An Efficient Algorithm for Dynamic Shortest Path Tree Update in Network Routing

Bin Xiao, Jiannong Cao, Zili Shao, and Edwin H.-M. Sha

Abstract: **Shortest path tree (SPT) construction is essential in high performance routing in an interior network using link state protocols. When some links have new state values, SPTs may be rebuilt, but the total rebuilding of the SPT in a static way for a large computer network is not only computationally expensive, unnecessary modifications can cause routing table instability. This paper presents a new update algorithm, dynamic shortest path tree (DSPT) that is computationally economical and that maintains the unmodified nodes mostly from an old SPT to a new SPT. The proposed algorithm reduces redundancy using a dynamic update approach where an edge becomes the *significant* edge when it is extracted from a built edge list Q. The average number of significant edges are identified through probability analysis based on an arbitrary tree structure. An update derived from significant edges is more efficient because the DSPT algorithm neglect most other redundant edges that do not participate in the construction of a new SPT. Our complexity analysis and experimental results show that DSPT is faster than other known methods. It can also be extended to solve the SPT updating problem in a graph with negative weight edges.**

Index Terms: **Dynamic update, network routing, open shortest path first (OSPF), shortest path tree (SPT).**

I. INTRODUCTION

As demands for broadband Internet applications have grown tremendously in recent years, it has become more important for computer networks to use high speed routing. In the widely deployed link-state-based routing protocols in Internet [1], such as open shortest path first (OSPF) [2], [3] and intermediate system-to-intermediate system (IS-IS) [4], [5], each router maintains a database describing the entire topology of an autonomous system. This database is built from the collected link-state advertisements of all other routers by flooding. From the topology database, each router constructs a shortest path tree (SPT) with itself as the root. A routing table can be constructed from the SPT and points out a lower-cost route (such as a route has a short delay) to each destination in a routing area (e.g., an OSPF area). Each router should update its routing table quickly to maintain QoS of networks whenever the network topology is modified [6], [7]. The updated routing table should provide the shortest packet delivery path, thereby reducing the transmission time of a packet from source to destination. In this way, the efficient rebuilding of an SPT in response to network topology change can improve packet delivery routing speeds.

Considerable work [8]–[10] has already been done on path determination in network routing with many routing protocols being studied and used in practical networks and with the shortest path derivation problem as a subject of continual research interest [11], like building cost-effective SPTs [12]. Efficient SPT construction is relevant to many research areas besides unicast routing protocols, in ad hoc mobile networks [13], transportation models, the robot motion planning [14], and VLSI design, to name just a few.

An SPT can be constructed using well-known static algorithms, such as the Dijkstra algorithm [15] and the Bellman-Ford algorithm [16] for a graph with negative weight edges (without negative loops). However, the Dijkstra algorithm is very inefficient when the occurrence of link state changes in a network requires the update of only a small part of an old SPT. Methods that make use of static algorithms require every router to recompute the whole SPT whenever there is a single link state change and this in turn requires that the router's routing table be completely scanned and updated. In most cases the new router SPT is a little or not at all different from its precursor. Using static algorithms to update an old SPT incurs a lot of unnecessary computation and routing table entry replacements. A complete reconstruction of a new SPT may further cause some undesirable traffic load fluctuation inside a particular route [17]. Thus, it is crucial that algorithms to dynamically update SPT be introduced to more efficiently handle link state changes in a network.

Dynamic updating procedures have been applied in a number of responses to the inefficiency of static SPT update methods. Dynamic updating has been analyzed in terms of all-pairs nodes [18], or in the environment of replacing a failing edge on an existing SPT [19]. [20] sought to reduce computation times by separating the *weight-increase* and *weight-decrease* operations. The weight-decrease operation is more complicated and has been fully addressed in [21]. [20] and [22] succeeded in reducing computation times compared to static methods, however, their approaches involved excessive computation. [23] is an improved version appeared of the work of [22], presenting an algorithm based on a ball and string model where an edge weight change corresponds to the length change of a string, however, it is not clear how this algorithm is applied when edge weights change in a graph with negative weight edges. An additional drawback is that it produces some redundant computation because its update process enqueues unnecessary edges in a dynamically refreshing queue.

This paper proposes a new dynamic shortest path tree (DSPT) algorithm for updating an old SPT to a new one when the link states of a network are assigned new weights. The efficiency of

Manuscript received April 21, 2005; approved for publication by Hussein Mouftah, Division III Editor, August 28, 2007.

This work has been supported by HK RGC Polyu 5196/04E and China National 973 Grant 2007CB307100.

B. Xiao, J. Cao, and Z. Shao are with the Hong Kong Polytechnic University, Hong Kong, email: {csbxiao, csjcao, cszlshao}@comp.polyu.edu.hk.

E. H.-M. Sha is with the University of Texas at Dallas, USA, email: edsha@utdallas.edu.

1229-2370/07/$10.00

the DSPT algorithm is demonstrated by an analysis of the probability of the relevant edges that contribute to the construction of a new SPT. This probability is shown in an arbitrary tree model since the nodes to be updated are linked in a tree structure when the weight of one edge increases. DSPT displays a number of advantages. It engages with far fewer edges than any other algorithm in the literature and not only reduces the computational complexity required to update an old SPT, it also maintains the routing table stability. DSPT is less complex than other dynamic SPT update algorithms. Further, although in this paper we describe it in the context of a network topology change where the weight of only a single edge changes, it may just as easily be applied in a context of multiple link weight changes. Finally, the proposed algorithm can also be extended to allow the rebuilding of new SPTs for graphs with negative weight edges. The experimental results show that, compared with the algorithm in [23], the number of edges put into a queue is reduced by up to 31.5%, and the time required to search for the minimum value edge from the same queue is reduced by up to 27.6% by applying the new algorithm.

In Section II, we will provide an example that both illustrates the redundant computation inherent in previous work and demonstrates for the theoretical bases of the proposed algorithm. In Section III, the probability of an edge contributed in the new SPT, which is modeled as the number of vertices kept in a tree, is explored to provide theoretical principles. In Section IV, some definitions and notations used in this paper are given, then the new DSPT algorithm is described in detail. Section V analyzes the theoretical bounds of the DSPT algorithm on asymptotic computational complexity. In Section VI, we present the experimental results. Section VII offers the concluding remarks. In the Appendix, we prove the correctness of the DSPT algorithm.

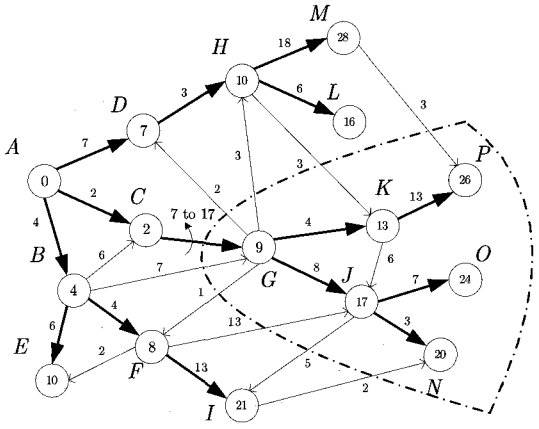


Fig. 1. The weight of edge (C, G) increased from 7 to 17.

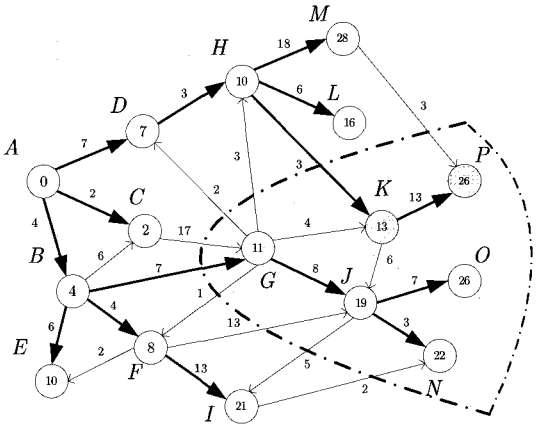


Fig. 2. The new SPT when the weight of edge (C, G) increased from 7 to 17.

II. DYNAMICALLY GENERATING A NEW SPT

Before formally describing the dynamic SPT algorithm, we provide an example, to illustrate how a dynamic update process generates a new SPT. Noting that in this paper edges in a graph are depicted as unidirectional, Fig. 1 depicts a situation in which the weight of one edge becomes larger. The whole graph represents a network topology and the nodes are labeled A to P, representing routers. The weight of a link between two nodes denotes the link state cost between different routers (for example, network traffic delay). If an edge e is $u \rightarrow v$, node u is the source node of edge e and node v is the end node. The existing SPT for the given graph with a root on A is composed of all directed bold edges. The shortest distance of a node is the summation of the weight of all edges on the shortest path, which starts from node A and goes through bold edges till arrives at the node itself. The number inside each node indicates the shortest distance from the source node A.

When the weight of the edge (C, G) increases from 7 to 17, the new SPT must be rebuilt. The shortest paths of nodes outside the area enclosed within the dotted lines can remain unchanged in the new SPT because the shortest distance of a node can not become smaller when the weight of an edge increases. In other words, the old shortest paths of nodes (i.e., A, B, C, D, E, F, H, I, and M) are legitimate in an updated new SPT and, as a result, a dynamic update process saves computation time. It is necessary, however, to modify the shortest paths from the source node A to nodes enclosed within the dotted line. These nodes are all descendants of node G (including G itself) in the old SPT.

Fig. 2 shows the new SPT. We will figure out how nodes G, J, K, N, O, and P are updated. If the shortest paths to them follow the same routes as in the old SPT, the shortest distances will increase by 10. We can choose other alternative paths through nodes beyond the bounded area to make their shortest distances increased smaller than 10. There may exist several incoming edges to nodes G, J, K, N, and P. Every incoming edge defines a new shortest distance to its end node. The incoming edge that incorporates the smallest increment should be investigated. For example, edge (C, G) and (B, G) can both reach node G. It is obvious that if the shortest path to node G goes through the former edge, the shortest distance to node G will increase by 10, while through the latter edge by 2. That implies edge (B, G) should be investigated during the updating process whereas edge (C, G) should not. Table 1 lists all edges that should be investigated in the second row. In the new SPT, we should update the end nodes of those edges. There is no incoming edge to node O from outside and we do not include node O in the table.

Table 1 contains information on the nodes to be updated. The

Table 1. Edges connected to nodes to be updated that may contribute to the change of the new SPT.

Nodes to be updated	G	J	K	N	P
Incoming edge	(B,G)	(F,J)	(H,K)	(I,N)	(M,P)
Increased distance	2	4	0	3	5
Contribution	Yes	No	Yes	No	No

first row in Table 1 lists all these nodes. The second row shows an incoming edge to a node. In the third row, we show the increased distance from the source node A to the node in the first row if the new shortest path goes through the edge in the second row. The fourth row indicates whether the edge in the second row contributes to the construction of the new SPT. The edge in the second row makes the smallest increment to its end node among all incoming edges. If an edge in the second row is included in the new SPT, that edge is denoted by Yes. Otherwise, a No is denoted. Among all nodes (G, J, K, N, O, and P) to be updated, node K can have the new smallest increment via its incoming edge (H, K) by 0. All descendants of node K (K and P) in the old SPT can reach new shortest distances via the shortest path from node A to node H and the edge (H, K) without any extra increments. We can update node G, J, N, O by choosing new paths through edge (B, G) and their new shortest distances increase by 2. The *significant Edge* is defined in this paper as the edge only in the new SPT but not in the old SPT. In our dynamic algorithm, an edge becomes a significant edge when it is extracted from an edge set Q. In this example, we observe that only edge (H, K) and (B, G) are significant edges. These significant edges have their contributions denoted as *Yes* in the fourth row in Table 1.

A dynamic update method is proposed in [23] to get a new SPT. When the weight of edge (C, G) increases from 7 to 17 as in Fig. 1, the method in [23] requires all edges in the second row in Table 1 to be added into a queue Q. Q is an edge list that contains edges and their related information. Using the edge list Q, the update process will extract an edge that has the minimum increased value (using the same definition as the value for edges shown in the third row in Table 1) from Q and perform modifications, such as updating SPT and extracting other edges. The process terminates when the edge set Q is empty. The nodes to be updated and their related edges involved in [23] are modeled as in Fig. 3(a). Every incoming edge has a weight to show the increment to the shortest distance of its end node if the shortest path goes through it. The data inside the node is the smallest increased value among all its incoming edges. The method in [23] will initially insert all 5 incoming edges in Fig. 3(a) into Q and at the end of the updating process these 5 edges will be removed from Q.

To reduce redundancy, we can only input edges to Q that are significant edges. For other edges that would not be used to construct the new SPT (such as edges (M, P),(F, J), and (I, N)), we save computations by avoiding the action of inputting them into the edge set Q first and removing them later. If the incoming edge is kept only if its increased value is smaller than its ancestor nodes, the new transformed graph from Fig. 3(a) is in Fig. 3(b). The data for each node is the smallest value either from the increments of its incoming edges, or from its own parent node. The transformation can be easily obtained following a depth-first-search (DFS) sequence, which is realized by a comparison of its parent's value and the smallest increased value among its all incoming edges. If the edge list Q only consists of incoming edges as in Fig. 3(b) (only 2 compared with 5 in Fig. 3(a)), the computation time for adding edges to Q, searching for the edge with the minimum increased value, and extracting related edges from Q will be reduced.

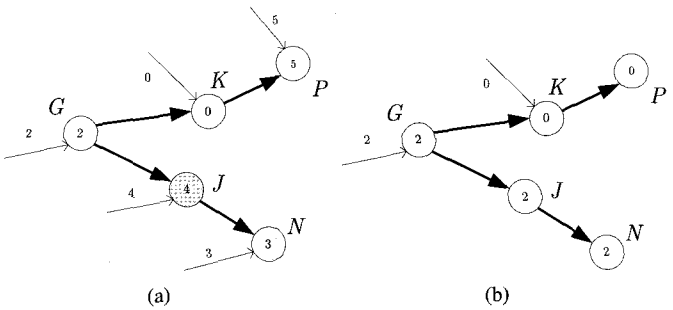


Fig. 3. (a) Nodes with one incoming edge to denote the possible smallest increment, and (b) two significant edges for five nodes to be updated.

III. PROBABILITY ANALYSIS

In this section, we build models to demonstrate the probability of an incoming edge to one node becoming a significant edge. When there is an edge weight increment, all nodes to be updated form a subtree that is a part of the old SPT (e.g., nodes G, J, K, N, O, and P construct a tree with the root at node G following the old SPT in Fig. 1). The tree model has been analyzed first as the simplest linear graph, then with a complete binary tree, and finally using an arbitrary tree. From the probability results, we conclude the average number of nodes to be kept in a common situation. The average number of nodes reflects the average number of edges to be extracted from Q during a dynamic update process in the DSPT algorithm. Because it is impossible to determine the exact bound of nodes to be updated before the exploration of dynamic update when there is an edge weight decrement, the proposed DSPT algorithm in Section IV does not incorporate the tree model analysis to handle edge weight decrements in a network topology change. The probability in the tree model can be successfully applied in the computation time analysis of dynamic SPT update to handle edge weight increments.

A. Linear Graph

Consider first a simple model as a linear graph structure, as depicted in Fig. 4. Suppose that each node a_i has a random value x_i uniformly distributed between 1 and m. This value represents the smallest increment among all incoming edges to node a_i. It follows the same concept as the data inside each node in

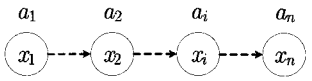


Fig. 4. A linear graph model and the value of each node is randomly distributed between 1 and m.

Fig. 3(a). Given n nodes $(a_1, a_2, \cdots, a_n)$ in a linear graph, let node a_1 be the source node (the root). Node a_i is kept when its value is smaller than the values of all its ancestors. If a node is kept, its corresponding incoming edge contributes to the rebuilding of a new SPT and should be input into the edge list Q where the edge is a significant edge. The average number of nodes that are kept reflects the average queue length (how many edges) of a constructed Q.

Lemma 1: Let p_i be the probability that node i is kept, then $p_i < 1/i$ for $i > 1$.

Proof: Let v_i be a random variable for node a_i. $v_i = 1$ when node a_i is kept, and $v_i = 0$ otherwise. The expected value of v_i is $E(v_i)$. Let x_i denote the random value for node a_i. Then, $p_i = E(v_i) = Prob(x_i < min(x_1, x_2, \cdots, x_{i-1}))$.

We know that $\forall i$, x_i is a uniformly distributed integer from 1 to m. When $x_i = j$, node i is kept if $\forall k < i$, x_k is a value between $j+1$ and m. This implies there are $(m-j)^{i-1}$ cases for $x_i = j$ and $v_i = 1$. Thus, the number of all cases for $v_i = 1$ (node i is kept) is $\sum_{j=1}^{m}(m-j)^{i-1}$ since the value of x_i can be from 1 to m. The number of all possible cases is m^i because the first i nodes (from a_1 to a_i) can take any random values from 1 to m.

Because $\sum_{j=1}^{m}(m-j)^{i-1} < \int_1^m x^{i-1}dx < \frac{m^i}{i}$, we have $E(v_i) = \frac{\sum_{j=1}^{m}(m-j)^{i-1}}{m^i} < \frac{m^i}{m^i i} = 1/i$. □

From Lemma 1, we know that the probability of a node to be kept is related to its position in the sequence of a linear graph. We always keep the first node and thus $p_1 = 1$.

Theorem 1: Given a linear graph with n nodes, the average number of nodes to be kept is $\ln n + 1$.

Proof: Let V be a random variable representing the number of nodes to be kept in a linear graph. Let v_i be a random variable for node a_i. $v_i = 1$ when node a_i is kept, and otherwise $v_i = 0$.

It follows that $V = \sum_{i=1}^{n} v_i$. The expected value of V is $E(V) = E(\sum_{i=1}^{n} v_i) \leq \sum_{i=1}^{n} E(v_i)$. By Lemma 1, we know $\forall i > 1$, $E(v_i) < 1/i$ and $E(v_1) = 1$. Thus, $\sum_{i=1}^{n} E(v_i) < 1+1/2+\cdots+1/n$. In [24], it is proved that $\sum_{i=1}^{n} 1/i \leq \ln n+1$. Therefore, $E(V) < \ln n + 1$. □

From Theorem 1, we know that if nodes to be updated are in a linear-graph structure, the average number of nodes to be kept is no more than $\ln n + 1$. If a node is kept, its incoming edge to the node should be input to the edge list Q. In other words, only a small number of edges should be in Q when nodes to be updated satisfy the linear-graph model. Therefore, we can remove many redundant edges in Q (thus Q has a short queue length) and greatly decrease the computation time required to add and delete edges and find the minimum value from the edge list Q.

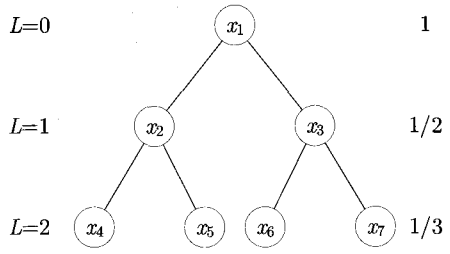


Fig. 5. A complete binary tree model.

B. Complete Binary Tree

A tree T is denoted as an *n-tree* if every vertex has at most n offspring. The binary tree is a 2-tree. If all vertices of T, other than the leaves, have exactly n offspring, we say that T is a complete n-tree. A complete binary tree model is shown in Fig. 5 to analyze the probability of nodes to be kept. Each node has the same property as in Fig. 4. Let the level of the root node be 0. Each child of the root is a level 1 node. The next generation is level 2 node and so forth. The depth of the tree in Fig. 5 is 2.

Given a complete binary tree with depth k, from Lemma 1 we know that the probability of keeping a node in level i is smaller than $1/(i+1)$. The number of all nodes in the complete binary tree is $N = 2^{k+1} - 1$.

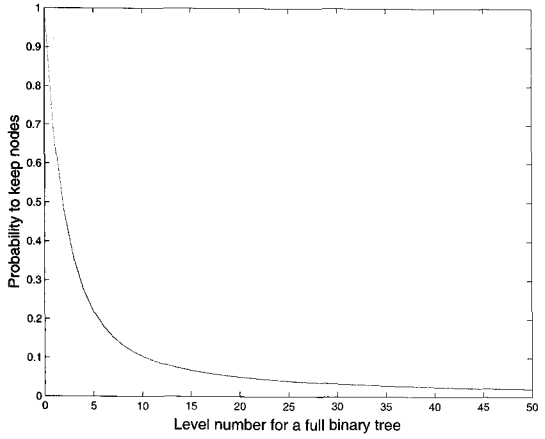
Theorem 2: Given a complete binary tree with depth k and the number of all nodes N is $2^{k+1} - 1$, the average number of nodes to be kept is $\frac{N}{k-1}$ for $k \geq 2$.

Proof: We prove the theorem by induction. Let p_i be the probability to keep nodes in the ith level. Thus, $p_0 = 1$ and $\forall i > 0$, $p_i < 1/(i+1)$. The number of nodes in the ith level is $N_i = 2^i$. Let X be the random variable representing the number of nodes to be kept.

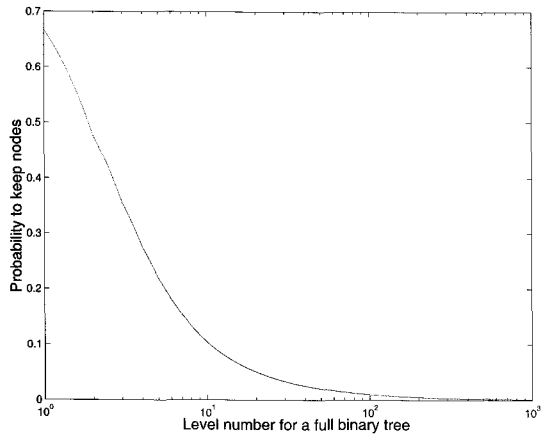
1. **Induction base:** For $k = 2$ as in Fig. 5, $N = 7$. The expected value of X is $E(X) = \sum_{i=0}^{2} p_i N_i < 1 + 2 \times \frac{1}{2} + 4 \times \frac{1}{3} = 3\frac{1}{3}$, which is smaller than $\frac{7}{k-1} = 7$.
   For $k = 3$, $N = 15$. The expected value of X is $E(X) = \sum_{i=0}^{3} p_i N_i < 1 + 2 \times \frac{1}{2} + 4 \times \frac{1}{3} + 8 \times \frac{1}{4} = 5\frac{1}{3}$, which is smaller than $\frac{15}{k-1} = 7.5$.
   For $k = 4$, $N = 31$. The expected value of X is $E(X) = \sum_{i=0}^{4} p_i N_i < 1 + 2 \times \frac{1}{2} + 4 \times \frac{1}{3} + 8 \times \frac{1}{4} + 16 \times \frac{1}{5} = 8\frac{8}{15}$, which is smaller than $\frac{31}{k-1} = 10\frac{1}{3}$.
2. **Induction hypothesis:** Let $k = m$ and m is an arbitrary natural number, $m \geq 4$ and $N = 2^{m+1} - 1$. Assume that the average number of nodes to be kept is $\frac{N}{k-1} = \frac{2^{m+1}-1}{m-1}$.
3. **Induction step:** When $k = m+1$, $p_{m+1} < \frac{1}{m+2}$, $N_{m+1} = 2^{m+1}$, and $N = 2^{m+2} - 1$. The expected value

$$
\begin{aligned}
E(X) &\leq (\sum_{i=0}^{m} N_i)\frac{1}{m-1} + N_{m+1}p_{m+1} \\
&\leq \frac{1}{m-1}(2^{m+1} - 1) + 2^{m+1}\frac{1}{m+2} \\
&= (\frac{1}{m-1} + \frac{1}{m+2})2^{m+1} - \frac{1}{m-1} \\
&< \frac{1}{m}(2^{m+2} - 1) = \frac{N}{m} = \frac{N}{k-1} \quad (\text{for } m \geq 4)
\end{aligned}
$$

□



(a)



(b)

Fig. 6. (a) The probability of keeping nodes for a complete binary tree, and (b) the probability of keeping nodes with a logarithm axis.

Theorem 2 indicates that the probability of keeping a node is smaller than $1/(k-1)$ given a complete binary tree with a depth of k. This implies that when the value of each incoming edge is randomly distributed to those N nodes among all incoming edges, only a small part ($N/(k-1)$ of N incoming edges) is useful for rebuilding the new SPT. In Fig. 6(b), a binary tree depth is measured by the logarithm axis. Fig. 6 shows that when the tree depth is k, the probability for a node to be kept is close to $1/(k-1)$. When the tree depth becomes larger, the probability is very small.

C. Arbitrary Tree

Theorem 3: Given a complete n-tree with depth k, the average number of nodes to be kept is $\sum_{i=0}^{k} \frac{n^i}{i+1}$ for $k \geq 1$ and $n \geq 2$.

Proof: Let p_i be the probability to keep nodes in the ith level. Thus, $p_0 = 1$ and $\forall i > 0$, $p_i < 1/(i+1)$. The number of nodes in the ith level is $N_i = n^i$ and the number of all nodes in the tree is $N = \sum_{i=0}^{k} n^i$. Let X be the random variable representing the number of nodes to be kept. Thus, we have

$$E(X) = \sum_{i=0}^{k} p_i N_i < \sum_{i=0}^{k} N_i \frac{1}{i+1} = \sum_{i=0}^{k} \frac{n^i}{i+1}.$$

□

Given an arbitrary n-tree T_1 with N nodes, we can construct a balanced n-tree T_2 by moving a vertex upward such that the height of T_2 is k and all vertices from level 0 to level $k-2$ have n offspring. The number of nodes to be kept in T_2 compared with in T_1 is non-decreasing because, according to Lemma 1, a vertex moving upward to the root of T_2 does not decrease the probability. T_2 is a balanced tree with N vertices and depth k, which means

$$\frac{n^k - 1}{n-1} < N \leq \frac{n^{k+1} - 1}{n-1}.$$

Hence, we have

$$k = \lceil \log_n [N(n-1)+1] \rceil - 1.$$

T_2 is a tree that contains a sub-complete n-tree with depth $k-1$. The number of vertices in level k is $N - \frac{n^k - 1}{n-1}$. From Theorem 3, the average number of nodes to be kept is $\sum_{i=0}^{k-1} \frac{n^i}{i+1} + \frac{N - \frac{n^k-1}{n-1}}{k+1}$. Thus, we have the following Corollary.

Corollary 1: Given an arbitrary n-tree with N nodes ($n \geq 2$), the average number of nodes to be kept is $\sum_{i=0}^{k-1} \frac{n^i}{i+1} + \frac{N - \frac{n^k-1}{n-1}}{k+1}$ and $k = \lceil \log_n [N(n-1)+1] \rceil - 1$.

IV. THE DSPT ALGORITHM

In this section, we will first give some definitions and notations, and then present the DSPT algorithm to be applied to dynamically update an old SPT so that it becomes a new SPT. We also explain how the DSPT algorithm works in detail to handle a link state change. When the network topology change is composed of multiple link state modifications, it is possible to extend the DSPT algorithm to efficiently solve the dynamic SPT update problem.

A. Definitions and Notations

The following provides the definitions and notations to be used in the remainder of this paper. Let $G = (V, E, w)$ denote a directed graph where V is the set of nodes, E is the set of edges and w represents the weight of each edge in E in the graph. G contains no negative-weight edges. Let $S(G) \in V$ denote the source node in the graph G. We use $w(e)$ to denote the weight for each directed edge $e \in E$ and use $w'(e)$ to denote the new weight. Every node in a shortest path tree (SPT) can be reached from $S(G)$ through a unique directed path that has edges only on the SPT. The length or distance of a directed path is the sum of the weights of all edges on the path.

We define the following notations for a node i. Given an edge $e : i \rightarrow j$, i is the source node and j is the end node of e. Let $S(e)$ be the source node of edge e ($S(e) = i$) and $E(e)$ be the end node ($E(e) = j$). Node i is the parent of node j if i is the source node and j is the end node of an edge e that belongs to the SPT. Let $P(i)$ be its parent node and $D(i)$ be its shortest distance. The value of $D(i)$ is calculated by the length of the unique path from the source node $S(G)$ to node i in an SPT. For an edge e with a new weight $w'(e)$, let $d = D(i) + w'(e) - D(j)$, where d represents the increased value to node j if the shortest

path to node j via the shortest path from $S(G)$ to node i and the edge e. The value of d can be negative if the weight of edge decreases. The descendants of a node i are all nodes that are reachable from i only through edges in a given SPT. We use $des(i)$ to denote the node set that comprises descendants of node i and itself.

A tree is consecutively changed during below DSPT dynamic algorithm to handle an edge weight change. The initial shortest path tree (SPT) can be gotten by the static method of Dijkstra algorithm [15] (for graphs without negative weight edges) or the Bellman-Ford algorithm [16] (for graphs without negative-weight cycles). Whenever a network topology change happens, a dynamic update process can update an old SPT to a new SPT. In the dynamic update process of the DSPT algorithm, we change the structure of the old SPT by re-configuring the parent-child relationships among nodes. The final new SPT holds once the algorithm terminates. A tree in the updating stage is called the *temporary* SPT in the paper.

Given a node set N, we define the following edge sets *Source_part{N}* and *End_part{N}*.

Definition 1: Given a graph $G = (V, E, w)$ and a node set N ($N \subseteq V$), let Source_part$\{N\}$ be an edge set ($\subseteq E$) and Source_part$\{N\}$ = $\{\ e|\ S(e) \in N,\ E(e) \notin N\ \}$; let End_part$\{N\}$ be an edge set ($\subseteq E$) and End_part$\{N\}$ = $\{\ e|\ E(e) \in N,\ S(e) \notin N\ \}$.

From Definition 1 and Fig. 1, suppose that $N = \{G, J, K, N, O, P\}$. Then, *Source_part*$\{N\}$ and *End_part*$\{N\}$ are as following:

Source_part$\{N\} = \{(G, D), (G, F), (G, H), (J, I)\}$ and

End_part$\{N\} = \{(C, G), (E, G), (F, J), (H, K), (I, N), (M, P)\}$.

B. DSPT Algorithm Specification

The DSPT algorithm maintains an edge list Q and a node list M to dynamically update an old SPT. Each element in Q is configured by the form $\{e, min\_inc\}$ where e denotes the edge and min_inc denotes the increased value to the shortest distance of node $E(e)$ given that the new shortest path goes through e. When the weight of an edge increases, M is used to denote a node set. One advantage of using M is to facilitate an easy computation of the new SPT. Every node to be updated is added to M. Given a node $v \in M$, there is an increased value $M.v.inc$ attached. $M.v.inc$ is the smallest incremental value of its incoming edges. Another advantage of using the node list M is that it ensures there are not two units in Q with the same min_inc.

We conduct the dynamic update process of the DSPT algorithm based on the edge list Q. Whenever an edge e needs to be put into edge list Q, an instruction $Enqueue(Q, \{e, min\_inc\})$ is executed. If the end node of edge e is already in Q, the new element $\{e, min\_inc\}$ will replace the old one when the new min_inc is a smaller value. The replacement proceeds with the min_inc value comparison. The instruction $Extract(Q)$ selects the element $\{e, min\_inc\}$ that has the smallest min_inc and removes it from Q. The extracted edge e becomes the significant edge in the DSPT algorithm. In the case of an edge weight decrement when two or more units contain the same smallest min_inc in Q, the one that has a smaller $D(E(e))$ will be extracted first. If there is a tie, we arbitrarily choose one. Whenever the node $S(e_1)$ becomes the parent of $E(e_1)$ in the continuously changed temporary SPT, which is denoted by $P(E(e_1)) = S(e_1)$, the node $E(e_1)$ automatically turns out to be a child of node $S(e_1)$.

Algorithm 1 shows the DSPT algorithm. To get the new

Algorithm 1 DSPT algorithm

Step 1: From $G = (V, E) \rightarrow SPT$ /* using a static Dijkstra algorithm */
Step 2: wait until one edge e : $i \rightarrow j$
1: **if** $w'(e) > w(e)$ and $e \in SPT$ **then**
2: $d = w'(e) - w(e)$ /* case 1: one edge weight increased */
3: go to *Step* 3
4: **else if** $w'(e) < w(e)$ and $D(i) + w'(e) < D(j)$ **then**
5: $d = D(i) + w'(e) - D(j)$ /* case 2: one edge weight decreased */
6: go to *Step* 4
7: **else**
8: go to *Step* 2
9: **end if**

Step 3 : /* for one edge weight increased */
1: Initialization
2: $M \leftarrow j$, $M.j.inc = d$, $Enqueue(Q, \{e, d\})$, /* $\forall v \in des(j)$ are updated */
3: **if** $v \in des(j)$, following the sequence of DFS from node j in SPT **then**
4: $min\_inc = MIN\{D(S(e)) + w(e) - D(E(e))\}$ where $E(e) = v$ & $S(e) \notin des(j)$
5: **if** $min\_inc < M.v.inc$ **then**
6: $Enqueue(Q, \{e, min\_inc\})$, $M.v.inc = min\_inc$
7: **end if**
8: **if** k is the direct child of v in SPT **then**
9: $M \leftarrow k$, $M.k.inc = M.v.inc$
10: **end if**
11: **end if**
12: **for** $Q \neq \emptyset$ **do**
13: $\{e_1, min\_inc\} \leftarrow Extract(Q)$, $P(E(e_1)) = S(e_1)$, /* temporary SPT is altered */
14: $\forall v \in des(E(e_1)), D(v) = D(v) + min\_inc$, /* all descendants of $E(e_1)$ updated */
15: Remove edges from Q, which have end nodes belonging to $des(E(e_1))$
16: Remove v from M
17: **if** $e \in Source\_part\{des(E(e_1))\}$ & $e \in End\_part\{M\}$ **then**
18: $min\_inc = D(S(e)) + w(e) - D(E(e))$
19: **if** $min\_inc < M.E(e).inc$ **then**
20: $M.E(e).inc = min\_inc$ /* update the old information in Q */
21: $Enqueue(Q, \{e, min\_inc\})$
22: **end if**
23: **end if**
24: **end for**
25: Go to *Step* 2

Step 4 : /* for one edge weight decreased */
1: Initialization /* descendent nodes of j are updated once */
2: $\forall v \in des(j)$, $D(v) = D(v) + d$, $P(j) = i$ /* $d < 0$ */
3: **if** $S(e) \in des(j)$ **then**
4: $min\_inc = D(S(e)) + w(e) - D(E(e))$
5: **if** $min\_inc < 0$ **then**
6: $Enqueue(Q, \{e, min\_inc\})$
7: **end if**
8: **end if**
9: **for** $Q \neq \emptyset$ **do**
10: $\{e_1, min\_inc\} \leftarrow Extract(Q)$, $P(E(e_1)) = S(e_1)$
11: $\forall v \in des(E(e_1)), D(v) = D(v) + min\_inc$
12: Remove edges from Q, which have end nodes belonging to $des(E(e_1))$
13: **if** $e \in Source\_part\{des(E(e_1))\}$ **then**
14: $min\_inc = D(S(e)) + w(e) - D(E(e))$
15: **if** $min\_inc < 0$ **then**
16: $Enqueue(Q, \{e, min\_inc\})$
17: **end if**
18: **end if**
19: **end for**
20: Go to *Step* 2

SPT, we exploit two operations depending on whether the edge weight has increased or decreased. One operation deals with an edge weight increment and the other for an edge weight decrement. The reason for the separated update operations is for a faster building of a new SPT. When an edge $e : i \rightarrow j$ has a larger weight, only nodes $des(j)$ in the old SPT need to update their shortest distances (or parent nodes). In this situation of an edge weight increment, the node list M is used. The purpose of this is to reduce the computation time required to add and remove elements and to search for the minimum min_inc from Q. However, when an edge $e : i \rightarrow j$ has a smaller weight, we do not know the exact range of nodes to be refreshed before the realization of the new SPT. In this situation of an edge weight decrement, all nodes in $des(j)$ can be updated once with smaller shortest distances. The decreased shortest distances of nodes in $des(j)$ can make their connecting neighbors to attain smaller shortest distances too. Thus, the effect of distance decrements can be propagated further to impact more connected nodes. Whenever the update process ends for a network topology change, the DSPT algorithm waits at Step 2 for another topology variance.

C. Explanation of the DSPT Algorithm

To help understand the DSPT algorithm, a simple example is shown in Fig. 7 in Section V-B and the following provides a brief description of how this algorithm works. In the DSPT algorithm, Step 1 adopts a static *Dijkstra* algorithm to obtain the initial shortest path tree from a known graph G with $S(G)$ as its source node. At Step 2, whenever there is a weight change for edge $e : i \rightarrow j$, DSPT determines whether it is necessary to update the old SPT. Only under case 1 and case 2 (as shown in Algorithm 1), it is necessary to go to Step 3 and Step 4, respectively, to create a new SPT. d is the increased (or decreased) value of edge e.

Step 3 deals with the situation that the weight of an edge in an old SPT becomes larger. Since we know exactly the set of nodes to be updated, the DSPT algorithm introduces a node set M to contain them. In M, each node has a value that represents an increment to its old shortest distance. This value is the smallest value either from its parent node, or from its incoming edges. During the initialization phase, M consists of all potential nodes to be updated. Node j is first input to M along with the increased value d. Then, all nodes in $des(j)$ are examined by the sequence of DFS from the root node j in the old SPT. The first node assigned to v is node j and has its initial value $M.j.inc$. The incoming edges of each node $v \in des(j)$, excluding those having source nodes in $des(j)$, are measured with the increased value of $D(S(e)) + w(e) - D(E(e))$. The edge e with the smallest increased value (min_inc) is selected. This minimum value min_inc is compared with $M.v.inc$. If min_inc is smaller, edge e is added to the edge list Q and $M.v.inc$ is updated. Subsequently, the direct child k of node v in the temporary SPT has the same inc value as v in M. This initialization process ensures that every edge in the edge list Q has a smaller increased value than any of its ancestors. At the end of the initialization, Q consists of useful edges (including all significant edges) for the second phase.

The second phase of Step 3 ends when Q is empty. If Q is empty, the dynamic update process terminates since the new SPT has already been built and the DSPT algorithm will move to Step 2 to wait for another edge weight change. Otherwise, we will extract the edge (e_1) with the minimum value (min_inc) from Q. Once e_1 is chosen, its source node $S(e_1)$ becomes the parent of the end node $E(e_1)$ in the maintained SPT. All descendants $des(E(e_1))$ can attain their new shortest distances by adding min_inc to their old ones. Moreover, any edge in Q with end node belonging to $des(E(e_1))$ should be removed. The updated nodes will be removed from M too. In addition, edges that have source nodes in $des(E(e_1))$ and end nodes in M are checked because they may allow some nodes in M to achieve even smaller shortest distance increments. These edges need be added to the edge list Q to continue the dynamic update process.

In Step 4, the DSPT algorithm yields a new SPT when the weight of an edge $e : i \rightarrow j$ decreases. Because we cannot know which node needs to be updated before the conducting of the update process, the node list M is no longer used. During the initialization, all nodes in $des(j)$ are updated once as their old distances increased by d (d is a negative value). Node i becomes the new parent of node j. We notice that all edges with source nodes in $des(j)$ are potential significant edges that could lessen the shortest distances of nodes outside the subtree $des(j)$. Edge e with the minimum increased (i.e., maximum decreased) value min_inc is selected to its end node $E(e)$. If min_inc is below 0, edge e and its accompanied value min_inc are added to the edge list Q. The edge list Q is created to contain all significant edges. The update process, which is similar to Step 3, terminates when Q is empty.

D. Multiple Link Weight Changes

The DSPT algorithm in Section IV-B does not address the problem of multiple link weight changes to derive a new SPT. It is very inefficient to update SPT if we take care of one single weight change after another. However, with the edge classification according to its weight change, the DSPT algorithm can be efficiently applied to solve this problem. It is required to classify edges into two groups at Step 2 in Algorithm 1. Group 1 consists of weight increased edges that meet case 1 and group 2 consists of weight decreased edges that meet case 2. A temporary SPT from an outdated SPT will be built after the dynamic update operations to handle edges in group 2. Then, the update process will continue to attain the final SPT for edges in group 1 based on the temporary SPT.

The dynamic update process deals with edges in group 2 first. All edges in group 2 paired with their decreased value d are included in the edge list Q, which has the same definition as the Q in Step 4. Based on the old SPT and Q, a temporary SPT can be built through the update iteration procedure as in the second phase in Step 4 in the DSPT algorithm.

When we finish the updates from an old STP to a temporary SPT for edges in group 2, we move to deal with edges in group 1. Edges in group 1 will be re-verified to meet the requirement of case 1 on the temporary SPT. Some can be removed because their end nodes already realize their smaller shortest distances via alternative paths. Among all remaining edges in group 1,

an edge $e_1 : i_1 \rightarrow j_1$, which makes its end node j_1 to obtain a new smallest shortest distance $D(j_1)$, will be extracted. We arbitrarily choose one from two or more edges if via them their end nodes can have the same smallest shortest distance. We create a node list M_1 that equals $des(j_1)$ in the temporary SPT. Given a node $v \in M_1$, let $M_1.v.inc = w'(e_1) - w(e_1)$. Then, we continue to extract another edge $e_2 : i_2 \rightarrow j_2$ from group 1 that has the smallest distance value $D(j_2)$. We create a node list M_2 following the same rule to create M_1. Let $M = M_1 \cup M_2$ where $\forall v \in M$, $M.v.inc = M_1.v.inc + M_2.v.inc$. If v is not in M_1, then $M_1.v.inc = 0$. If v is not in M_2, then $M_2.v.inc = 0$. When group 1 is empty after the extraction of all edges, the node list M can be built as $M = M_1 \cup M_2 \cup \cdots$. M comprise all nodes to be updated and those nodes are in some subtrees. We conduct the instructions in the *for* loop at the initialization part in Step 3 as in Algorithm 1 for each subtree and the edge list Q can be obtained accordingly. From M, Q and the temporary SPT after the update of edges in group 2, the dynamic update procedure for handling multiple edge weight increments is the same as the second part in Step 3 in the DSPT algorithm. When the update procedure finishes, the final new SPT is achieved.

V. ALGORITHMIC COMPLEXITY AND APPLICATION

After the algorithm complexity illustration of the DSPT algorithm we extend this algorithm to be applied to obtain a new SPT in a graph with negative weight edges. The DSPT algorithm has less computation time compared with previous work. We also show the application of DSPT in a complicated situation that a graph contains negative weight edges without negative cycles.

A. Algorithm Complexity

This subsection illustrates the theoretical bounds of the complexity of the DSPT algorithm. As discussed previously with a single edge weight change, in most cases it is necessary to update only a small part of the old SPT. Hence, the new dynamic algorithm will be less complex than a static algorithm in most cases. Note that in the worst case, a dynamic algorithm may not make any improvements compared to a static algorithm, like using Dijkstra algorithm to get a new SPT. The complexity of the proposed algorithm is measured using the following metrics:

- The total number of comparisons for all edges that are connecting to a node to be updated, from which the edge with the minimum increased value is put into Q.
- The total number of comparisons required to extract edges with the minimum increased value from Q.
- The total number of computations to update nodes and remove edges from Q.

To accurately derive the algorithm complexity for the dynamic update of old SPTs, let t be the total number of nodes that are updated. We can assume that an updated node has either a new shortest distance, or a new parent node in the new SPT, or both. Let t_p denote the total number of *new* edges in the newly constructed SPT. A *new* edge is an edge that either does not belong to the old SPT or that has a new weight. It is obvious that $t_p \leq t$ and t_p represent the number of significant edges in the DSPT algorithm. The value of t and t_p are related only to the network topology, the old SPT and the edge with a new weight. Let t_r be the number of edges that have been put into Q during the dynamic update process in DSPT. Usually, t_r is close to t_p and $t_p \leq t_r \leq t$. Given a graph G that represents the whole network topology, let N_{max} be the maximum node degree. If E_t is the number of incident edges connecting to t updated nodes, E_t should be $O(N_{max}t)$.

The DSPT algorithm removes lots of redundant computations due to its unique updating process. In the DSPT algorithm, we check the incoming edges for those t nodes and select edges whose values display the smallest increase (or greatest decrease). The algorithm in [23] inserts those t possible edges (one edge for one node) into an edge list Q. The example in Section II and the analysis in Section III illustrate that redundant edges exist among t edges in Q. Only a small part engages in the rebuilding of a new SPT. To fully utilize this property to reduce computation in the dynamic update process, only edges having the least increased value smaller than all its ancestors' are added in Q in the DSPT algorithm in the environment of edge weight increment. Moreover, when an edge e is extracted from Q, all nodes in $des(E(e))$ (a branch on the temporarily maintained SPT) are updated once. The DSPT algorithm does not trigger any incoming edges to nodes in $des(E(e))$ to be repeatedly shown in Q. Therefore, the number of edges added to Q is far less than t even for the case of edge weight decrements. In the following algorithmic complexity analysis, it is assumed that the computation is conducted sequentially on single processor architecture.

Theorem 4: When there is only one processor to implement the dynamic update process, the complexity of the DSPT algorithm is $O(t_p t_r + E_t)$.

Proof: First let the edge list Q be maintained in a linear array. The DSPT algorithm can be divided into three parts: inserting correlated edges to the edge list Q, extracting the edge with the smallest value from Q, and updating nodes in the maintained SPT and updating Q. The asymptotic complexity of edge extraction and Q update is $O(t_p t_r)$. Each instruction *EXTRACT(Q)* and the action of removing edges from Q take time $O(t_r)$ because only t_r edges are stored in Q and we extract the edge with the smallest value. In total, we have $|t_p|$ iterations of *EXTRACT(Q)* operation and Q update behaviour. When an edge is extracted from the edge list Q, it becomes a new edge on the ultimate new SPT.

The asymptotic complexity of edge insertion and nodes update is $O(E_t)$. An updated node must have its incident edges examined. The edge with the least increased value is added to Q in the algorithm. The comparison between min_inc and $M.v.inc$ can also be viewed as examining incident edges on the maintained SPT to node v. Since the total number of edges connected to all updated nodes is E_t, there are a total of $|E_t|$ edge examinations and each takes $O(1)$ time. The number of nodes updated is t and each updating takes $O(1)$ time. Therefore, the time consumed in terms of edge insertion and nodes update is $O(E_t) + O(t) = O(E_t)$. The running time of the entire algorithm is thus $O(t_p t_r + E_t)$. □

Obviously, $t_p \leq t_r \leq t$ and in most cases the value of t_p and t_r is much smaller than t. When a network topology has

its edge weight increased, t_r equals t_p that is greatly minimized from t by the introduced node set M in the proposed DSPT algorithm. When one edge weight decreases, t_r will be the total number of edges that have a negative increased value during the updating process. $t_r \geq t_p$ because we can not forecast the relationships among edges with negative increased values and some of them might be redundantly inserted into Q. The example in Section II shows $t_p = t_r = 2$ and $t = 6$. The probability of an edge among t edges to be enqueued to Q is analyzed and the expected value of t_r is given in a general model in Section III. This probability is around $\frac{t_p}{t}$ in a practical network. In [22], there are two dynamic approaches to creating a new SPT. One is called *First Incremental Method*. Its linear Dijkstra computation time is $O(t^2 + E_t)$. The other method is *Second Incremental Method*. Its linear Dijkstra computation time is $O(t_p t + \gamma E_t)$, where γ denotes the *redundancy* factor, which represents how many times that the *Second Incremental Method* visits each node. This factor, although in practice very close to 1, can take values between 1 and E_t. The authors improved their algorithms in [23] with less computation time. But the complexity of the improved algorithms still remains $O(t_p t + E_t)$ time running, which is larger than $O(t_r t_p + E_t)$ in our DSPT algorithm.

Theorem 5: Using a binary search tree (data structure) to implement Q, the worst-case complexity of the DSPT algorithm is $O(t_r \log t_r + E_t)$.

Proof: From above analysis, we know that E_t denotes the occasions when the incident edges to updated nodes are examined. Given a binary search tree of height h, any dynamic-set operations (i.e., search, minimum, maximum, insertion, and deletion) can run in $O(h)$ time. In the DSPT algorithm, t_r edges are added into the edge list Q. The algorithm terminates when all t_r edge are removed from Q. Thus, the running time related to maintaining Q is $2t_r O(h)$. In a binary search tree $h = \log t_r$. We obtain the worst-case complexity of the DSPT algorithm by $O(t_r \log t_r + E_t)$. □

B. Application in a Graph with Negative Weight Edges

The new DSPT algorithm can also be efficiently applied to a graph with negative weight edges (no negative weight cycles). Step 1 in the DSPT algorithm should be replaced by the static Bellman-Ford algorithm [16] to obtain an old SPT. The following updating steps are the same as one edge weight change as in Algorithm 1.

Given a graph with its old SPT constructed by the bold edges as in Fig. 7(a), and the weight of edge (V, U) increased from -2 to 4, it is necessary to update the old SPT. Since the edge with weight increment is (V, U) and it is in the old SPT, all descendants of node U, which are U and Y, will be updated. The incoming edges that have the least increased value to node U, Y are edge (Z, U) with increased value of 4 $(= 6 - 2)$ and edge (X, Y) with increased value of 5 $(= 7 + (-4) - (-2))$, respectively. According to the update process, U is the parent node of Y and only one edge (Z, U) is added to the edge list Q since edge (X, Y) has a bigger increased value than edge (Z, U). In other words, our algorithm only includes one edge (Z, U) in Q. After the construction of Q, the proposed DSPT algorithm moves to phase 2 in Step 3. Edge e_1 that is (Z, U) is extracted from Q. Then, the new shortest distances for all descendants of node U (U, Y) are increased by 4 and node Z becomes the parent of node U in the new SPT. No more edges would be added to Q and Q becomes null. Thus, we update all nodes and the new SPT is in Fig. 7(b) (the bold edges construct the new SPT).

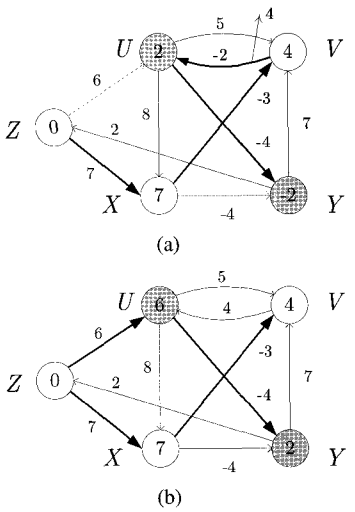


Fig. 7. (a) The weight of edge (V, U) increased from -2 to 4, and (b) the new SPT, in bold.

VI. SIMULATION RESULTS

In this section, we will compare the DSPT algorithm with the algorithm in [23] (denoted as NSPT thereafter) when one edge weight change occurs in a graph. The major difference between the two algorithms is that the proposed DSPT algorithm introduces the node list M when the weight of an edge increases and directly updates the node set $des(j)$ when the weight of an edge decreases. These techniques greatly reduce the time to enqueue and dequeue edges from Q and consequently have less time to search for the edge with the minimum value in Q. The edge list Q is essential for the performance of the whole update process.

A. Network Generator

The simulation was based on a randomly generated network, which is similar to the random topology generator (RTG) software. The simulated area for the network was a 300×300 grid. All nodes were randomly distributed in this area. The probability of an edge connecting every two nodes was determined by (1). Suppose d is the Euclidean distance between them and the probability is $P(d)$,

$$P(d) = \begin{cases} k, & \text{if } d \leq l, \\ ke^{\frac{l-d}{l}}, & \text{if } d > l. \end{cases} \tag{1}$$

The weight of each link was randomly chosen from 1 to w. If the distance between two nodes was within l, the probability that a link connects them was k. Otherwise, the probability would decay exponentially according to their distance. l is defined in terms of the normalized distance L:

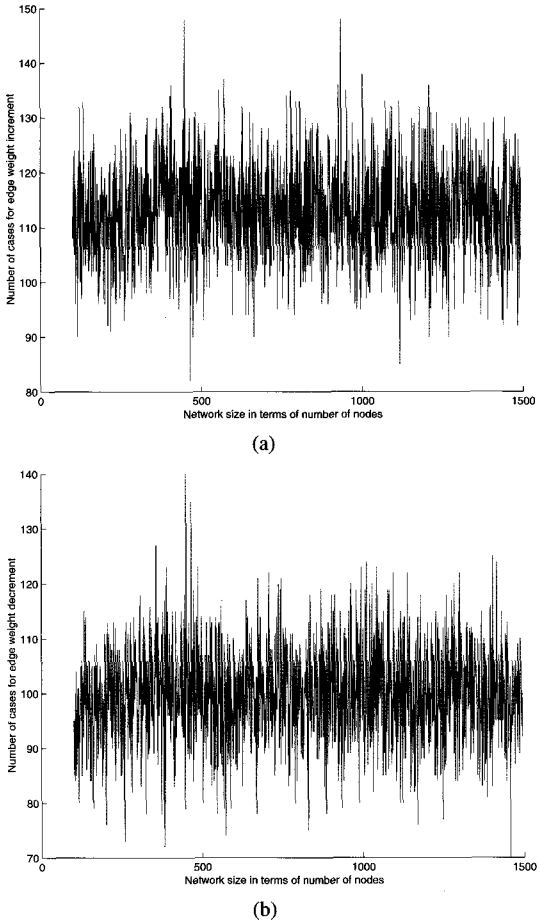


Fig. 8. (a) Number of occasions to derive a new SPT for edge weight increment, and (b) number of occasions to derive a new SPT for edge weight decrement.

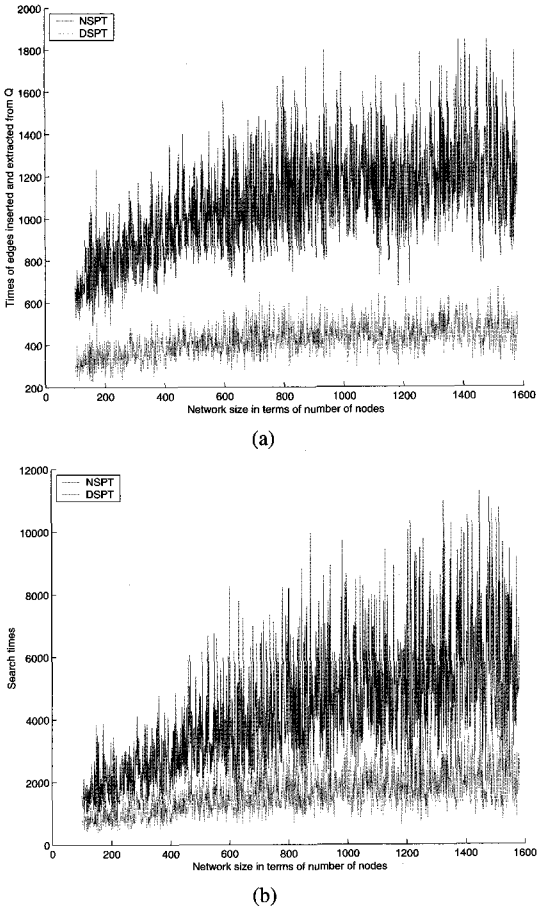


Fig. 9. (a) Comparison results for edges added and extracted from Q, and (b) comparison results of searching for the edge with the minimum value from Q.

$$l = \frac{300}{\sqrt{N}}L. \tag{2}$$

In the simulation model, k was fixed to 0.8 and L was adjusted according to the network average node degree D, which is (number of edges) $\times$ 2 $\div$ (number of nodes). We maintained D to be around 7 in different network node size. The maximum node degree was 9.

B. Event Generation

In the simulation for a specific network size, 500 continuous weight changes were tested based on an initialized graph. These changes were randomly made among all edges. The new weight of an edge could be from 1 to w and was different from the old one. This approach simulates the weight change of a randomly selected edge.

Among all 500 edge weight changes, the new SPT could in some cases be the same as the old SPT. We must update an old SPT to a new one in other cases. The criteria to carry out the dynamic update are described in Step 2 of the DSPT algorithm. We kept track of the computation time of the proposed DSPT algorithm and the NSPT [23] algorithm whenever an update was performed. The observed complexities of algorithms are compared for the main update process, which includes:

- Adding and removing edges from edge list Q.
- Searching for the edge with the minimum value from edge list Q.

In the DSPT algorithm, we introduce the node list M, which consumes no more memory space than in the NSPT. Before the adding of an edge to Q, we compare min_inc to $M.v.inc$ using the sequence of the DFS. This comparison operation can be viewed as checking incident edges connecting to the updated nodes in the SPT. Thus, it does not require any more computation time and can be incorporated in the algorithm complexity $O(E_t)$.

C. Performance Results

We used randomly generated edge weight changes to verify different algorithms. Figs. 8(a) and 8(b), respectively, illustrate the number of occasions it was necessary to update an old SPT for edge weight increment and decrement. No matter what the network size was, which could have number of nodes from 100 to 1500, the number of cases required to update an old SPT fluctuated but was around an average value. Among all 500 continuous edge weight changes in a specific network size, there were around 110 occasions of edge weight increments that we must dynamically update the old SPT. Moreover, there were about 100 occasions of edge weight decrements that required the update of outdated SPTs. Thus, there were about 290 occasions

Table 2. The performance comparison results between DSPT and NSPT in terms of edge weight w.

w	weight decrement					weight increment				
	cases	NSPT		DSPT		cases	NSPT		DSPT	
		# edge	# search	# edge	# search		# edge	# search	# edge	# search
5	78	356	297	200	219	110	1466	2395	462	661
10	94	530	761	342	667	121	1480	2339	560	819
15	103	578	640	372	537	107	1558	1953	438	568
20	111	726	1400	504	1289	114	1458	2926	530	1000

that the old SPT was still valid when an edge weight change happened.

Fig. 9 shows the comparison results between the NSPT algorithm in [23] and the proposed DSPT algorithm where preliminary simulation fixed the edge weight range w as 10 (the weight for each edge was an integer from 1 to 10) in different network sizes. The upper curves are the NSPT algorithm's computation time while the lower curves represent the results for the new DSPT algorithm. The x axis denotes the network size. The y axis shows the required computation in the dynamic update process to compute new SPTs when 500 continuous edge weight changes occurred. Fig. 9(a) illustrates how many edges that have been added to or removed from the edge list Q. The number of edges involved in the NSPT algorithm was almost 3 times of the one in the DSPT. Fig. 9(b) illustrates the computation complexity to search for the edge from Q with the minimum increased value min_inc. The new DSPT algorithm consumed about half as much computation as the NSPT algorithm did. The simulation results demonstrate that the DSPT algorithm took much less computation time than the NSPT algorithm did in the main update procedure.

When we performed the same experiment but using different edge weight range w, we notice that w also plays an important role for the computation time. Table 2 shows the results of a comparison between the DSPT algorithm and the NSPT algorithm in terms of w. The network node size is $N = 500$. The average node degree was 6.87 and one experiment comprised 500 times of edge weight change. In column "cases" under "weight decrement", we show how often it is necessary to update an old SPT to a new one among the total 500 times of edge weight changes when the edge weight decrement occurred. The data in the column "NSPT" is the simulation results for the NSPT algorithm while the column "DSPT" is for the proposed DSPT algorithm in this paper. Under column "# edge", we show the number of edges added and extracted from the edge list Q. The search times of the edge with the minimum value from Q are shown in column "# search". The data in column "weight increment" shows the simulation results when the edge weight increment occurred. The larger the weight range w was, the higher probability we had to update an outdated SPT, which implied that routers inside a computer network with heavy traffic fluctuation consumed more computation time to refresh the network's SPT (and the corresponding routing table). The DSPT algorithm always yielded less computation load to dynamically update an old SPT than the NSPT did. When the weight range w equalled 5 given weight decrements, the DSPT algorithm only introduced 56.2% $(200/356)$ # edges and 73.7% $(219/297)$ # searching times related to the edge list Q compared with NSPT. The DSPT algorithm improved performance more obviously when edges had larger weights, where DSPT engaged only 31.5% $(462/1,466)$ # edges and took 27.6% $(661/2,395)$ # searching times as NSPT did.

VII. CONCLUSION

In this paper, a new efficient DSPT algorithm has been presented to dynamically compute a new SPT in a network based on its outdated SPT. The DSPT algorithm not only minimizes the computation time, but also minimally changes the SPT structure. Thus, it removes disadvantages caused by static algorithms for updating SPT. Compared with all other known dynamic algorithms, the proposed DSPT algorithm achieves the least running time. This can be seen from the complexity analysis and experimental results.

When rebuilding a new SPT, the DSPT algorithm always updates a branch inside a maintained SPT. An efficient algorithm must select a branch cautiously. A branch in a tree is unique when the root vertex is chosen. The proposed DSPT algorithm concentrates on the edge selection to determine the root to update nodes in a branch and chooses edges which really contribute to the construction of a new SPT. With fewer edges in the constructed edge list Q, the computation time for the DSPT algorithm is greatly reduced during a dynamic update process. We use different models (from a linear graph to an arbitrary tree) to analyze the small probability of edges that involve in the rebuilding process among all incident edges that connect to all updated nodes. The new technique facilitated by the node set M in DSPT algorithm distinguishes the dynamic update process from some dynamic algorithms that update one node each time, or other dynamic algorithms that modify an entire branch but accumulate a lot of redundant edges into Q. As a result, the improved DSPT algorithm has a lower asymptotic complexity than any other known algorithm. Furthermore, this efficient DSPT algorithm can be further applied in a graph with negative weight edges, and handle simultaneous multiple edge weight changes for dynamic SPT updates.

REFERENCES

[1] J. Li, G. Mohan, E. C. Tien, and K. C. Chua, "Dynamic routing with inaccurate link state information in integrated IP-over-WDM networks," *Computer Netw.*, vol. 46, no. 6, pp. 829–851, Dec. 2004.

[2] J. Moy, "OSPF version 2," 1994.

[3] R. Rastogi, Y. Breitbart, M. Garofalakis, and A. Kumar, "Optimal configuration of OSPF aggregates," *IEEE/ACM Trans. Netw.*, vol. 11, no. 2, pp. 181–194, Apr. 2003.

[4] O. Sharon, "Dissemination of routing information in broadcast networks: OSPF versus IS-IS," *IEEE Netw.*, vol. 15, no. 1, pp. 56–65, Jan/Feb 2001.

[5] B. Fortz and M. Thorup, "Optimizing OSPF/IS-IS weights in a changing world," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 4, pp. 756–767, May 2002.
[6] W. Liu, W. Lou, and Y. Fang, "An efficient quality of service routing algorithm for delay-sensitive applications," *Computer Netw.*, vol. 47, no. 1, pp. 87–104, Jan. 2005.
[7] B. Wang and J. Hou, "An efficient QoS routing algorithm for quorumcast communication," *Computer Netw.*, vol. 44, no. 1, pp. 43–61, Jan. 2004.
[8] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Trans. Netw.*, vol. 10, no. 2, pp. 232–243, Apr. 2002.
[9] S. Nelakuditi, Z. Zhang, and D. Du, "On selection of candidate paths for proportional routing," *Computer Netw.*, vol. 44, pp. 79–102, 2004.
[10] M. Gouda and M. Schneider, "Maximizable routing metrics," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 663–675, Aug. 2003.
[11] S. Zhu and G. Huang, "A new parallel and distributed shortest path algorithm for hierarchically clustered data networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 9, pp. 841–855, Sep. 1998.
[12] B. Zhang and H. Mouftah, "Destination-driven shortest path tree algorithms," *J. High Speed Netw.*, vol. 15, no. 2, pp. 123–130, 2006.
[13] S. Gupta and P. Srimani, "Adaptive core selection and migration method for multicast routing in mobile ad hoc networks," *IEEE Trans. Parallel Distri. Syst.*, vol. 14, no. 1, pp. 27–38, Jan. 2003.
[14] M. Barbehenn and S. Hutchinson, "Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees," *IEEE Trans. Robot. Autom.*, vol. 11, no. 2, pp. 198–214, Apr. 1995.
[15] E. Dijkstra, "A note two problems in connection with graphs," *Numerical Math.*, vol. 1, pp. 269–271, 1959.
[16] R. Bellman, "On a routing problem," *Quarterly Appl. Math.*, vol. 16, pp. 87–90, 1958.
[17] A. Shaikh, R. Dube, and A. Varma, "Avoiding instability during graceful shutdown of OSPF," in *Proc. IEEE Twenty-First Annual Joint Conf. the IEEE Computer and Commun. Soc.*, 2002, pp. 883–892.
[18] V. King, "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs," in *Proc. IEEE Symp. Foundations of Computer Science*, 1999, pp. 81–91.
[19] E. Nardelli, G. Proietti, and P. Widmayer, "Swapping a failing edge of a single source shortest paths tree is good and fast," *Algorithmica*, vol. 35, pp. 56–74, 2003.
[20] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic output bounded single source shortest path problem," in *Proc. 7th Annu. ACM-SIAM Symp. Discrete Algorithms*, 1998, pp. 212–221.
[21] B. Xiao, J. Cao, Q. Zhuge, Z. Shao, and E. Sha, "Shortest path tree update for multiple link state decrements," in *Proc. IEEE GBOLECOM*, 2004, pp. 1163–1167.
[22] P. Narvaez, K. Siu, and H. Tzeng, "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Trans. Netw.*, vol. 8, no. 6, pp. 734–746, Dec. 2000.
[23] P. Narvaez, K. Siu, and H. Tzeng, "New dynamic SPT algorithm based on a ball-and-string model," *IEEE/ACM Trans. Netw.*, vol. 9, no. 6, pp. 706–718, Dec. 2001.
[24] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001, pp. 1062–1069.

Bin Xiao received the B.Sc. and M.Sc. degrees in Electronics Engineering from Fudan University, China, in 1997 and 2000, respectively, and Ph.D. degree from University of Texas at Dallas, USA, in 2003 from Computer Science. Now, he is an assistant professor in the Department of Computing of Hong Kong Polytechnic University, Hong Kong. His research interests include communication and security in computer networks, peer-to-peer networks, and wireless mobile ad hoc and sensor networks.

Jiannong Cao received the B.Sc. degree in computer science from Nanjing University, Nanjing, China in 1982, and the M.Sc. and the Ph.D. degrees in computer science from Washington State University, Pullman, WA, USA, in 1986 and 1990, respectively. He is currently a professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the director of the Internet and Mobile Computing Lab in the department. Before joining Hong Kong Polytechnic University, he was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile and wireless computing, fault tolerance, and distributed software architecture. He has published over 180 technical papers in the above areas. His recent research has focused on mobile and pervasive computing systems, developing test-bed, protocols, middleware, and applications.

Zili Shao received the B.E. degree in Electronic Mechanics from the University of Electronic Science and Technology of China, China, 1995. He received the M.S. and Ph.D. degrees from the Department of Computer Science at the University of Texas at Dallas, in 2003 and 2005, respectively. He has been an assistant professor in the Department of Computing at the Hong Kong Polytechnic University since 2005. His research interests include embedded systems, high-level synthesis, compiler optimization, hardware/software codesign, and computer security.

Edwin Hsing-Mean Sha received the M.A. and Ph.D. degree from the Department of Computer Science, Princeton University, Princeton, NJ, in 1991 and 1992, respectively. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at University of Notre Dame, Notre Dame, IN. He served as associate chairman from 1995 to 2000. Since 2000, he has been a tenured full professor in the Department of Computer Science at the University of Texas at Dallas. He has published more than 210 research papers in refereed conferences and journals. He has served as an editor for many journals, and as program committee members and chairs in numerous international conferences. He received Oak Ridge Association Junior Faculty Enhancement Award, Teaching Award, Microsoft Trustworthy Computing Curriculum Award, and NSF CAREER Award.