New Dynamic SPT Algorithm based on a Ball-and-String Model

Paolo Narváez, Kai-Yeung Siu, Hong-Yi Tzeng

Abstract—

A key functionality in today's widely used interior gateway routing protocols such as OSPF and IS-IS involves the computation of a shortest path tree (SPT). In many existing commercial routers, the computation of an SPT is done from scratch following changes in the link states of the network. As there may coexist multiple SPTs in a network with a set of given link states, such recomputation of an entire SPT not only is inefficient but also causes frequent unnecessary changes in the topology of an existing SPT and creates routing instability.

This paper presents a new dynamic SPT algorithm that makes use of the structure of the previously computed SPT. Our algorithm is derived by recasting the SPT problem into an optimization problem in a dual linear programming framework, which can also be interpreted using a ball-and-string model. In this model, the increase (or decrease) of an edge weight in the tree corresponds to the lengthening (or shortening) of a string. By stretching the strings until each node is attached to a tight string, the resulting topology of the model defines an (or multiple) SPT(s). By emulating the dynamics of the ball-and-string model, we can derive an efficient algorithm that propagates changes in distances to all affected nodes in a natural order and in a most economical way. Compared with existing results, our algorithm has the best-known performance in terms of computational complexity as well as minimum changes made to the topology of an SPT. Rigorous proofs for correctness of our algorithm and simulation results illustrating its complexity are also presented.

I. Introduction

In today's Internet, each datagram is forwarded by a router based on a forwarding table. Routing protocols are employed to exchange topology information among routers to facilitate the construction of forwarding tables. Examples of widely used link-state based routing protocols include Open Shortest Path First (OSPF) and IS-IS [16], [22]. With these routing protocols, each link is associated with a cost (weight) and routers exchange link state information so that each router in a routing area has a complete description of the network topology. Using the link costs, each router computes a path with minimum cost from itself to each other router in the region, yielding a shortest path tree (SPT). The corresponding SPT is then used to build a forwarding table which contains routing information for forwarding a datagram to its destination along the shortest path.

When the topology in a routing area changes (e.g., a link fails, recovers, or changes its routing cost), every router in the region is notified of the change. After updating the corresponding topology changes in its link state database, each router recomputes its SPT. In most of today's commercial routers, this recomputation is done by deleting the current SPT and recomputing it from scratch by using the well known Dijkstra algorithm [20].

Usually, after some changes in the link states, the topology of the new SPT does not differ significantly from the old one. (In fact, most often it does not change at all.) *Static algorithms* that recompute the SPT from scratch are clearly inefficient because they do not take advantage of available information about the outdated SPT. Another drawback of using static SPT algorithms is that there may coexist multiple routes of the same shortest distance from one router to another; by recomputing a new SPT from scratch, a router may unnecessarily choose a different route of the same distance to forward its packets. This in turn may cause the router to change many entries in its forwarding table frequently, increasing the risk of routing errors or router failures.

In addition to redundant updates in forwarding tables, unnecessary changes in SPT also cause undesirable fluctuation of traffic load on a given route. (For an excellent discussion on instability of other routing protocols in the Internet, see [21], [23].)

In this paper, we will present an efficient algorithm that can dynamically update the SPT following changes in the link states. Our dynamic algorithm uses information of the outdated SPT and updates only the part of the SPT that is affected by the change. Our design objectives for this dynamic algorithm are twofold. The first objective is to minimize the computational complexity required to update an SPT. The second objective is to maintain routing stability by making minimal changes to the topology of an existing SPT.

In the next section, we shall further discuss some prior works. Section III sets up an example to illustrate the problem and to give some intuition on the algorithm. Section IV introduces graph-theoretic definitions and notations to be used in the paper. In Section V, we describe our algorithm in detail. Section VI proves how the algorithm works, proves its correctness, and analyzes its complexity. Section VII presents simulation results illustrating the complexity of the new algorithm. Concluding remarks are given in VIII.

II. Related Works

The problem of routing in data networks has been a subject of continual research interest for the past two decades [1], [4], [10], [15], [25], [26] and many routing pro-

P. Narváez and K.-Y. Siu are with the d'Arbeloff Laboratory for Information Systems & Technology, Massachusetts Institute of Technology, Cambridge, MA. H.-Y. Tzeng is with the High Speed Networking Department, Bell Labs, Lucent Technologies, Holmdel, NJ. Email: pnarvaez@list.mit.edu, siu@list.mit.edu, htzeng@lucent.com

0-7803-5420-6/99/$10.00 (c) 1999 IEEE

tocols have been studied and used in practical networks [6], [12], [16], [24], [27]. Recently, the stability issues in Internet routing have attracted much attention [21], [23]. In fact, our interest in dynamic SPT algorithms was partly motivated by the problem of routing instability in the Internet.

To our knowledge, the earliest work on dynamic SPT algorithms that appeared in the literature was [18]. A dynamic SPT algorithm for the Arpanet was also discussed in [12]. This algorithm is basically a generalization of Dijkstra's static SPT algorithm [20]. However, the work [12] contains no analytical proofs nor simulation results.

Two semi-dynamic algorithms to update SPTs are presented in Franciosa et al. [7]. Each semi-dynamic algorithm handles the case when the change in an edge weight is either positive or negative. The algorithms are dynamic versions of the Dijkstra algorithm, but can only handle integer edge weights. The paper only analyzes the worst-case complexity of the algorithm in terms of the total size of the graph rather than the number of nodes whose distances have changed (denoted by δ_n in this paper). Therefore, the complexity cannot be shown to be better than that of a static algorithm.

Frigioni et al. [9], [8] present an algorithm similar to the one in [7] but it allows the edge weights to be non-integers. The algorithm is also a dynamic version of the Dijkstra algorithm. The complexity of the algorithm in [8] is analyzed in terms of the number of nodes whose optimal distance to the source change. Their algorithm also introduces some optimization for cases where there is a large number of edges (where the node degree is not bounded) in order to bound the worst-case complexity in such cases.

Our previous work on dynamic SPT algorithms [13] introduces a new framework to convert various conventional static SPT algorithms into dynamic ones. By adjusting various parameters, the framework can produce dynamic versions of not only the Dijkstra algorithm, but also the Bellman-Ford [2] and D'Esopo-Pape [3] algorithms. This framework characterizes dynamic SPT algorithms in a unified way and establishes a general proof of correctness for all the algorithms that fit the framework. The difference between each of the specific dynamic algorithms in this framework is the way in which nodes are stored and extracted from a general set. The work in [13] only considers extraction criteria that are derived from the search criteria of the corresponding static SPT algorithms. The new algorithm presented in this paper fits in this framework but uses an original search criterion, not derived from any existing static SPT algorithm, and is provably better than the existing dynamic SPT algorithms. An extended version of this paper is can be found in [14].

III. Example

We introduce the SPT recomputation problem with a simple example. Figure 1 shows a network comprising six nodes, where each node is labeled with a letter (A to F). The links between the nodes are illustrated with either dotted or solid lines, and the number next to each line is the weight of that edge. When the link between nodes A and C has a weight of 8, the SPT rooted at node A consists of the solid lines in the figure; the shortest distance of each node is the number inside each node.

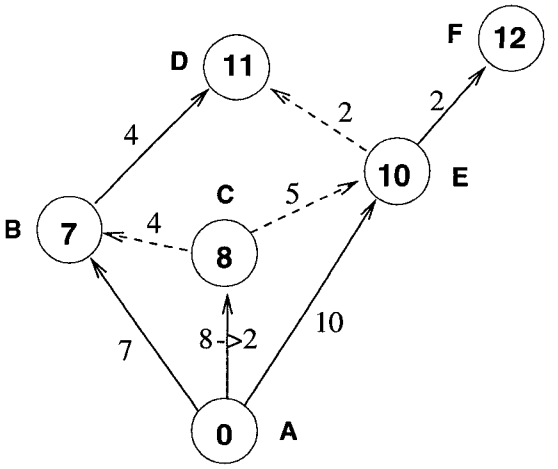

Fig. 1. Shortest path tree before link change.

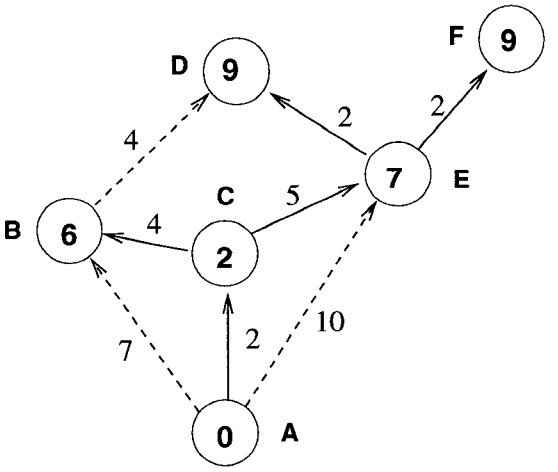

Fig. 2. Shortest path tree after link change.

At some point in time, the edge between nodes A and C changes its weight from 8 to 2. This weight change will significantly alter the structure of the SPT. The new SPT is shown in Figure 2. We will follow step by step the different computations that one needs to do in order to dynamically change the tree corresponding to the old SPT into the new SPT. The entire computation will be done using two different methods. The first method uses the familiar notion of shortest distance as used by the Dijkstra algorithm (and its dynamic versions). The second method employs a new notion (that of maximum decrement) which will prove to be more efficient for this task.

First of all, it is clear that the edge between A and C (which decreases its weight) will remain in the SPT and that the shortest distance of node C will decrease to 2. Because node C has a shortest distance of 2, it can further decrease the potential shortest distance of nodes B (from 7 to 6) and E (from 10 to 7) by becoming their parent in the new tree. Now we have the choice of continuing the modification of the tree by following node B or following

node E.

If we give preference to nodes with the smallest potential distances (similar to the Dijkstra algorithm), we will choose node B because its potential shortest distance 6 is smaller than the potential shortest distance of node E (7). When we are at node B, its parent in the tree is changed from A to C and its distance label is reduced to 6. Since node B has a child D, the distance label of D is now reduced from 11 to 10. Since D does not have a directed edge to another neighbor, its distance change does not affect any other node. Then, we return to node E and decrease its distance label to 7 by making it a child of C. Since F is the child of E, its distance label is reduced from 12 to 9. Finally, node D decreases its distance label from 10 to 9 by becoming a child of E.

On the other hand, if after C's distance label change, we decide to modify E before B (E has the the greatest distance label decrease), the total computation becomes simpler. In this case, the parent of E becomes C and its distance label becomes 7. The distance label of its child F becomes 9. We now have a choice of modifying node B or node D. Since node D has the largest potential shortest distance decrease (from 11 to 9, rather than from 7 to 6), node D will become a child of node E and acquire a distance label of 9. Finally, B will become a child of C and acquire a distance label of 6.

Notice that when nodes E and D were chosen ahead of node B, the total computation is smaller. Specifically node D is visited only once and its distance label changes directly from 11 to 9, rather than changing in two steps first from 11 to 10 and then from 10 to 9. The notion that we have used in the second computation is to give priority to nodes whose potential shortest distance decreases the most (or increases the least). It is by generalizing this notion, that we obtain the general algorithm described in this paper.

IV. Problem Formulation

A. Shortest Path Tree

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a directed graph where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges in the graph. Let N denote the total number of nodes in $\mathcal{V}$ and E the total number of edges in $\mathcal{E}$. The graph $\mathcal{G}$ consists of a root node denoted by n_0, and all other nodes $n_1, ..., n_{N-1}$ are assumed to be reachable from n_0 by a directed path in $\mathcal{G}$. Each edge e_i $(i = 0, ..., E-1)$ has an associated weight (cost) of w_i, which is assumed to be positive.

A rooted tree $\mathcal{T}$ is a subgraph of $\mathcal{G}$ such that every node in $\mathcal{G}$ is reachable from the root n_0 through a unique directed path using only edges in $\mathcal{T}$. The length of a path is the sum of the weights of the edges in that path. The distance of a node n_i $(i = 1, ..., N-1)$ in $\mathcal{T}$ is the length of the directed path in $\mathcal{T}$ connecting n_0 to n_i. The shortest distance of a node n_i is the length of a shortest path in $\mathcal{G}$ connecting connecting n_0 to n_i. A tree $\mathcal{T}$ is said to be a shortest path tree (SPT) for $\mathcal{G}$ if the distance of every node n_i in $\mathcal{T}$ is the shortest distance of n_i in $\mathcal{G}$. Note that there can be multiple SPTs in a graph, but the shortest distance of any node is unique.

B. Linear Programming Framework

It is well known that the SPT problem can be formulated as an optimization problem in a linear programming framework.[1] To see this, for every directed edge e_i connecting node n_j to node n_k, we create a (row) vector $\bar{v}_i$ of dimension N (number of nodes) which contains all 0's except a '-1' at the j^{th} entry and a '$+1$' at the k^{th} entry. The concatenation of all such vectors $\bar{v}_i$ results in a connectivity matrix $\mathbf{A}$ (of dimension $E \times N$). Let $\bar{u}$ be an N-dimensional (column) vector such that its i^{th} entry u_i corresponds to a potential distance of node n_i. If u_i^* is the shortest distance of node n_i ($u_0^* = 0$), then clearly $u_j^* - u_i^* \leq w_\ell$, where w_ℓ is the weight of the edge connecting n_i to n_j. Thus, if $\bar{w}$ denotes the (column) vector of edge weights and $\bar{u}^*$ represents the vector of shortest distances, then $\bar{u}^*$ must satisfy the following *feasibility constraint:*

$$\mathbf{A}\bar{\mathbf{u}} \leq \bar{\mathbf{w}} \; ; \qquad \mathbf{u_0} = \mathbf{0} \; . \tag{1}$$

Clearly, there are infinitely many vectors $\bar{u}$ that satisfy the above matrix inequality and only one of them corresponds to the vector of shortest distances $\bar{u}^*$. However, it can be shown that $\bar{u}^*$ can be found by solving the following linear optimization problem:

$$max(\bar{b}^T \bar{u}) \tag{2}$$

subject to the feasibility constraint given by Eqn. (1), where $\bar{b}$ is an N-dimensional vector with each entry equal to 1. Note that $\bar{b}^T \bar{u}$ represents the sum of the potential distances of all nodes.

The feasibility constraint of Eqn. (1) ensures that the difference of the distances u_i and u_j associated with nodes n_i and n_j is always no greater than the weight of the edge connecting the nodes. This is clearly the case in an SPT where the distance u_i is the length of a shortest path from n_0 to n_i. Interestingly, the vector of shortest distances can be obtained by maximizing the sum of the potential distances u_i of all nodes, while respecting the feasibility constraint.

In order to determine an SPT from the solution $\bar{u}^*$ of the above optimization problem, one can substitute $\bar{u}^*$ into the feasibility constraint $\mathbf{A^T}\bar{\mathbf{u}} \leq \bar{\mathbf{w}}$. If the row i of the matrix inequality is an equality, it means that edge e_i can be used in an SPT.

C. Physical Interpretation of SPT in a Ball-and-String Model

There is an intuitive way of interpreting the linear programming formulation of the SPT problem using a ball-and-string model. Consider a collection of balls, each of which corresponds to a node in $\mathcal{V}$. Each edge e_i with weight $w_i > 0$ connecting nodes n_j and n_k in $\mathcal{G}$ is associated with

[1] There are both primal and dual formulations of the SPT problem in a linear programming framework. We present here the dual formulation to motivate the ball-and-string model.

an inelastic string of length w_i connecting the balls representing n_j and n_k. Let u_i denote the Euclidean distance between the balls representing n_0 and n_i. In the following, the terms "ball" and "node" will be used interchangeably.

Since none of the strings is allowed to be stretched beyond its length, it is clear that in this illustration, the vector of Euclidean distances $\bar{u}$ always satisfies the feasibility constraint (1). Moreover, a string becomes "tight" as soon as the Euclidean distance between the two nodes attached equals its length. The optimal solution of (2) is then achieved by sequentially pulling the balls (in the same direction) so as to maximize their distances from the ball representing the root node n_0. In other words, when the balls are stretched out from n_0 as much as possible, their distances form a vector $\bar{u}^*$ that corresponds to the optimal solution. If there is only one SPT in the graph, the strings that are tight will then define the SPT. If there are multiple SPTs, each string that is tight will correspond to an edge that is part of some SPT.Our algorithm can be easily extended to compute all such edges.

In fact, the execution of the original (static) Dijkstra algorithm can also be interpreted in terms of the above ball-and-string model. Let us imagine that the ball corresponding to the root node n_0 is anchored at a fixed position, while other balls are "floating" in the sense that their attached strings are not tight. All the balls are initially in the same position. In the first iteration, the ball connected to n_0 with the shortest string is first pulled away from n_0 (always in the same direction) and will be anchored along with n_0.

In the second iteration, the floating ball that has a string connected to the anchored nodes and is closest (through that string) to n_0 is then pulled away (along the same direction as before) until that string becomes tight and the ball is anchored. The procedure is repeated until all floating balls are anchored. The Euclidean distance of an anchored ball from the root ball then represents its shortest distance.

V. Dynamic SPT Algorithm

A. Algorithm Execution in the Ball-and-String Model

Recall that in computing a new SPT from an outdated SPT, our objectives are to minimize the computational complexity and the changes to the topology of the SPT. In fact, the ball-and-string model gives us useful insights into how an efficient dynamic SPT algorithm with these objectives should be designed. Before we present a formal specification of our algorithm, we next describe its execution in terms of the ball-and-string model.

Let us now imagine that the root node n_0 is anchored at a fixed position and the rest of the nodes are allowed to drop under gravity. Since gravity pulls down on these nodes as much as possible, the resulting Euclidean distance of a node n_i from the anchored root node is equal to the shortest distance of n_i. When the length of a string increases, the ball attached to the lower end of the string (as well as others) will fall down for some distance until at least one of its attached string becomes tight. When the length of a string decreases, the ball attached to the lower end of the string (as well as others) will be raised accordingly. After the length of the string is modified, the hanging ball-and-string model always represents the set of valid SPTs.

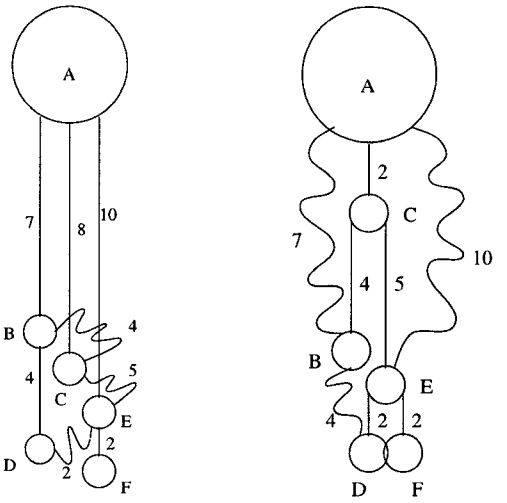

Fig. 3. The ball-and-string models for the networks in Fig. 1 (left) and Fig. 2 (right).

Figure 3 illustrates the ball-and-string models for the networks in Figure 1 and Figure 2. The straight lines represent the tight strings and the curly lines represent the loose strings. The shortest paths from node A to all other nodes are composed of the straight lines in the figures. When the link distance between A and C is reduced from 8 to 2, node C is "raised" to a higher level. It can be seen from the figure that some loose strings in the original model now become tight and that the affected nodes rearrange themselves in a natural way into their optimal positions. When the length of the string between A and C is reduced gradually, the sequence of distance changes that takes place with this ball-and-string model corresponds to the optimal execution we discussed for our example at the end of section III.

The execution of our algorithm simulates the dynamics of the balls when a string gets longer or shorter. When there are multiple strings that change lengths, the algorithm separates the length increments and decrements of the strings into two sets, and then simulates the dynamics for one set after the other. Within each set, the changes in the string lengths are simulated as if they took place at the same rate. In other words, all the balls that will change distances will move at the same rate. The efficiency of our algorithm is due to the fact that the changes in distances are propagated to all affected nodes in a natural order and in a most economical way.

If the directed edge e_k connecting n_i to n_j has a different weight than the edge e_l connecting n_j to n_i, then we need to stretch our imaginations a little and change the inelastic string for a magic elastic string. This elastic string representing the two edges has a nominal zero length but can be stretched up to a certain distance when pulled. When n_i is above n_j (due to gravity), the elastic will stop stretching at length w_k; when n_j is above n_i, it will stop stretching

at length w_l.

B. Algorithm Specification

For each directed edge $e \in \mathcal{E}$, we use $\mathsf{W(e)}$ to denote the weight (distance) associated with e, $\mathsf{S(e)}$ and $\mathsf{E(e)}$ to denote respectively the source node and the end node of e. The length or distance of a directed path is the sum of weights of the edges on the path. Given a set of nodes $\mathcal{N} \subseteq \mathcal{V}$, we associate with it two sets of edges: $\mathsf{I}(\mathcal{N}) = \{\mathsf{e} \in \mathcal{E} \mid \mathsf{E(e)} \in \mathcal{N}\}$ (the set of edges directed *into* the nodes in $\mathcal{N}$) and $\mathsf{O}(\mathcal{N}) = \{\mathsf{e} \in \mathcal{E} \mid \mathsf{S(e)} \in \mathcal{N}\}$ (the set of edges directed *out* of the nodes in $\mathcal{N}$). The variable $root(\mathcal{G})$ denotes n_0, the root node of the SPT. These parameters only depend on the topology of the network, and their values do not change during the execution of the algorithm.

A tree data structure, denoted by $\hat{\mathcal{T}}$, is maintained by our algorithm to keep track of an existing potential shortest path tree. In particular, every node n in the graph $\mathcal{G}$, along with its parent attribute $\mathsf{P(n}, \hat{\mathcal{T}})$, a list of children $\mathcal{C}(n, \hat{\mathcal{T}})$, its distance attribute $\mathsf{D(n}, \hat{\mathcal{T}})$, and a Boolean attribute $\mathsf{V(n}, \hat{\mathcal{T}})$ with value *floating* or *anchored*, is present in $\hat{\mathcal{T}}$. This data structure $\hat{\mathcal{T}}$ changes progressively during the computation, and when the execution of the algorithm is completed, it will represent an SPT and each node n will have its Boolean attribute $\mathsf{V(n}, \hat{\mathcal{T}}) = \mathsf{anchored}$.

In addition to the data structure $\hat{\mathcal{T}}$, our algorithm also maintains a list $\mathcal{Q}$ that temporarily contains a subset of nodes together with three attributes. In particular, each element in $\mathcal{Q}$ is of the form $\{n, (p, d, \Delta)\}$, where p denotes a potential parent for node n, d denotes a potential distance for node n, and Δ denotes the potential distance change for n.

The instruction ENQUEUE$(\mathcal{Q}, \{n, (p, d, \Delta)\})$ adds one more element to $\mathcal{Q}$. If node n is already in $\mathcal{Q}$, the new attributes (p, d, Δ) will replace the old ones only if the new Δ is smaller than the old one. At any instant, only one set of attributes is maintained for each node in $\mathcal{Q}$. When an EXTRACTMIN$(\mathcal{Q})$ instruction is executed, the element with the smallest Δ is selected and removed from $\mathcal{Q}$ If there is more than one element with the minimum Δ, the smallest distance d is used as a tie breaker. If more than one element has the smallest d among the elements with the smallest Δ, then any of these elements can be chosen by EXTRACTMIN. A REMOVE$(n, \mathcal{Q})$ operation removes the entry for node n from the list $\mathcal{Q}$.

The algorithm also uses two more auxiliary subroutines. The function $\mathsf{B_{max}(n}, \hat{\mathcal{T}})$ denotes the set of nodes (including n) that are descendents of n in the tree $\hat{\mathcal{T}}$. This set can be easily computed by descending directly down $\hat{\mathcal{T}}$. Likewise, the function $\mathsf{B_{val}(n}, \hat{\mathcal{T}})$ denotes the subset of $\mathsf{B_{max}(n}, \hat{\mathcal{T}})$ that is reachable from n through some path of anchored nodes. It can be easily computed by descending directly down $\hat{\mathcal{T}}$ and stopping when a node found in the path is floating.

The algorithm contains an initialization procedure (step 1) and an iterative loop (steps 2-4). If there are multiple edge weight changes, only the weight increments or the weight decrements can be handled at a time.

STEP 1: Initialization

(A) *Static Version*
$\forall\ (n \in \mathcal{V}) \neq root(\mathcal{G})$
  $\mathsf{P(n}, \hat{\mathcal{T}}) \leftarrow \emptyset$
  $\mathsf{D(n}, \hat{\mathcal{T}}) \leftarrow 0$
  $\mathsf{V(n}, \hat{\mathcal{T}}) \leftarrow$ floating
  $\mathcal{C}(n, \hat{\mathcal{T}}) \leftarrow \emptyset$
ENQUEUE $(\mathcal{Q}, \{root(\mathcal{G}), (\emptyset, 0, 0)\})$
Go To Step 2

(B) *Dynamic Version*
(applicable when an outdated SPT exists)
Case 1 (Edge $e_i \in \mathcal{E}^+$ increases its weight by Γ_i):
$\hat{\mathcal{N}} \leftarrow \emptyset$
$\forall e_i \in \mathcal{E}^+$
$\mathsf{W(e_i)} \leftarrow \mathsf{W(e_i)} + \Gamma_i$
  if $\mathsf{S(e_i)} = \mathsf{P(E(e_i)}, \hat{\mathcal{T}})$
    $\mathcal{N} \leftarrow \mathsf{B_{val}(E(e_i)}, \hat{\mathcal{T}})$
    $\forall\ n \in \mathcal{N}$
      $\mathsf{V(n}, \hat{\mathcal{T}}) \leftarrow$ floating
    $\hat{\mathcal{N}} \leftarrow \hat{\mathcal{N}} \cup \mathcal{N}$
$\forall\ e \in \mathsf{I}(\hat{\mathcal{N}})$
  if $\mathsf{V(S(e)}, \hat{\mathcal{T}}) = \mathsf{anchored}$
    $newdist = \mathsf{D(S(e)}, \hat{\mathcal{T}}) + \mathsf{W(e)}$
    ENQUEUE$(\mathcal{Q}, \{\mathsf{E(e)}, (\mathsf{S(e)}, \mathsf{newdist}, \Delta)\})$
Go To Step 2

Case 2 (Edge $e_i \in \mathcal{E}^-$ decreases its weight by Γ_i):
$\forall e_i \in \mathcal{E}^-$
$\mathsf{W(e_i)} \leftarrow \mathsf{W(e_i)} - \Gamma_i$
  $newdist \leftarrow \mathsf{D(S(e_i)}, \hat{\mathcal{T}}) + \mathsf{W(e_i)}$
  if $newdist < \mathsf{D(E(e_i)}, \hat{\mathcal{T}})$
    $\Delta \leftarrow newdist - \mathsf{D(E(e_i)}, \hat{\mathcal{T}})$
    ENQUEUE $(\mathcal{Q}, \{\mathsf{E(e_i)}, (\mathsf{S(e_i)}, \mathsf{newdist}, \Delta)\})$
Go To Step 2

STEP 2: Node Selection
if $\mathcal{Q} = \emptyset$
  `Terminate`
else
  $\{y, (x, d, \Delta)\} \leftarrow$ EXTRACTMIN$(\mathcal{Q})$
  $\mathcal{C}(x, \hat{\mathcal{T}}) \leftarrow \mathcal{C}(x, \hat{\mathcal{T}}) + \{y\}$
  $\mathcal{C}(\mathsf{P(y,}\hat{\mathcal{T}}), \hat{\mathcal{T}}) \leftarrow \mathcal{C}(\mathsf{P(y,}\hat{\mathcal{T}}), \hat{\mathcal{T}}) - \{\mathsf{y}\}$
  $\mathsf{P(y,}\hat{\mathcal{T}}) \leftarrow \mathsf{x}$
  $\mathcal{N} \leftarrow \mathsf{B_{max}(y,}\hat{\mathcal{T}})$

STEP 3: Distance Update
$\forall\ n \in \mathcal{N}$
  $\mathsf{D(n}, \hat{\mathcal{T}}) \leftarrow \mathsf{D(n}, \hat{\mathcal{T}}) + \Delta$
  $\mathsf{V(n}, \hat{\mathcal{T}}) = \mathsf{anchored}$
  if $n \in \mathcal{Q}$ AND $\mathsf{P(n}, \hat{\mathcal{T}}) \neq \mathsf{P(n}, \mathcal{Q})$
    REMOVE$(n, \mathcal{Q})$

STEP 4: Node Search

$\forall\, e \in O(\mathcal{N})$
if $(D(E(e), \hat{T}) > D(S(e), \hat{T}) + W(e))$ OR $(V(E(e)) = \text{floating})$
$newdist \leftarrow D(S(e), \hat{T}) + W(e)$
$\Delta \leftarrow newdist - D(E(e), \hat{T})$

ENQUEUE $(\mathcal{Q}, \{E(e), (S(e), \text{newdist}, \Delta)\})$
Go to Step 2

C. Informal Discussion of the Algorithm

To help understand the algorithm, here we shall give an informal discussion of its executions. As the execution is the same for the static version and the dynamic version after initialization, we shall only discuss the dynamic version, since the initialization of the static version is trivial. Also note that the execution of the static version of this algorithm is equivalent to Dijkstra's algorithm.

First, the goal of the initialization phase is to identify those nodes that may be affected by the changes in the link states. The potentially affected nodes are those that are no longer connected to the root through the same shortest path as before. Moreover, in the initialization, we only want to select a minimal set of such nodes whose changes in distance will be subsequently propagated to their descendents in the original tree, which can be updated.

It is assumed that in the existing (outdated) SPT, all nodes were anchored. The potentially affected nodes when there are edge weight increments will be marked *floating* in the initialization. As mentioned earlier, the algorithm works by first updating the SPT with all edges that increase their weights (case 1) and then updating the SPT with all edges that decrease their weights (case 2).

In case 1, after updating each edge e_i with the new increased weight, we check if the edge is in the existing SPT. If it is, we select its end node $E(e_i)$ and all descendent nodes of $E(e_i)$ that are reachable via anchored nodes in the existing SPT. All such nodes will be marked floating and included in a set $\hat{\mathcal{N}}$ for further updating. The intuition is that the set $\hat{\mathcal{N}}$ covers all the affected nodes. Moreover, we only have to consider those nodes that are directly attached to anchored nodes in the existing SPT (their descendent nodes will be updated in subsequent steps of the algorithm).

Now for each anchored node that is attached to a link directed into a floating node in $\hat{\mathcal{N}}$, we compute the potential distance *newdist* of the floating node by adding the distance of the anchored node and the weight of the edge connecting the two nodes. Moreover, we keep track of the difference Δ between the old distance and the potential new distance. The floating node with Δ, its potential distance *newdist*, and its potential new parent (the anchored node) will be enqueued in the list $\mathcal{Q}$.

Case 2 is a bit simpler in the initialization step. After updating each edge e_i with the new decreased weight, we compute the potential new distance *newdist* of its end node $E(e_i)$ by adding the distance of its source node $S(e_i)$ and the new weight $W(e_i)$. And if this potential new distance is in fact smaller than the old distance of $E(e_i)$, then we enqueue to the list $\mathcal{Q}$ the node $E(e_i)$ with its potential new parent $S(e_i)$, its potential new distance *newdist*, and the difference Δ between its old distance and *newdist*.

After the initialization step, we extract from the list $\mathcal{Q}$ in step 2 an element with the smallest (least positive or most negative) Δ. In other words, we select a node with the smallest increase in distance (in case 1) or largest decrease in distance (in case 2). This selected node is then updated with its new parent indicated in $\hat{\mathcal{T}}$ and the structure of $\mathcal{T}$ is modified accordingly to reflect the new child-parent relation. Moreover, the selected node together with all its descendent nodes (denoted by the set $\mathcal{N}$) in the existing tree $\hat{\mathcal{T}}$ will now have their correct distances updated as their old distances incremented by Δ (step 3). In addition, these nodes will be marked anchored. And if any such node is already in the list $\mathcal{Q}$ (unless its parent attribute in $\mathcal{Q}$ is the same as its parent in $\hat{\mathcal{T}}$, indicating that further change will take place) it will be removed from $\mathcal{Q}$ and no longer considered by the algorithm.

In step 4, we consider each node $E(e_i)$ that is attached to an edge e_i directed out of a node $S(e_i)$ in $\mathcal{N}$. If the potential new distance *newdist* of $E(e_i)$ (which is equal to $W(e_i)$ plus the distance of $S(e_i)$) is smaller than its old distance, or if $E(e_i)$ has been marked floating, we then enqueue in the list $\mathcal{Q}$ the node $E(e_i)$ with its potential new distance, change in distance, and its potential new parent $S(e_i)$ for further updating. After this step is completed, the execution of steps 2 and 3 will be repeated until the list $\mathcal{Q}$ is empty.

VI. Algorithm Correctness and Complexity

A. Correctness

In this paper, we will only sketch the proof of correctness of the algorithm and state some of its properties. A complete proof and analysis can be found in [14].

The object of the algorithm is to compute a new SPT after the weights of the graph change. Let $\mathcal{G}$ and $\mathcal{G}'$ be the old and new network respectively, and Γ is the set of weight changes that occur in $\mathcal{G}$. For brevity, we will refer to the event when the topology of the network changes as β. The input to the algorithm is $\mathcal{G}$, Γ, as well as an old SPT tree $(\mathcal{T})$ valid for $\mathcal{G}$. The desired output of the algorithm is a new SPT tree $(\mathcal{T}')$ valid for $\mathcal{G}'$. Furthermore, when there are multiple new SPTs for $\mathcal{G}'$, the desired output of the algorithm $\mathcal{T}'$ is the new SPT that differs by the minimum number of edges from $\mathcal{T}$ [2].

We first take a look at how the optimal SPT tree is affected by the weight changes. We separate the full set of nodes in the graph $\mathcal{N}$ into k smaller subsets $\mathcal{K}_1$, $\mathcal{K}_2$, ... $\mathcal{K}_k$. Within each subset (referred to as branch), the structure of the SPT can remain unchanged after the link weight changes. The branch containing the root node in $\mathcal{N}$ will be called the root branch $\mathcal{K}_0$. More formally, we use the following definition:

Definition 1: Two nodes n_j and n_k are said to be in the same branch $\mathcal{K}_i$ if they are connected in *both* the old

[2] $\mathcal{T}'$ is not necessarily unique since there can still be several SPTs that differ from $\mathcal{T}$ by the minimum number of edge changes

and new optimal trees by some edge that does not change weight during event β.

This definition of can be used transitively to find all the nodes in a given branch. For a given old $\mathcal{T}$ and new $\mathcal{G}'$, the definition of branch gives *unique* sets of nodes. This can be easily seen graphically by using the ball and string model. We first represent $\mathcal{G}'$ using the ball and string model, and then discard all the string (edges) that are not tight. The remaining tight edges gives us all the possible edges that can be part of some SPT for $\mathcal{G}'$. Clearly this set of edges is unique. If we now overlap the old SPT $\mathcal{T}$ onto this set of edges, we immediately obtain our branches $\mathcal{K}_i$. The branches will consists of the nodes that are connected by overlapping edges of $\mathcal{T}$ (that did not change weight) and the tight edges of the ball-and-string model.

We can think of a branch $\mathcal{K}$ as a subtree of the original SPT ($\mathcal{T}$) that does not change internally but only changes its position with respect to the rest of the tree. We show in [14] that our algorithm recomputes the SPT by rearranging these branches in the minimum number of iterations, without modifying their internal structure. At each iteration of the algorithm, a new branch is identified and placed in the correct position. If there are k branches in the graph (excluding the root branch), after k iterations, the algorithm is done.

B. Complexity

From the proof of correctness, the complexity of the algorithm follows directly. For the single edge change, as well as well for case 1 and 2 of the multiple edge changes, the algorithm follows an easily determined number of steps.

Every edge departing from a node in a non-root branches is visited exactly once. Furthermore, for every branch, its mini-root needs to be extracted from $\mathcal{Q}$. The maximum size of $\mathcal{Q}$ is the total number of nodes in the non-root branches. The maximum number of insertions and key decrements in $\mathcal{Q}$ is the total number of edges whose source is a node in a non-root branch.

We let δ_c be the number of branches defined by the old tree $\mathcal{T}$ and the new graph $\mathcal{G}'$ (excluding the root branch). Let δ_n be the total number of nodes in these branches, and δ_e be the total number of edges whose sources correspond to these nodes. The number of operations of the algorithm can be expressed as follows:

- Algorithm iterations: δ_c
- Extractions from $\mathcal{Q}$: δ_c
- Node visits: δ_n
- Maximum size of $\mathcal{Q}$: δ_n
- Maximum number of insertions into $\mathcal{Q}$: δ_n
- Maximum number of removals from $\mathcal{Q}$: $\delta_n - \delta_c$
- Maximum number of key decrements in $\mathcal{Q}$: δ_e

The exact complexity of the algorithm depends on how the underlying priority queue $\mathcal{Q}$ is implemented. The simplest way of implementing it is using a linked list. A more efficient, as well as practical, way involves using binary heaps. In theory, using Fibonacci heaps should yield the lowest asymptotic complexity; however, in practice they are inefficient because of the large overhead involved in using them. The following are the total resulting complexities:

- Linked List: $O(\delta_e + \delta_c\delta_n)$
- Binary Heap: $O(\delta_e log(\delta_n))$
- Fibonacci Heap: $O(\delta_e + \delta_n log(\delta_n))$

Note that the best assymptotic complexity of existing dynamic algorithms is that of dynamic Dijkstra and its variations [13]. For linked list structure, the above complexity is better than for dynamic Dijkstra ($(\delta_n)^2$). For the heap structures, the above complexity matches that of Dynamic Dijkstra. Nevertheless, the number of extractions δ_c performed by our algorithm is smaller than for any known algorithm that satisfies the above upper bounds (all algorithms that satisfy this bound, have extraction operations from a priority queue). Even though in the worst-case the smaller number of extractions does not improve the upper bound, in practice a lower number of extractions means that less nodes need to enter the priority queue, and the number of operations performed on the queue is smaller. The number of insertions can be as small as δ_c, while for all the known algorithms that satisfy the above bound, the number of insertions is at least δ_n.

Furthermore, the highest key (in this case Δ) used in the priority queue of the new algorithm is at most the maximum weight of an edge. For dynamic Dijkstra, the maximum key can be as much as the length of the path from one side of the network to the other. Because the keys are much smaller, it is much easier to implement the priority queue using bucket search.

VII. Simulation Results

In this section we investigate via simulations the complexity of our new shortest path algorithm. The algorithm is compared with other existing dynamic SPT algorithms. The existing algorithms used for comparison purposes are the dynamic versions of the Bellman-Ford, D'Esopo-Pape, and Dijkstra algorithms. For a full treatment of these algorithms and on how to create dynamic SPT algorithms based on static ones, please see [13].

The dynamic Dijkstra algorithm implemented in this simulation is basically the same as most dynamic algorithms that have been proposed in the literature [12], [7], [9], [8], [13]. It consists of propagating the disturbance caused by the edge weight change throughout the network one node at a time. The next node is selected based on the minimum distance criterion. On the other hand, our new algorithm speeds up the operation by spreading the disturbance one branch (containing usually more than one node) at a time, using the minimum distance change criterion. Because both of these two algorithms require searching for a minimum key, the simulation uses a standard binary heap to perform this operation. The dynamic Bellman-Ford and dynamic D'Esopo-Pape are dynamic generalizations of well-known Bellman-Ford and D'Esopo-Pape algorithms. A detailed specification of these algorithms can be found in [13]. The details of how the simulation was performed can be found in [14].

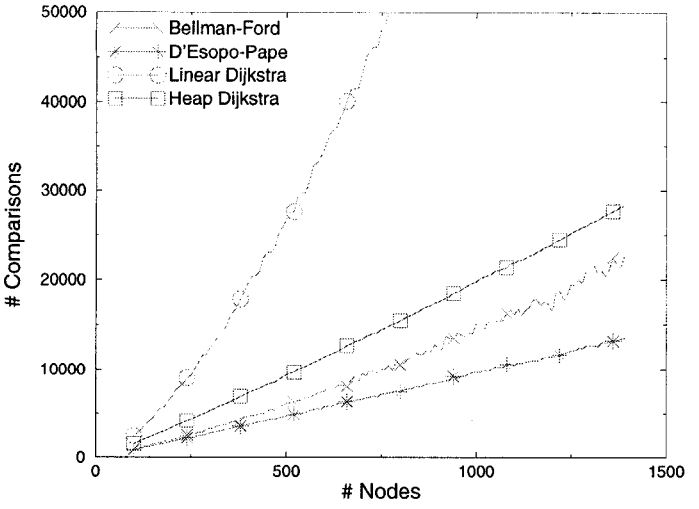

Fig. 4. Complexity of static algorithms.

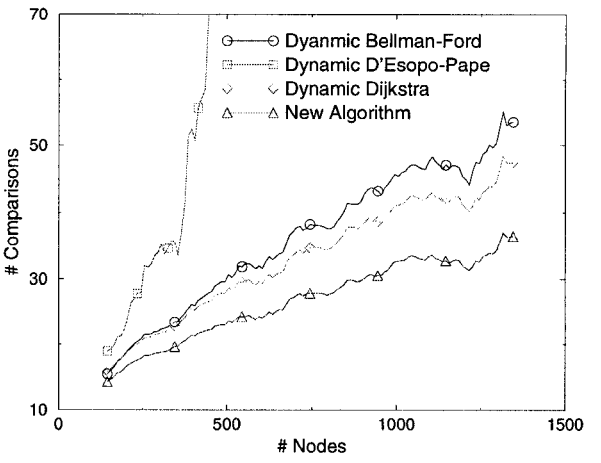

Fig. 5. Complexity of dynamic algorithms.

A. *Observed Complexity*

During the simulation, we compare the average observed complexity of the different dynamic SPT algorithms as the network size increases. For each network size, 40 different networks of that size are randomly generated ($L = 0.5$, $D = 7$). For each network, 175 link failures and 175 link recoveries were simulated. After each simulation the observed complexity (the number of comparisons performed) of that single-failure event is recorded. The average observed complexity over the 40×350 events of the same network size is then computed.

Figure 4 first illustrates the average observed complexity of conventional static SPT algorithms as the network size increases. These results are obtained from [13] where they are analyzed in detail. It is worth noticing the really large number of comparisons needed as the network increases its size. It is clear that using linear search for the priority queue in Dijkstra's algorithm quickly becomes inefficient. It is also interesting to note that the algorithm with the worst worst-case bound (D'Esopo-Pape) actually performs the best on average. This is because of some clever heuristic that it uses (revisited nodes are given higher priority - see [13]. However, this heuristic will prove to be counterproductive when using the dynamic version of these algorithms.

When we perform the same experiment using the various dynamic algorithms discussed, we notice a tremendous reduction in the algorithmic complexity. Figure 5 compares the average observed complexities, which as expected, are orders of magnitudes lower than for the static algorithms.

The dynamic version of the D'Esopo-Pape algorithm now becomes quite inefficient with large networks. Dynamic Dijkstra performs slightly better than Dynamic Bellman-Ford because of its improved node selection method. Our now algorithms clearly performs much better on average than the other three dynamic algorithms.

We now give an example of what happens inside each algorithm when some important link failure occurs. Table I shows the events that took place during the execution of each algorithm after such a failure took place.

The first column contains the number of edges visited by each algorithm. Recall that each time an edge is visited a comparison is made between the nodes at each end of the edge. Dynamic Bellman-Ford and D'Esopo-Pape end up visiting the same edges more than once. Dynamic D'Esopo-Pape seems to be particularly inefficient at this. On the other hand, both Dynamic Dijkstra and our new algorithm visit only the minimum number of edges necessary, and each edge is visited only once. This number (2627) is equivalent to what we previously defined as δ_e, the number of edges departing from the nodes of the non-root branches (nodes affected by the change).

The second column shows the number of extractions or iterations performed by each algorithm. This is the number of times that a new node has to be chosen from the list Q. The number of extractions is by far lower with the new algorithm. In fact, the new algorithm performs the minimum number of extractions. This number (13) is equivalent to δ_c, the number of non-root branches (clusters of nodes that need to be reordered).

The third column shows the number of comparisons performed by the search functions in each algorithm. Since Dynamic Bellman-Ford and D'Esopo-Pape have no priority queues, they don't have to perform any such comparisons. However, the extra cost suffered by Dynamic Dijkstra and our new algorithm is compensated by the smaller number of edges needed to be visited. Since our new algorithm needs to do less extractions and searches than Dynamic Dijkstra, less search comparisons are performed.

Finally, the last column shows the number of times that the parent attributes are changed during the execution of the algorithms. Dynamic Bellman-Ford and D'Esopo-Pape change the parent attributes of a large number of nodes more than once. Dynamic Dijkstra only changes once the parent attribute of any given node. However, it changes more parent attributes than necessary. The resulting SPT has 6 nodes unnecessarily changing parent. On the other hand, our new algorithm only makes the minimum number

	Edges	Extr.	Comps.	Par.
Dyn. Bellman-Ford	5582	646	0	347
Dyn. D'Esopo-Pape	30676	4880	0	2375
Dyn. Dijkstra	2627	186	1876	19
New Algorithm	2627	13	602	13

TABLE I

OBSERVED ALGORITHM BEHAVIOR AFTER A LINK FAILURE.

of changes in the SPT structure. There is only one edge change for each of the 13 non-root branches.

VIII. CONCLUSION

In this paper, we have presented a new algorithm to recompute the shortest path tree (SPT) in a network after some edges have changed weights. The algorithm is heavily based on the dual linear programming technique and on its physical interpretation. This physical interpretation uses balls and strings to model the system of constraints involved in computing a SPT. Our new algorithm can be seen as a simulation of how such a physical model would naturally reconfigure itself.

The new algorithm uses a new search criterion (that of minimum distance change) to determine in which direction the information should be propagated. Unlike existing dynamic SPT algorithms which select one node at a time, the new algorithm can select entire branches at a time. This allows the algorithm to use the intact structure of the SPT to spread information in the right direction without having to perform as many searches. The new algorithm also makes the minimum number of changes to the SPT structure.

The algorithm visits each affected node only once. It makes fewer extractions from the priority queue (one per branch) than any other algorithm of comparable complexity (one per node). As a result, when using underlying linear search or bucket search, the new algorithm has a lower asymptotic complexity than any other known algorithm. When using a more complicated ordered structure (such as a heap), the asymptotic complexity is the same as for the best known algorithms. Nevertheless, simulations indicate that on average, the new algorithm has the lowest observed complexity.

REFERENCES

[1] C. Baransel, W. Dobosiewicz, and P. Gburzynski, "Routing in multihop packet switching networks: Gb/s challenge," IEEE Network, vol. 9, pp. 38-61, May/June 1995.

[2] R. Bellman. "On a Routing Problem," Quarterly of Applied Mathematics, vol. 16, 1958, p. 87-90.

[3] D. Bertsekas. *Linear Network Optimization: Algorithms and Codes* The MIT Press, Cambridge, Massachusetts.

[4] L. Breslau and D. Estrin. "Design of inter-administrative domain routing protocols," Proceedings of SIGCOMM'90. Sept. 1990, pp. 231-241.

[5] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms* The MIT Press, Cambridge, Massachusetts.

[6] S. Deering and D. Cheriton. "Multicast routing in datagram internetworks and extended LANs," ACM Transactions on Computer Systems, vol. 8, no. 2, pp. 85-110, May 1990.

[7] P. Franciosa, D. Frigioni, and R. Giaccio. "Semi-Dynamic Shortest Paths and Breadth-First Search in Digraph," Proceedings of 14th Annual Symposium on Theoretical Aspects of Computer Science, March 1997, p. 33-46.

[8] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, "Fully Dynamic Output Bounded Single Source Shortest Path Problem," Technical Report.

[9] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. "Incremental Algorithms for Single-Source Shortest Path Trees," Proceedings of Foundations of Software Technology and Theoretical Computer Science, Dec. 1994, p. 113-124.

[10] C. Huitema. *Routing in the Internet.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

[11] G. Italiano, A. Marchetti-Spaccamela, U. Nanni, "Incremental Algorithms for Minimal Length Paths," Journal of Algorithms, vol. 12, 1991, p. 615-638.

[12] J. McQuillan, I. Richer, E. Rosen, "The New Routing Algorithm for the ARPANET," IEEE Transactions on Communications, vol. COM-28, no. 5, May 1980, p. 711-719.

[13] P. Narvaez, K.-Y. Siu, H.-Y. Tzeng, "New Dynamic Algorithms for Shortest Path Tree Computation," Technical Memorandum BL0113470-980505-04TM, Bell Labs, Lucent Technologies, May 5, 1998.

[14] P. Narvaez, K.-Y. Siu, H.-Y. Tzeng, "New Dynamic SPT Algorithm based on a Ball-and-String Model ," Technical Report, December 1998. http://list.mit.edu/ pnarvaez/publications.html

[15] R. Perlman and G. Varghese. "Pitfalls in the design of distributed routing algorithms," Proceedings of SIGCOMM'88. August 1988, pp. 43-54.

[16] R. Perlman. "A comparison between two routing protocols: OSPF and IS-IS," IEEE Network, Sept. 1991, vol. 5, pp. 18-24.

[17] G. Ramalingam and T. Reps. "An Incremental Algorithm for a Generalization of the Shortest-Path Problem," Journal of Algorithms, vol 21, 1996, p. 267-305.

[18] P. Spira and A. Pan. "On Finding and Updating Spanning Trees and Shortest Paths," SIAM Journal on Computing, vol. 4, no. 3, Sept. 1975, p.375-380.

[19] E. Feuerstein, A. Marchetti-Spaccamela "Dynamic Algorithms for Shortest Paths in Planar Graphs," Theoretical Computer Science, vol. 116, 1993, p. 359-371.

[20] E. Dijkstra "A note two problems in connection with graphs," Numerical Mathematics, vol. 1, 1959, p. 269-271.

[21] C. Labovitz, G. Malan, and F. Jahanian. "Internet Routing Instability," Proceedings of SIGCOMM'97. Sept. 1997, pp. 115-126.

[22] J. Moy. "OSPF Version 2," Internet Draft, rfc 2178, July 1997.

[23] V. Paxson. "End-to-end routing behavior in the Internet," IEEE/ACM Transactions on Networking, Oct. 1997, vol.5, (no.5):601-15.

[24] Y. Rekhter and T. P. Gross. "Applications of the Border Gateway Protocol in the Internet," RFC 1772, DDN Network Information Center, March 1995.

[25] M. Schwartz and T. Stern. "Routing techniques used in computer communication networks," IEEE Transactions on Communications, April 1980, vol.28, pp. 539-552

[26] M. Streenstrup, Ed. *Routing in Communications Networks*, Englewood Cliffs, NJ: Prentice-Hall, 1995.

[27] P. Traina, Ed., "BGP-4 protocol analysis," RFC 1774, DDN Network Information Center, March 1995.