

## 5 Algorithmic Complexity

Given that there is a weight change of an edge, let  $t$  denote the minimum number of nodes that must change their distance *or* parent attributes (or both) and  $t_p$  the minimum number of nodes that must change their distance *and* parent attributes. So  $t_p$  gives the number of iterations. Note that the parameters  $t$  and  $t_p$  depend only on the topology of the network, the current SPT, and the link state change, but not on the algorithm used. Let  $E_t$  denote the number of edges connected to the update nodes.

When there is only one processor to update all nodes, the complexity should be  $O(t \cdot t_p + E_t)$ . Consider first the case in which we maintain the minimum weight edge set  $B$  as a linear array. For such an implementation, each  $SELECT\_MIN(B, T)$  and update  $B$  takes time  $O(t)$ , and there are  $|t_p|$  such operations. Each changed node is inserted into (or detracted from) set  $Q$  exactly once, so each edge connected to  $Q$  is examined for selecting minimum weight to  $B$  and updating exactly once during the course of the algorithm. Since the total number of edges connected to update nodes are  $E_t$ , there are a total of  $|E_t|$  operations with each taking  $O(1)$  time. The running time of the entire algorithm is thus  $O(t \cdot t_p + E_t)$ .

In [5], there are two dynamic ways to get a new SPT. One is named *First Incremental Method*. Its linear Dijkstra computation time is  $O(t^2 + E_t)$ . The other method is *Second Incremental Method* with linear Dijkstra computation time  $O(t \cdot t_p + \gamma E_t)$ , where  $\gamma$  denotes the *redundancy* factor, which represents the average time that each node is visited by the algorithm. This factor, can take values between 1 and  $t_p$ .

## 6 Discussion

Our solution is optimal in the sense that there is no redundancy in our algorithms. During every update iteration we first search the minimum weight edge from edge set  $B$  and change the shortest path properties for some nodes. This property promises us that whenever we update the distance for one node, its value is the final result. In [5], the author prefers the second incremental method. However, the second incremental method produces some redundancy to achieve a better computation, which can be seen from its time complexity  $O(t \cdot t_p + \gamma E_t)$  and  $\gamma$  denotes the *redundancy* factor.

If we define addition, subtraction, comparison, enqueue and dequeue as operation needs 1 time unit consuming, we can have the computation results for examples in Section 4 for our algorithms compared with the second incremental method in [5]. The result is shown in Table 1. The data in Table is the time consuming. The difference is caused by the sequence of updated nodes. Our algorithms eliminate all redundancy and thus with a better result.

Our algorithms can also be used when one edge deleted, added or one node deleted, added. When one edge is deleted, it is in the case for one edge's weight increasing to infinity; when one edge is added, it is in the case for one edge's weight of infinity decreasing to

Algorithms	one edge larger	one edge smaller
Basic Algorithm	69	79
Second Algorithm	65	69
Second Incremental Algorithm [5]	104	126

Table 1: Time complexity comparison of algorithms

finite. The same cases are for one node deleted and added.

## 7 Conclusion

In this paper we introduced new dynamic algorithms for computing an SPT in a directed graph, which can represent the topology of routers of a local area. However, this algorithm can be extended to a graph with negative edges because we adopt a new weight-change  $G^*$ , which is nonnegative after its conversion. Thus we can have more applications to a graph with any weight edges using our dynamic update algorithms. That helps us to have an efficient update of shortest path tree while remaining the same time complexity. The new dynamic algorithms have theoretical asymptotic complexity bounds that match the best known results.

## References

- [1] E. Dijkstra, "A note two problems in connection with graphs," *Numerical Math.*, vol. 1, pp. 269–271, 1959.
- [2] G. C.Labovitz and F.Jahanian, "Internet routing instability," in *Proc. SIGCOMM'97*, pp. 115–126, Sept.
- [3] V.Paxson, "End-to-end routing behavior in the internet," *IEEE/ACM Trans. on Networking*, vol. 5, pp. 601–615, Oct. 1997.
- [4] W.T.Zaumen and J.J.Garcia-Luna, "Steady-state response of shortest-path routing algorithms," in *Computers and Communications*, pp. 323–332, 1992.
- [5] H.-Y. T. P.Narvaez, Kai-Yeung Siu, "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 734–746, Dec. 2000.