Minimum Dynamic Update for Shortest Path Tree Construction

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

Abstract— **Shortest path tree (SPT) computation is the major overhead for routers using any link-state routing protocols including the most widely used OSPF and IS-IS. Changes of link states are nowadays commonly occurred. It is not efficient and stable for network routing to use traditional static SPT algorithms to recompute the whole SPT whenever a change happens. In this paper, we present new dynamic algorithms to compute and update the SPT with the minimum computational overhead. And routing stability is achieved by having the minimum changes in the topology of an existing SPT when some link states are changed. To the authors' knowledge, our algorithms outperform the best existing ones in the literatures.**

Keywords— **Routing, Shortest path tree, OSPF.**

I. INTRODUCTION

The Internet is growing rapidly in both size and traffic load. The number of routers in a routing domain is becoming larger. And link failures, recoveries or changes also appear more frequently. In the networks using link-state based routing protocols, such as widely used OSPF and IS-IS, each router recompute a new shortest path tree (SPT) rooted from itself on the changes of link state. Most of the commercial routers today do this computation by deleting the SPT and building a new one using static SPT algorithms such as Dijkstra algorithm [1]. As a result, the SPT computation becomes a bottleneck of the employment of high throughput networks, and the size of routing area becomes unnecessarily limited. And routing instability is increased because of redundant updates of routing table entries [2], [3], [4]. Dynamic algorithms for updating SPT have the potential to render a much better performance because usually there are only a few link-state changes at a time. This is achieved by taking advantage of the available information in the original SPT. By reducing the complexity of the SPT computation, and hence eliminating this performance bottleneck, a larger routing area for high-throughput networks is allowed. By removing the redundancy in updating routing table, the routing instability is controlled in a low level. Therefore it is important to study the efficient dynamic SPT algorithms which can significantly improve the computation complexity and reduce the update redundancy.

The problem of dynamically updating a shortest paths tree may directly benefit many research areas, including communication networks, VLSI design, transportation networks, and scheduling in manufacturing shops. In this paper, we present a new algorithmic framework that eliminates the redundancy in SPT computation and keeps the minimum updating of SPT. Our algorithm design achieves two optimization objectives. One is that it minimizes the computation complexity of updating SPT. The other is that it ensures the change in SPT is the minimum. The complexity analysis in section 6 shows that our algorithms achieve significant improvement compared to the algorithms in [5]. Furthermore, our algorithms can be extended gracefully to solve the similar problems in a graph with negative-weight edges.

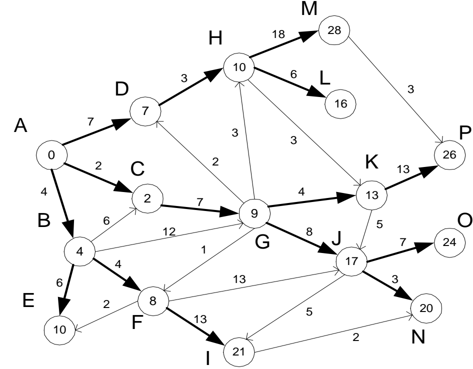

Fig. 1. A graph with nodes and edges

The remainder of the paper is organized as follows. Section II introduces graph-theoretic definitions and notations to be used in the paper. Section III describes our algorithm for computing a new SPT. Some examples have been given in Section IV for better understanding of dynamic algorithm. Section V analyzes theoretical bounds on the asymptotic computational complexity of this algorithm. Then we discuss how our solution improves the efficiency by comparing previous results in Section VI.

II. DEFINITION AND NOTATIONS

A. 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

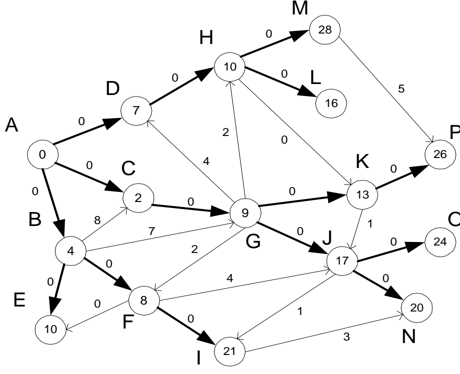

Fig. 2. A weight-change graph G^*

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.

Definition II.1
If i is a node of V in graph G, let $des(i)$ be a node set and $des(i)$ = { v | v = i or v is the descendant of i in the shortest path tree T, $v \in V$ }.

B. 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 II.2
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 II.2, 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.

If we have a node set Q, we define some edge sets, which have some relationship with Q.

Definition II.3
If Q is a node set of $G^* = (V, E, w^*)$, let Source_part{Q} be an edge set of graph G^* and Source_part{Q} ={e | S(e) $\in$ Q, E(e) $\notin$ Q, e $\in$ E}.

Definition II.4
If Q is a node set $G^* = (V, E, w^*)$, let End_part{Q} be an edge set of graph G^* and End_part{Q} ={e | E(e) $\in$ Q, S(e) $\notin$ Q, e $\in$ E}.

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.

III. Algorithms

A. 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)* as in Figure 3. 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.

```
SELECT_MIN(B, T)
{
IF {there are two or more equal minimum weight
    edges in B}
    select one minimum weight edge e1 whose
    end (or source) node is closer to the root;
    remove e1 from B and return(e1);
ELSE
    remove the minimum weight edge e1 from B
    and return(e1);
END
}
```

Fig. 3. A Function for select minimum edge weight from an edge set B

The basic algorithm 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. Before we execute

the basic algorithm, we should have the old SPT and weight-change graph G^* by Definition II.2.

Basic Algorithm:

Step 1: wait until the weight $w^*(e)$ of one 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

Step 2: /* when one weight increases */
1. Initialize
$Q \leftarrow des(j)$,
B={e | e $\in$ End_part{Q}, and $w'^*(e)$<d },
2. if $B = \Phi$ 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 $e_1 \leftarrow$ SELECT_MIN(B,T), $P(E(e_1)) = S(e_1)$,
$Q_1 = \{$v|v $\in des(E(e_1))$ 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 $Source_part\{Q_1\}$ with weight smaller than d, and removing edges belonging to $End_part\{Q_1\}$.
Go to 2.

Step 3: /* when one weight decreases */,
1. Initialize /* descendants of node j update once */
$Q_j \leftarrow des(j)$, $P(j) = i$,
$\forall$ v $\in Q_j$, D(v)=D(v)-d,
Q=Φ, B_1=Φ /* Q is a updated node set */,
Update all edges connected to nodes in Q_j,
B={e | e $\in$ Source_part{Q_j}, and $w^*(e)$<0} ,
2. If $B = \Phi$ then go to Step 1.
3. else $e_1 \leftarrow$ SELECT_MIN(B,T), $P(E(e_1)) = S(e_1)$
$Q_1 = \{$v|v $\in des(E(e_1))$ 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 $Source_part\{Q_1\}$ with weight below 0,
Update B by removing edges belonging to $End_part\{Q_1\}$,
If $B \neq \Phi$, then go to 3,
Else $B = B_1$,B_1= Φ,
Go to 2.

To make it much easier to understand 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 needs 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 $B = \Phi$.

Step 3 deals with the weight of one edge decrease. During the initialization, all nodes in $des(j)$ are updated once. Edges in B are potential elements that could make the shortest distance of nodes outside the subtree $des(j)$ 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 $des(j)$. 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, Φ to B_1 and 0 to d. If $B = \Phi$, the update process ends. Otherwise we do update work from the nearer nodes from subtree $des(j)$ to further nodes. The update process is similar to step 2.

B. Multiple link weight changes

When several link weight changes occur at one time, the easiest way is to run the algorithm sequentially for each weight changed. However, the optimization of computation overhead can be achieved by grouping changes together into two groups: weight increased edges and weight decreased edges. It is obvious that 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. Set 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 produces 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 weight decreased edge and maintain one edge set B. The rest of the algorithm in *Step 3* is executed similarly.

C. 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 to form the minimum weight edge set B with the similar idea in Definition II.2 and do not update the edge weight in graph G.

IV. Examples

We will illustrate how the basic algorithms work on a simple network. In the following figures, the solid thick arrows represent the directed edges of the original SPT. The numbers in parenthesis denote the order of nodes updated.

A. One edge weight becomes larger

When the weight of an edge (C, G) becomes larger from 7 to 17 in Figure 1, which is from 0 to 10 in weight-change graph G^* in Figure 2, the graph with new SPT is shown in Figure 4. The result in Figure 4 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 4, 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 4, in which we select the only one edge (B,G) from edge set B and update node G.

B. 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 5. The result shows the process for the basic algorithm in *step 3*.

In Figure 5, all nodes which are right to the left dotted line are potential nodes to have a new shortest distances or parent

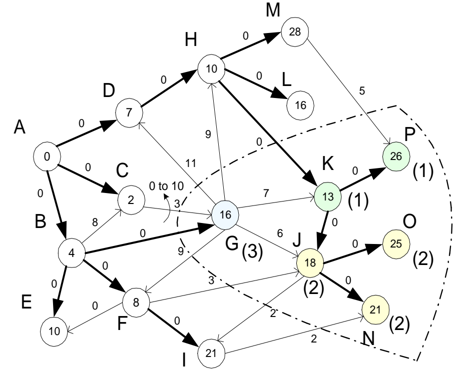

Fig. 4. The final result for G^* with edge (C, G) changes from 0 to 10

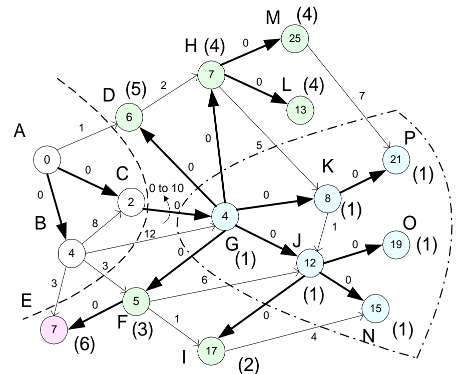

Fig. 5. The result for G^* with the weight of edge (C, G) changes from 0 to -5

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= Φ. 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 $B_1=\Phi$.

V. 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 upgrade 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 upgrade 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 γ 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.

VI. 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. When the weight of one edge in graph G becomes larger, our idea is to make the shortest distance of these nodes with less increment. While for the weight of one edge in graph G becomes smaller, we try to make the shortest distance of these nodes with more decrement. This property promises us that whenever we upgrade the distance for one node, its value is the final result. Thus we need not do any redundant comparisons, enqueue and dequeue. 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 γ denotes the *redundancy* factor with $\gamma > 1$.

Assume that each operation of addition, subtraction, comparison, enqueue and dequeue takes 1 time unit to complete. We compare the computation complexities among our algorithms and the one in [5]. Table I shows the consumed time units for the example presented in Section IV. The difference in computation time is mainly caused by the different updating sequences. Our algorithms eliminate all redundancy and thus gives a better result.

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

TABLE I

TIME COMPLEXITY COMPARISON OF ALGORITHMS

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.