New Dynamic Algorithms for Shortest Path Tree Computation

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

Abstract—**The Open Shortest Path First (OSPF) and IS-IS routing protocols widely used in today's Internet compute a shortest path tree (SPT) from each router to other routers in a routing area. Many existing commercial routers recompute an SPT from scratch following changes in the link states of the network. Such recomputation of an entire SPT is inefficient and may consume a considerable amount of CPU time. Moreover, as there may coexist multiple SPTs in a network with a set of given link states, recomputation from scratch causes frequent unnecessary changes in the topology of an existing SPT and may lead to routing instability. In this paper, we present new dynamic SPT algorithms that make use of the structure of the previously computed SPT. Besides efficiency, our algorithm design objective is to achieve routing stability by making minimum changes to the topology of an existing SPT (while maintaining shortest path property) when some link states in the network have changed.**

We establish an algorithmic framework that allows us to characterize a variety of dynamic SPT algorithms including dynamic versions of the well-known Dijkstra, Bellman–Ford, D'Esopo–Pape algorithms, and to establish proofs of correctness for these algorithms in a unified way. The theoretical asymptotic complexity of our new dynamic algorithms matches the best known results in the literature.

Index Terms—**Routing, shortest path trees.**

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 [22], [16]. 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 (e.g., an OSPF 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 area, 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.

Manuscript received September 24, 1998; revised August 26, 1999; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Ammar.

P. Narváez was with Bell Labs, Lucent Technologies, Holmdel, NJ USA. He is now with Sycamore Networks, Chelmsford, MA 02138 USA (e-mail: paolo.narvaez@sycamorenet.com).

K.-Y. Siu is with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: siu@list.mit.edu).

H.-Y. Tzeng is with Amber Networks, Santa Clara, CA 95054 USA (e-mail: htzeng@ambernetworks.com).

Publisher Item Identifier S 1063-6692(00)10928-8.

When the topology in a routing area changes (e.g., a link fails, recovers, or changes its routing cost), every router in the area is notified of the change. After updating the corresponding topology changes in its data structure, 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 [7].

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. Such recomputation can consume a considerable amount of CPU time, preventing other critical routing functions from being executed. Thus, it is desirable to compute the SPT using as little CPU time as possible.

To cope with the complexity of SPT computation, the traditional approach has been simply to limit the size of the routing area (e.g., an OSPF area). In fact, it has been recommended that the size of an OSPF area be limited to 200 routers, based on the cost of the SPT computation [17]. Today, most router vendors limit the OSPF area to a size between 50 and 500 routers. However, there is a growing interest among network operators to increase the size of a routing area in order to increase redundancy and optimize routing. By reducing the complexity of the SPT computation and thus eliminating this performance bottleneck, a larger routing area can be allowed.

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 minimum 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 the 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 [15], [20].)

In this paper, we will explore a rich class of algorithms that can dynamically update the SPT following changes in the link states. These dynamic algorithms use information of the outdated SPT and update only the part of the SPT that is affected by the change. Our design objectives for these dynamic algorithms 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.

The purpose of our work is restricted to dynamic SPT algorithms that can be used in link-state protocols. Our algorithms need to be executed in a centralized processor and require a complete topology database (such as the link-state database in OSPF). There are other routing algorithms in the literature [12], [23] which use tree-like structures. However, these algorithms are distributive and operate under very different assumptions. Comparisons and analogies between such distributed routing algorithms and algorithms where all path calculation is centralized (like ours) are beyond the scope of this paper.

Unlike previous work on dynamic SPT algorithms that is based on the static Dijkstra algorithm only, we shall present an algorithmic framework that also yields dynamic versions of other well-known static SPT algorithms such as Bellman–Ford and D'Esopo–Pape. This framework allows us to characterize dynamic SPT algorithms in a unified way and to establish proofs of correctness for these algorithms. In particular, within our framework, we propose two different incremental methods to transform static algorithms into new dynamic algorithms. The first method when applied to the Dijkstra algorithm yields a dynamic algorithm similar to existing ones [11], [10], [14]. The first method can also be used to transform other static algorithms (i.e., Bellman–Ford) into their dynamic versions. The second incremental method yields new dynamic algorithms that are faster on average. Furthermore, with the second incremental method, the resulting dynamic algorithm will make the minimum number of changes to the SPT topology following a link-state update, thus improving routing stability.

In the next section, we shall further discuss some prior works. Section III introduces graph-theoretic definitions and notations to be used in the paper. In Section IV, we describe our algorithmic framework by way of a basic algorithm for computing an SPT. Section V explains how the well-known static SPT algorithms can be characterized in our algorithmic framework. Sections VI and VII describe two specific methods for implementing the basic algorithm, which convert static algorithms into dynamic algorithms. Section VIII discusses theoretical bounds on the asymptotic computational complexity of each specific dynamic algorithm. Concluding remarks are given in Section IX. The proofs of the theoretical results are presented in the Appendix.

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], [12], [13], [21], [23], [26], [28] and many routing protocols have been studied and used in practical networks [6], [18], [22], [25], [30]. The stability issues in Internet routing have attracted much attention [15], [20]. In fact, our interest in dynamic SPT algorithms was partly motivated by the problem of routing instability in the Internet.

To the best of our knowledge, the earliest work on dynamic SPT algorithms that appears in the literature is [27]. The work proves a lower-bound complexity for the worst-case scenario. The first efficient dynamic SPT algorithm that we know of is discussed in [18]. This algorithm is very similar to the First Incremental Dijkstra algorithm presented in our paper, though the work [18] contains no analytical proofs nor simulation results.

Two semidynamic algorithms to update SPTs are presented in Franciosa *et al.* [9]. Each semidynamic 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_d in this paper). Therefore, the complexity cannot be shown to be better than that of a static algorithm.

Frigioni *et al.* [11], [10] present an algorithm similar to the one in [9] but it allows the edge weights to be nonintegers. The algorithm is also a dynamic version of the Dijkstra algorithm. It is quite similar to our First Incremental Dijkstra algorithm; the main difference is that our algorithm optimizes the initialization procedure. The complexity of the algorithm in [10] is analyzed in terms of the number of nodes whose distance attributes 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.

III. Definitions and Notations

We now define some notation to be used in the rest of the paper. We assume that the reader is familiar with basic graph theoretic definitions. 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 *Source* $(\mathcal{G}) \in \mathcal{V}$ denote the root or source node of $\mathcal{G}$.

For each directed edge $e \in \mathcal{E}$, we use $\mathtt{W}(e)$ to denote the weight (distance) associated with e, $\mathtt{S}(e)$ and $\mathtt{E}(e)$ 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: $\mathtt{I}(\mathcal{N}) = \{\mathtt{e} \in \mathcal{E} \mid \mathtt{E(e)} \in \mathcal{N}\}$ (the set of edges directed *into* the nodes in $\mathcal{N}$) and $\mathtt{O}(\mathcal{N}) = \{\mathtt{e} \in \mathcal{E} \mid \mathtt{S(e)} \in \mathcal{N}\}$ (the set of edges directed *out* of the nodes in $\mathcal{N}$). These parameters only depend on the topology of the network, and their values do not change during the execution of the algorithm.

A rooted tree $\mathcal{T}$ is a subgraph of $\mathcal{G}$ such that *Source* $(\mathcal{G})$ is in $\mathcal{T}$ and every node in $\mathcal{T}$ is reachable from $Source(\mathcal{G})$ through a unique directed path using only edges in $\mathcal{T}$. A node x is a parent node of y in $\mathcal{T}$ if x is the source node and y is the end node of an edge in $\mathcal{T}$. We associate with each node n in a tree $\mathcal{T}$ the following attributes: $\mathtt{P}(\mathtt{n}, \mathcal{T})$ is the parent node of n and $\mathtt{D}(\mathtt{n}, \mathcal{T})$ is the distance attribute of n. Since $\mathcal{T}$ is a tree, invoking $\mathtt{P}(\mathtt{n}, \mathcal{T})$ recursively determines a unique path from *Source* $(\mathcal{G})$ to any node in $\mathcal{T}$.

The descendents of a node n in $\mathcal{T}$ are all the nodes that are reachable by n. We use $\mathtt{B}(\mathtt{n}, \mathcal{T})$ to denote a subset that includes n and some descendents of n in tree $\mathcal{T}$. As we will see later, an execution of our algorithm involves the distance update of the subset $\mathtt{B}(\mathtt{n}, \mathcal{T})$ for various nodes n. The different ways the subset $\mathtt{B}(\mathtt{n}, \mathcal{T})$ can be chosen for each node n will lead to different incremental methods for computing the SPT. For ex-

ample, $\mathrm{B}(\mathrm{n}, \mathcal{T})$ can be chosen to be simply n itself, the children (the next generation) of n with itself, or all its descendents with itself in tree $\mathcal{T}$. Specifically, the notation $\mathrm{B}_{\max}(\mathrm{E}(e), \hat{\mathcal{T}})$, which is always used in the initialization of the algorithm denotes the set including the end node of edge e and all its descendents in $\hat{\mathcal{T}}$.

An SPT in $\mathcal{G}$ is a rooted tree such that the length of every path from *Source* $(\mathcal{G})$ to every node in $\mathcal{G}$ is minimized. Since there can be different paths of the same minimum length between two nodes, an SPT in $\mathcal{G}$ is not unique. However, it follows from the definition that the length of the path from *Source* $(\mathcal{G})$ to any node $n \in \mathcal{G}$ in any SPT is unique.

IV. Algorithmic Framework

In this section, we present a broad class of algorithms that can compute an SPT from a source node to every other node in the graph. This class of algorithms is presented in a unified framework as the *basic algorithm*. Depending on the initialization procedure (see below), the basic algorithm includes as special cases some well-known static SPT algorithms, such as Bellman–Ford, Dijkstra, and D'Esopo–Pape as well as their dynamic versions. (The proof of correctness of the basic algorithm in computing an SPT will be presented later, in the Appendix.)

A tree data structure, denoted by $\hat{\mathcal{T}}$, is maintained by the basic algorithm to keep track of an existing potential SPT. In particular, every node n in the graph $\mathcal{G}$, along with its parent attribute $P(n, \hat{\mathcal{T}})$ and distance attribute $D(n, \hat{\mathcal{T}})$ 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.

In addition to the data structure $\hat{\mathcal{T}}$, the basic algorithm also maintains a list $\mathcal{Q}$ that contains a subset of nodes and their parent and distance attributes temporarily. In particular, each element in $\mathcal{Q}$ is of the form $\{n, (p, d)\}$, where p and d respectively denote the parent of n and the potential distance of n from the source node with respect to some tree. The instruction ENQUEUE$(\mathcal{Q}, \{n, (p, d)\})$ adds one more element to $\mathcal{Q}$; if node n is already in $\mathcal{Q}$ with attributes p_{old} and d_{old}, the new attributes (p, d) will replace the old ones only if the new distance d is less than d_{old}. At any instant, only one set of attributes is maintained for each node in $\mathcal{Q}$. When an EXTRACT$(\mathcal{Q})$ instruction is executed, a single element (to be defined by a specific algorithm) is selected and removed from $\mathcal{Q}$. Different implementations of ENQUEUE$(\mathcal{Q}, \{n, (p, d)\})$ and EXTRACT$(\mathcal{Q})$ in the basic algorithm yield different specific algorithms to be later discussed.

The basic algorithm also makes use of arithmetic using the symbol for infinity (∞). By this term, we refer to any number that is much larger than the distance of the longest path in the network. It is also the largest number we can have. On most computers, this term would be the largest number that can be held by a variable type. For example if we perform the arithmetic operation $\infty - x - \infty$, the result would be $-x$. On the other hand, since no number can be larger than ∞, $\infty + x - \infty = 0$.

The basic algorithm contains an initialization procedure (step 1) and an iterative loop (steps 2–4). The following pseudocode formally describes the algorithm. An informal description of the algorithm will follow after the pseudocode.

```
The Basic Algorithm
STEP 1: Initialization
,
(A) Static Version
∀(n ∈ V)
  P(n, T̂) ← ∅
  D(n, T̂) ← ∞
ENQUEUE (Q, {Source (G), (∅, 0)})

(B) Dynamic Version (an outdated SPT ex-
  ists)
Case 1 (Edge e increases its weight by
  Δ):
W(e) ← W(e) + Δ
if e is in T̂
  ∀n ∈ N = B_max(E(e), T̂)
    D(n, T̂) ← D(n, T̂) + Δ        /* Increase
  distance*/
/* of each descendent of E(e) and itself by
  Δ*/
  ∀e' ∈ I(N)      ./* Consider all edges di-
  rected into /*
/* all descendents of E(e) and itself*/
    if D(E(e'), T̂) > D(S(e'), γT̂) + W(e')
/* if the increased distance of a node
  is*/
/* larger than its distance in another
  path */
      newdist = D(S(e'), T̂) + W(e')  /* choose */
/* the smaller distance as the new poten-
  tial distance */
      ENQUEUE(Q, {E(e'), (S(e'), newdist)})
/* update the node with the new distance
  and */
/* new parent in the list Q*/

Case 2 (Edge e decreases its weight by
  Δ):
W(e) ← W(e) − Δ
if D(S(e), T̂) + W(e) < D(E(e), T̂)  /* if the */
/* reduced weight of e yields a smaller
  distance for E(e) */
  Δ' ← [D(S(e), T̂) + W(e)] − D(E(e), T̂)
/* Keep track of the difference */
  P(E(e), T̂) ← S(e)    /* set the parent at-
  tribute */
/* of the end node */
  ∀n ∈ N = B_max(E(e), T̂)
    D(n, T̂) ← D(n, T̂) + Δ'  /* Reduce the */
/* distance of each descendent of E(e) and
  itself */
/* by the difference */
  ∀e' ∈ O(N)  /* Consider all edges directed
  out */
/* of the descendents of E(e) and itself */
```

```
    if D(E(e'), T̂) > D(S(e'), T̂) + W(e')   /* if the
  */
/* reduced distance of a node also reduces
  the distance */
/* of another node */
    newdist ← D(S(e'), T̂) + W(e')  /* choose */
/* the smaller distance as the new poten-
  tial distance */
    ENQUEUE (Q, {E(e'), (S(e'), newdist)})
/* update the node with the new distance
  and new */
/* parent in the list Q */

STEP 2: Node Selection
if Q = ∅
  Terminate
else
  {y, (x, d)} ← EXTRACT (Q)  /* select and */
/* remove an element from the list */
  Δ ← (d − D(y, T̂))  /* compute the difference
  */
/* between new and old distances of the
  selected node */
  if Δ ≥ 0 /* if there is no improvement in
  distance */
    Go to Step 2   /* discard this informa-
  tion */
/* and go back to step 2 to extract an-
  other node */
  P(y, T̂) ← x     /* otherwise, update the
  parent */
/* attribute of the selected node */
  N ← B(y, T̂)  /* consider the selected node
  and */
a selected subset of its descendents */

STEP 3: Distance Update
∀n ∈ N
  D(n, T̂) ← D(n, T̂) + Δ /* update the distance
  of */
/* the selected node and each descendent
  considered */

STEP 4: Node Search
∀e ∈ O(N)   /* consider all edges directed
  out of the */
/* selected subset of nodes */
    if D(E(e), T̂) > D(S(e), T̂) + W(e)  /* if the */
/* reduced distance of a selected node
  also reduces */
/* the distance of another node */
      newdist ← D(S(e), T̂) + W(e) /* choose the */
/* smaller distance as the new potential
  distance */
    ENQUEUE (Q, {E(e), (S(e), newdist})
/* update the node with the new distance
  */
/* and new parent in the list Q */

Go to Step 2
```

A. Informal Discussion of the Algorithm

To help understand the basic algorithm, here we shall give a brief description of its executions. As the execution is the same for the static versions and the dynamic versions after initialization, we shall only discuss the dynamic version, since the initialization of the static version is trivial.

First, the goal of the initialization phase is to identify those nodes that may be affected by the change in the link state. 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.

In case 1, after updating the changed edge e 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) and all descendent nodes of E(e) that are reachable in the existing SPT. All such nodes will be included in a set $\mathcal{N}$ for further updating. The intuition is that the set $\mathcal{N}$ covers all the affected nodes. Now for each node that is attached to a link directed into a node in $\mathcal{N}$, we compute the potential distance *newdist* by adding the distance of the parent node and the weight of the edge connecting the two nodes.

In case 2, after updating edge e with the new decreased weight, we update the distance of all its descendents (the set $\mathcal{N}$) using the same difference change δ'. For every edge e originating from nodes in $\mathcal{N}$ we compute the potential new distance *newdist* of its end node E(e) by adding the distance of its source node S(e) and the new weight W(e). If this potential new distance is in fact smaller than the old distance of E(e), the node E(e) is inserted in the list $\mathcal{Q}$.

After the initialization step, we extract from the list $\mathcal{Q}$ in step 2 any element (the exact procedure is determined by the specific algorithm). This selected node is then updated (if the new distance is better) with its new parent indicated in $\mathcal{Q}$ and the structure of $\hat{T}$ is modified accordingly to reflect the new child-parent relation. Moreover, in step 3, the selected node together with any (to be specified by the method used) of its descendent nodes (denoted by the set $\mathcal{N}$) in the existing tree $\hat{T}$ will now have their correct distances updated as their old distances incremented by Δ.

Finally, in step 4, we consider each node E(e) that is attached to an edge e directed out of a node S(e) in $\mathcal{N}$. If the potential new distance *newdist* of E(e) [which is equal to W(e) plus the distance of S(e)] is smaller than its old distance, we then enqueue in the list $\mathcal{Q}$ the node E(e). After this step is completed, the execution of steps 2 and 3 will be repeated until the list $\mathcal{Q}$ is empty.

At each iteration, a new node and possibly some of its descendents are updated with a better distance and a better parent that reflects that distance. In this manner a change in a link-state will propagate along the old SPT until a new valid SPT is constructed.

B. Multiple Link Weight Changes

When there are several link weight changes occurring at once, it is always possible to run the algorithm sequentially for each weight changed. However, some optimization can be achieved

by updating several of these changes together. Even though a *separate* initialization needs to be done for every weight change, the main body of the algorithm needs to be executed only twice. We now describe how such optimization can be realized.

First of all, we initialize all the weight increases. For each link whose weight increases, the first four lines of the code for Case 1 are executed. This ensures that all the affected nodes have updated their distance attributes in $\hat{T}$. Then, we execute the next four lines of the code in Case 1 for a set $\mathcal{N}$ that includes all the nodes previously updated. This ensures that all the necessary nodes are enqueued in the list $\mathcal{Q}$. The rest of the algorithm (steps 2–4) is then executed in a normal way.

After the SPT is updated with all the weight increments, we update it again for all the weight decrements. As before, the first six lines of the initialization (Case 2) are executed for each weight decrement, and then the last four lines are executed with a set $\mathcal{N}$ which includes all the nodes previously updated. After the initialization, the rest of the algorithm (steps 2–4) is executed normally.

V. Special Cases of the Basic Algorithm

Here we will show how the best-known static algorithms can be viewed as special cases of our basic algorithm. For these cases, the static version of the initialization procedure in our basic algorithm is used. The branching function $\mathrm{B}(\mathrm{n}, \mathcal{T})$ in the static case is defined as

$$\mathrm{B}(\mathrm{n}, \mathcal{T}) = \{n\} \qquad \forall \mathrm{n} \in \mathcal{V}\,. \tag{1}$$

In this way, the new SPT is constructed in $\hat{T}$ by changing the attributes of one node at a time during each iteration. This method is used in several well-known static algorithms for computing the SPT. The only differences between these algorithms consist in the different ways in which the temporary list $\mathcal{Q}$ is implemented.

A. Bellman–Ford Algorithm

The simplest way of implementing the list $\mathcal{Q}$ is by using a FIFO queue. New nodes and their attributes are enqueued at bottom of the queue and extracted from the top of the queue. When this queueing discipline is used in the static method, it gives rise to the Bellman–Ford algorithm [2]. We will refer to it as the static Bellman–Ford algorithm.

B. D'Esopo–Pape Algorithm

Another way of implementing $\mathcal{Q}$ is by using a queue where (as in Bellman–Ford's case) nodes are extracted from the top of the queue. However, unlike Bellman–Ford's case, a node is enqueued at the bottom of the queue only when that node has not been enqueued in the list $\mathcal{Q}$ before. If some node n has already been inserted and extracted from $\mathcal{Q}$, the next time when n is inserted in $\mathcal{Q}$ it will be at the top. When the static version of the initialization procedure is used, this queueing discipline gives rise to the D'Esopo–Pape algorithm [3]. We will refer to it as the static D'Esopo–Pape algorithm.

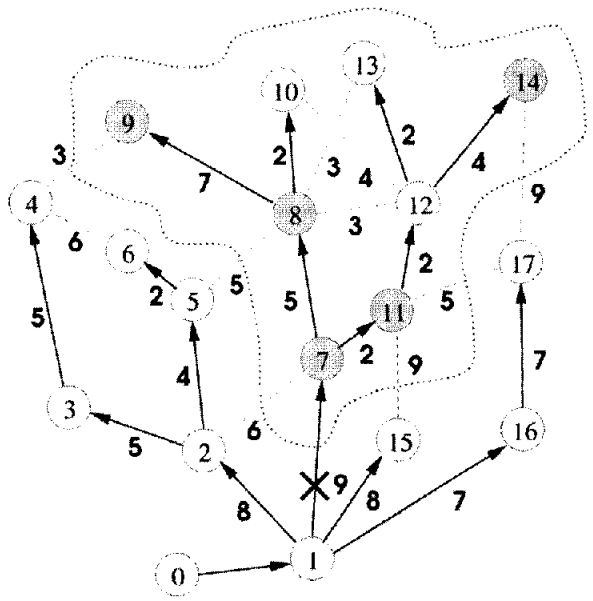

Fig. 1. Initialization after a link failure.

C. Dijkstra Algorithm

Yet another way of implementing $\mathcal{Q}$ is by using a priority queue where the node enqueued with the smallest distance attribute is always extracted first. When the static version of the initialization procedure is used, this algorithm is equivalent to the Dijkstra algorithm [7].

To implement the priority queue, one can use a linked list where inserted nodes can be placed anywhere. The node with the smallest distance attribute can then be extracted by searching through the linked list sequentially. The resulting algorithm will be referred to as the static Linear Dijkstra algorithm. A more efficient way to implement the priority queue is to use a binary heap data structure, and the resulting algorithm in this case will be called the static Heap Dijkstra algorithm.

VI. First Incremental Method

The four static SPT algorithms presented in the preceding section can be transformed into dynamic algorithms by using the dynamic version of the initialization procedure of the basic algorithm. The branching function $\mathrm{B}(\mathrm{n}, \mathcal{T}) = \{\mathrm{n}\}$ is the same as in the static algorithms.

We call this method the first incremental method and the resulting algorithms "First Incremental Bellman–Ford," "First Incremental D'Esopo–Pape," "First Incremental Linear Dijkstra," and "First Incremental Heap Dijkstra."

We will illustrate with a number of figures how the first incremental algorithms work on a simple network. For simplicity, we assume each link in the network is bidirectional and has the same cost (indicated in bold-face numbers in Fig. 1) in each direction. In these figures, the solid thick arrows between nodes represent the directed edges that are in the current SPT ($\hat{T}$). The thin dashed segments represent other edges in the network that are not in $\hat{T}$.

Fig. 1 shows how the algorithm is initialized after a link failure between nodes 1 and 7 [link (1,7)]. First link (1,7) is removed from $\mathcal{G}$. The dotted curve includes the set of nodes $\mathcal{P}$

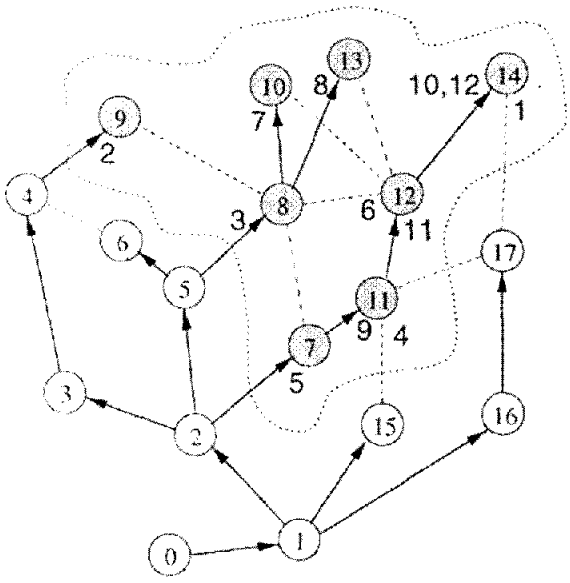

Fig. 2. New SPT using First Incremental Bellman–Ford.

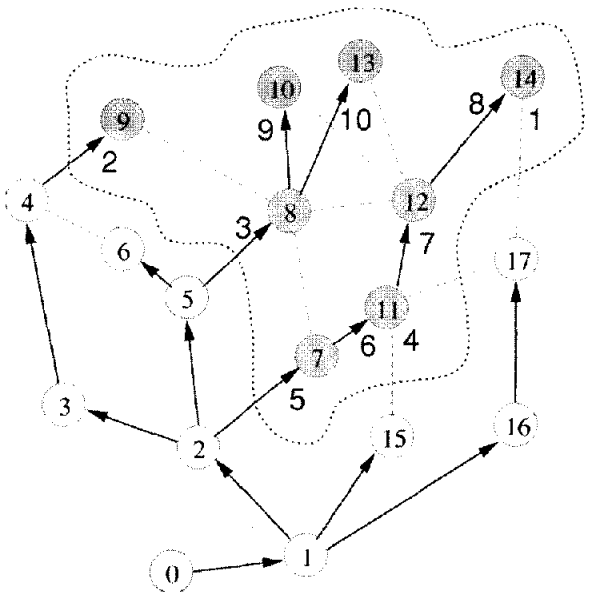

Fig. 3. New SPT using First Incremental D'Esopo–Pape.

that are descendents of node 7. These are the nodes that might have their distance attributes increased as a result of the link failure. The nodes outside $\mathcal{P}$ will be unaffected by the link failure. The shaded nodes denote the nodes in the *germinal set* (to be defined later in the Appendix when the theoretical results are proved) that are initially stored in $\mathcal{Q}$. These nodes are necessarily the point of entry into $\mathcal{P}$ for all potential new shortest paths. This initialization procedure is the same for all the dynamic algorithms.

Fig. 2 illustrates how the First Incremental Bellman–Ford algorithm recomputes the SPT. We assume the link distance is the same as in Fig. 1 (except for the failed link) and for simplicity, the link distance is not indicated in Fig. 2 and subsequent figures referring to the same network. The number next to each node represents the order in which that node is extracted from $\mathcal{Q}$. Each number is also placed next to the link indicating the parent attribute with which it was extracted. Note that in Bellman–Ford's algorithm, the initial order of extraction from $\mathcal{Q}$ is arbitrary. Our example indicates that node 14 is the first node extracted from $\mathcal{Q}$, and its parent attribute is node 17. The next node extracted is node 9 with parent attribute node 4.

Note that with this algorithm, nodes 11, 12, and 14 are extracted more than once. Since a node might be extracted with incorrect attributes, it will need to be enqueued and extracted again to obtain correct attributes. Also, note that the parent attribute of node 13 has changed from node 12 to node 8 in a new SPT. This change still gives a correct (but different) SPT. The changes that were performed on the tree data structure were unnecessary. This algorithm does not alter the SPT structure of the nodes whose distance is not affected by the change. However, when there are several possible SPTs, this algorithm cannot guarantee that the minimum number of changes will be made in the tree structure. (In fact, the unnecessary routing change in this example motivates us to consider more stable dynamic algorithms using the second incremental method to be discussed in the next section.)

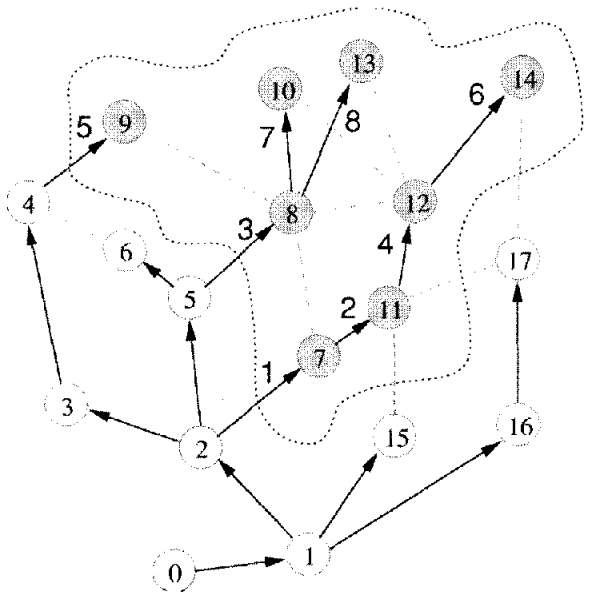

Fig. 4. New SPT using First Incremental Dijkstra.

Fig. 3 shows how the First Incremental D'Esopo–Pape obtains a new SPT for the same problem. Some nodes are still extracted more than once, but the total number of extractions is smaller than in the Bellman–Ford algorithm (10 extractions instead of 12). In this example, the improvement is caused by the heuristics of D'Esopo–Pape. When node 11 is enqueued for a second time in $\mathcal{Q}$, it receives first priority in $\mathcal{Q}$ so that it is extracted immediately after with correct attributes. Because node 11 is extracted earlier (extraction #6 rather than #9), it allows for nodes 12 and 14 to have correct attributes the very first time they are extracted.

Fig. 4 shows how the First Incremental Dijkstra algorithm obtains a new SPT for the same problem. Because of the order in which nodes are extracted (the node with the smallest distance attribute first), every node is extracted only once. Therefore, this

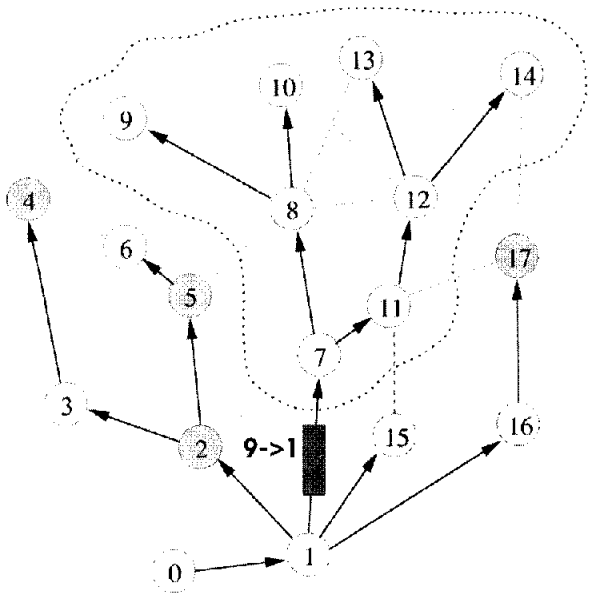

Fig. 5. Initialization after a link cost improvement.

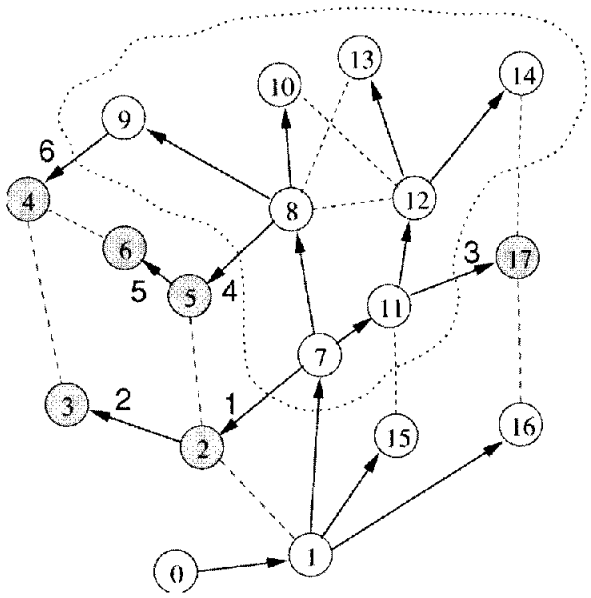

Fig. 6. New SPT using First Incremental Dijkstra.

algorithm performs the minimum number of extractions. The cost incurred in such an improvement is that some data structure is needed in order to search for the node with the smallest distance attribute in $\mathcal{Q}$.

On the other hand, Fig. 5 shows how the basic algorithm is initialized when edge (1,7) decreases its cost by some $\Delta = 8$. The set of nodes inside the dotted curve (set $\mathcal{P}$) will have their distance attributes decreased by Δ, but the SPT structure inside $\mathcal{P}$ will remain unchanged. Only the nodes outside of $\mathcal{P}$ might have to change parent attributes in order to maintain shortest distances from the source node. In this case, the shaded nodes can decrease their distance attributes by changing their parent attribute to some node in $\mathcal{P}$ (indicated by the thin arrow). Note that all SPT paths from $\mathcal{P}$ to nodes outside $\mathcal{P}$ must necessarily pass through the shaded nodes (in the *germinal* set). All nodes in the germinal set are inserted in the list $\mathcal{Q}$. This initialization procedure is the same for all our dynamic algorithms.

Fig. 6 shows how the First Incremental Dijkstra algorithm reconstructs an SPT after the initialization in Fig. 5. Each number next to a node represents the order and the parent attribute of the extracted node. This algorithm only extracts each node once from $\mathcal{Q}$. The shaded nodes have all been visited by the algorithm and have had their distance attributes decreased. The nodes outside of $\mathcal{P}$ that are not shaded are the nodes that could not decrease their distance attributes, and therefore, they are not affected by the algorithm.

VII. Second Incremental Method

Another set of dynamic algorithms can be obtained by changing the branching function $\mathrm{B}(y, \mathcal{T})$ used in Step 2 of the basic algorithm so that it includes as many descendents as possible. For optimization purposes, the only nodes that should not be included in $\mathrm{B}(y, \mathcal{T})$ are those that will be directly affected by some node extraction from $\mathcal{Q}$ (there is no point in updating a node that we know will be later updated with a smaller distance once it is extracted from $\mathcal{Q}$). Likewise, every node included in $\mathrm{B}(n, \mathcal{T})$ should be removed from $\mathcal{Q}$ (there is no point in keeping a node with a known nonoptimal distance in $\mathcal{Q}$). To specify exactly how the selection of the nodes is done, we present the following algorithm that defines the set $\mathcal{N} = \mathrm{B}(y, \hat{\mathcal{T}})$ in Step 2 of the basic algorithm. The instruction DEQUEUE$(n, \mathcal{Q})$ removes node n and its attributes from $\mathcal{Q}$. The notation $\mathrm{B}_1(k, \hat{\mathcal{T}})$ denotes the set including node k and its children in $\hat{\mathcal{T}}$.

```
K ← {y}  /* the temporary queue starts with
  node y */
N ← ∅     /* the branch set is initially
  empty */

step i)
if K = ∅     /* if the temporary queue is
  empty */
  Stop      /* done. N contains the desired
  output */
else
  k ← EXTRACT(K)  /* pick any node from the
  */
/* temporary queue */
  N ← {N, k} /* add this node to the output
  set */

step ii)
∀n ∈ B1(k, T̂)  /*for all the immediate chil-
  dren of k */
  if n ∈ Q /* if the child is in the candi-
  date list Q*/
    if D(n, T̂) + Δ ≤ D(n, Q)  /* and if the new
  */
/* flooded distance for the child is
  better */
      DEQUEUE(n, Q) /* remove it from */
/* the candidate list */
```

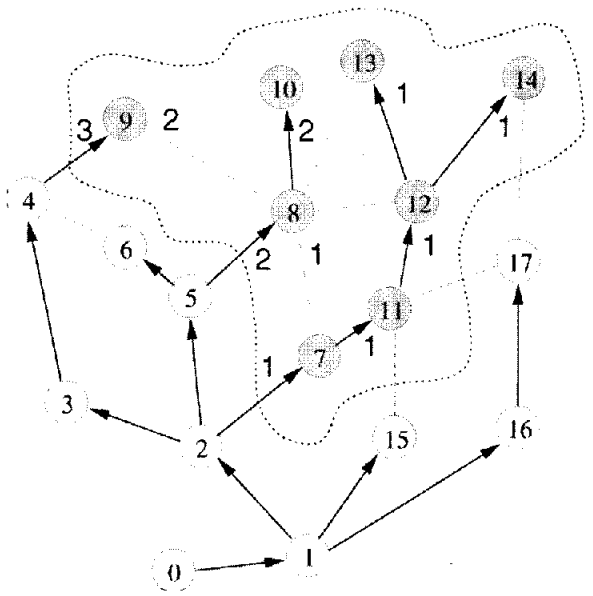

Fig. 7. New SPT using Second Incremental Dijkstra (after a link failure).

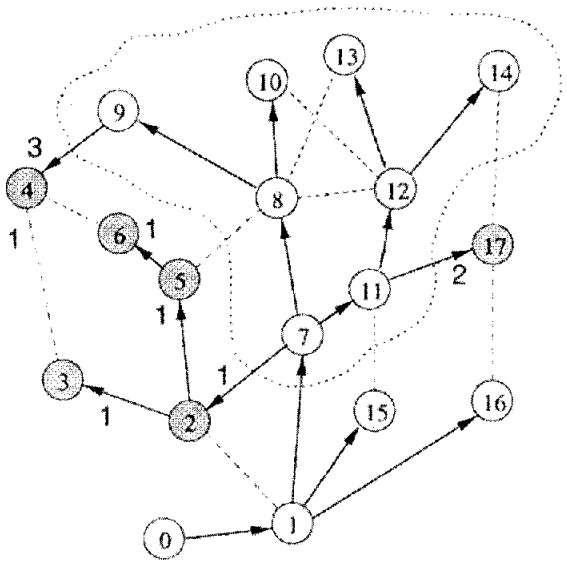

Fig. 8. New SPT using Second Incremental Dijkstra (after a link cost improvement).

```
    K ← {K, n} /* add the child node n */
/* to the temporary queue */
  else
    K ← {K, n} /* add the child node n */
/* to the temporary queue */
Go to step i)
```

We call this method the second incremental method and its four algorithms are referred to as "Second Incremental Bellman–Ford," "Second Incremental D'Esopo–Pape," "Second Incremental Linear Dijkstra," and "Second Incremental Heap Dijkstra."

The first incremental method changes the attributes of only one node during each iteration. The second incremental method is less conservative since at each iteration an entire branch of $\hat{T}$ changes its attributes. Simulation results also show that the extra work performed by the second incremental method pays off in terms of average performance. See [19] for an extensive simulation study of these algorithms.

For example, Fig. 7 shows how the Second Incremental Dijkstra algorithm obtains a new SPT after the initialization in Fig. 1. First, node 7 is extracted from $\mathcal{Q}$ with node 2 as its parent attribute. All the descendents of node 7 have their distance attributes updated. Then node 8 is extracted with node 5 as its parent attribute; all its descendents have their distance attributes updated. Finally, node 9 is extracted with node 4 as its parent attribute. Each node is extracted at most once from $\mathcal{Q}$ (there are only three extractions in this case). Nevertheless, because of the distance updates, some nodes (8 and 9) are visited more than once by the algorithm. The advantage of this second incremental method over the first one (shown in Fig. 4) is that even though some nodes are visited more than once, many fewer iterations take place, the list $\mathcal{Q}$ is much smaller and simpler, fewer search operations are needed, and most visited nodes only require a simple distance attribute update.

Another advantage of the second incremental method is that it guarantees that the minimum number of parent attributes will be changed. In other words, a link is removed from the current SPT tree $\hat{T}$ only if that link cannot be part of *any* possible SPT. This result is proved in Theorem 5 in the Appendix. In this example, node 13 maintains node 12 as its parent attribute and does not change it to node 8 as done in the first incremental method.

Fig. 8 shows the new SPT calculated by the Second Incremental Dijkstra algorithm after the initialization stage of the basic algorithm (see Fig. 5). As seen from the figure, a minimum number of parent attribute changes are made.

VIII. Algorithmic Complexity

Here we will state theoretical bounds for the complexity of the dynamic algorithms presented in the preceding sections (four dynamic algorithms corresponding to each incremental method). The proofs of these theoretical bounds will be given in the Appendix.

The complexity of algorithms is measured in terms of the following:

- total number of comparisons made between the distances of two nodes;
- total number of comparisons made between two elements in the list $\mathcal{Q}$;
- total number of times the nodes are extracted or dequeued from the list $\mathcal{Q}$.

Note that in the worst case, the updated SPT can be completely different from the outdated SPT, and the complexity of dynamic SPT algorithms is no better (or sometimes even worse) than the static algorithms. However, as we discussed earlier, it is most often the case that a change in the weight of an edge will only yield small changes in the topology of the SPT. Thus, it is more useful to measure the complexity of dynamic SPT algorithms in terms of parameters that reflect the changes in the SPT.

Given that there is a change in the weight of some edge, let δ_d denote the minimum number of nodes that must change their distance *or* parent attributes (or both) and δ_{pd} the minimum number of nodes that must change their distance *and* parent attributes. Note that the parameters δ_{pd} and δ_d depend only on the topology of the network, the current SPT, and the link-state change, but not on the algorithm used. They reflect the minimum amount of information needed by any dynamic algorithm in order to recompute the SPT. Also, let $D_{\max}$ denote the maximum node degree, which corresponds to the maximum number of ports any router or switch in the network can have. We will evaluate the complexity of our algorithms in terms of these parameters. We shall focus on the case where there is a change in the weight of only a single link.

For comparison with the well-known static algorithms, we state the following known results regarding their complexity [5]:

Bellman–Ford	$O(\lvert\mathcal{E}\rvert \cdot \lvert\mathcal{V}\rvert)$
D'Esopo–Pape	*No Polynomial Upper Bound*
Linear Dijkstra	$O(\lvert V\rvert^2)$
Heap Dijkstra	$O(\lvert\mathcal{E}\rvert \cdot \lg\lvert\mathcal{V}\rvert)$
Fibonacci Dijkstra	$O(\lvert\mathcal{V}\rvert \cdot \lg\lvert\mathcal{V}\rvert + \lvert\mathcal{E}\rvert).$ (2)

The Fibonacci Dijkstra algorithm uses the data structure of a Fibonacci heap to implement the priority queue for $\mathcal{Q}$. Although the algorithm has the lowest asymptotic complexity among others shown above, it is mostly of theoretical interest because of the large constant overhead involved in operating Fibonacci heaps [5].

A. First Incremental Method

The following summarizes the complexity of the dynamic SPT algorithms derived using the first incremental method:

Bellman–Ford	$O(D_{\max} \cdot \delta_d^2)$
D'Esopo–Pape	*No Polynomial Upper Bound*
Linear Dijkstra	$O(\delta_d^2 + D_{\max} \cdot \delta_d)$
Heap Dijkstra	$O(D_{\max} \cdot \delta_d \cdot \lg\delta_d)$
Fibonacci Dijkstra	$O(\delta_d \cdot \lg\delta_d + D_{\max} \cdot \delta_d).$ (3)

The complexity of the initialization is just the number of edges visited during step 1 of the basic algorithm because this stage does not involve queueing or searching operations. In Case 2 of the initialization procedure (when an edge decreases its weight), $\lvert \mathtt{B}_{\max}(\mathtt{E}(e))\rvert \leq \delta_d$. Therefore, the initialization complexity is always bounded by $O(D_{\max} \cdot \delta_d)$, which is always less than the complexity of the rest of the algorithm.

However, in Case 1 of the initialization procedure (when an edge increases its weight), $\lvert \mathtt{B}_{\max}(\mathtt{E(e)})\rvert \geq \delta_d$. Therefore, in rare cases, the initialization complexity might be higher than that of steps 2–4. If we want to obtain a strict linear complexity bound, we need to slightly modify the initialization procedure so that only the nodes that must change distance attributes are included in the set $\mathcal{N}$. The initialization complexity after such a modification is then bounded by $O(D_{\max} \cdot \delta_d)$, which is always less than the complexity of the rest of the algorithm.

The following pseudocode illustrates how the set $\mathcal{N}$ can be obtained in $O(D_{\max} \cdot \delta_d)$ time if our algorithm keeps track of all the possible shortest paths. Keeping track of all shortest paths can be easily achieved with an additional small constant overhead [$O(D_{\max})$ times more space] in the algorithm. This is done by adding an alternative parent attribute to every node in $\mathcal{Q}$ or $\hat{T}$ whenever an alternative shortest path is found [when $\mathtt{D(E(e)}, \hat{T}) = \mathtt{newdist}$ in Step 4 of the basic algorithm or when $\mathtt{D(n}, \hat{T}) + \Delta = \mathtt{D(n}, \mathcal{Q})$ in step ii), see Section VII].

```
K ← {y}
N ← ∅
P ← ∅
step (A)
if K = ∅
  Goto step (C)
else
  k ← EXTRACT(K)
  N ← {N, k}
step (B)
∀n ∈ B1(k, T̂)
  if n has an unmarked alternative parent
    mark k in node n
    P ← {P, n}
  else
    K ← {K, n}
if k is the alternative parent of some
  node n
  mark k in node n
    if all parent and alternative parents
  are marked in n
      K ← {K, n}
      P ← P − {n}
      unmark everything in n
Go to step (A)

step (C)
∀n ∈ P
  P(n, T̂) = any unmarked alternative parent
  of n
  remove the marked alternative parents of
  n
```

Because in practice $\lvert \mathtt{B}_{\max}(\mathtt{E(e)})\rvert$ hardly exceeds δ_d by much, this modification is not needed in practical cases. The modification is given mainly to obtain a better theoretical bound on the complexity of the initialization procedure.

B. Second Incremental Method

The following summarizes the complexity of some dynamic algorithms that can be derived using the second incremental method:

Bellman–Ford	$O(D_{\max} \cdot \delta_d^3)$
D'Esopo–Pape	*No Polynomial Upper Bound*
Linear Dijkstra	$O(\delta_{pd}\delta_d + \gamma \cdot D_{\max} \cdot \delta_d)$
Heap Dijkstra	$O(\gamma \cdot D_{\max} \cdot \delta_d \cdot \lg\delta_d)$

$$\text{Fibonacci Dijkstra} \qquad O(\delta_d \cdot \lg \delta_d + \gamma \cdot D_{\max} \cdot \delta_d) \qquad (4)$$

where γ denotes the *redundancy* factor, which represents the average time that each node is visited by the algorithm. This factor, although very close to 1 in practice, can take values between 1 and δ_{pd}.

Since in theory the term γ can be as large as δ_{pd}, the worst-case asymptotic complexity of the second incremental algorithm is worse than that of the first incremental algorithm. However, since in practice, there are very good reasons to believe that γ is very close to one, in practice the algorithms in the second incremental method generally outperform those in the first incremental method. The simulation results in [19] show how γ is generally small and why the second incremental method performs well.

IX. Conclusion

We have presented new dynamic algorithms for computing an SPT in a directed graph. Our dynamic algorithms are derived using a unified algorithmic framework, under which the best known SPT algorithms (static or dynamic) can also be derived as special instances. Within this algorithmic framework, we have proposed two specific approaches that transform known static SPT algorithms to new dynamic ones. These new dynamic algorithms have theoretical asymptotic complexity bounds that match the best known results. Among all the dynamic algorithms we studied, the algorithms using the first incremental method give rise to the best theoretical worst-case time bounds, while those using the second incremental method give rise to the best practical or average running times. The most promising algorithm for practical purposes is the Second Incremental Dijkstra algorithm. Its novelty is that it provably minimizes the number of link changes in the SPT.

Appendix
Proofs of Theoretical Results

A. Correctness of the Basic Algorithm

We will first show that the basic algorithm correctly computes a new SPT for the case where only one edge in the network changes its weight. We will then indicate how the proof can be extended to the case where multiple edges can change their weights.

Recall from Section III that the SPT for a given $\mathcal{G}$ is not unique. Let $\mathcal{S}$ be the set of SPTs in $\mathcal{G}$. For every $\mathcal{T} \in \mathcal{S}$, the distance attribute $\mathtt{D(n,}\ \mathcal{T})$ contains the shortest distance between node n and $Source(\mathcal{G})$. Note that even though the SPT is not unique, the distance attributes in $\mathcal{T} \in \mathcal{S}$ are identical, and hence $\mathtt{D(n,}\ \mathcal{T})$ is unique.

Lemma 1: In each stage of the basic algorithm, if $d = \mathtt{D(n,}\hat{\mathcal{T}}) \neq \infty$, d is the length of the path in $\hat{\mathcal{T}}$ between node n and $Source(\mathcal{G})$.

Proof: In the static version of initialization, the only finite distance attribute belongs to the source (= 0). In the dynamic version of initialization, every node n in $\hat{\mathcal{T}}$ has a distance attribute equal to the length of the path in $\hat{\mathcal{T}}$ between n and the source.

During steps 2–4, a new distance attribute for some node n is updated in $\hat{\mathcal{T}}$ by adding the distance attribute of n's new parent to the weight of the edge between the parent and n. Therefore, the new distance attribute represents the length of the some path in $\hat{\mathcal{T}}$ from the source to n going through the new parent node. ■

Lemma 2: In the basic algorithm, $\hat{\mathcal{T}}$ always maintains a tree structure.

Proof: It is clear that at the end of initialization, $\hat{\mathcal{T}}$ maintains a tree structure. During steps 2–4 of the basic algorithm, every node in $\hat{\mathcal{T}}$ has a distance that is smaller than the distance of any of its descendents. The only time when the structure of $\hat{\mathcal{T}}$ can change is during step 2 when a node n selected from the list $\mathcal{Q}$ changes parent in $\hat{\mathcal{T}}$. However, for the node n to be selected from $\mathcal{Q}$, the new parent must have a smaller distance than n. Therefore, the parent of n cannot be a descendent of n and hence there cannot be any cyclic path in $\hat{\mathcal{T}}$. Moreover, each node in $\hat{\mathcal{T}}$ has only one parent. Thus, $\hat{\mathcal{T}}$ maintains a tree structure. ■

Lemma 3: The basic algorithm will terminate.

Proof: A node n and its new attributes can only enter $\mathcal{Q}$ if the new distance attribute is smaller than the distance attribute for n in $\hat{\mathcal{T}}$. Because there are only a finite number of paths from the source to each node, using Lemma 2 and Lemma 1 we know that the distance attribute for some particular node can only improve a finite number of times. In other words, a node can only enter $\mathcal{Q}$ a finite number of times. Since at least one node is selected and removed from $\mathcal{Q}$ in each iteration, the algorithm must terminate. ■

For convenience, we introduce the following definition:

Definition 1: A node n is said to be *consolidated* at some step of the algorithm if its attributes in $\hat{\mathcal{T}}$ will not change further in the execution of the algorithm.

Lemma 4: If node n is enqueued in $\mathcal{Q}$ with its distance attribute $d_{\min}$ equal to its shortest distance from the source, after a finite number of steps, node n will be consolidated with its distance attribute $d_{\min}$.

Proof: By Lemma 3, we know that n will be eventually extracted from $\mathcal{Q}$ before the algorithm terminates. Since a node can enter $\mathcal{Q}$ only when its distance can be reduced, node n with its shortest distance $d_{\min}$ cannot re-enter $\mathcal{Q}$. In other words, n will be eventually consolidated. ■

Lemma 5: If node n is consolidated with its shortest distance, all descendents of n with respect to any SPT will be consolidated with their shortest distances.

Proof: It suffices to prove the statement for all children (next generation) nodes of n in any SPT since we can then secly the result recursively for all descendents. Let n' be any of ns children nodes connected to n via edge e in any SPT. Since n is consolidated with its shortest distance $d_{\min}$, the value $d_{\min} + \mathtt{W(e)}$ must be the shortest distance of n'.

After node n is consolidated with its shortest distance $d_{\min}$, during step 3, its child n' will either have an updated distance $d_{\min} + \mathtt{W(e)}$ or be examined in step 4 of the algorithm. In the former case, n' cannot improve its distance further and will be consolidated. In the latter case, n' will enter the list $\mathcal{Q}$ with distance $d_{\min} + \mathtt{W(e)}$, and according to Lemma 4, n' will also be consolidated eventually. ■

We now define what a germinal set is in order to proceed with the rest of the proof.

Definition 2: Let $\mathcal{D}$ be the set of nodes whose distance to the root through some path defined by some current rooted tree $\hat{T}$ is larger than the minimum distance. We define a germinal set $\mathcal{A}$ with respect to a current tree $\hat{T}$ to be any set of nodes whose descendents with respect to any valid SPT include all nodes in $\mathcal{D}$.

Lemma 6: Suppose at some instant in the execution of the basic algorithm, the tree data structure $\hat{T}$ is not an SPT. If all nodes in any germinal set $\mathcal{A}$ (with respect to current tree $\hat{T}$) are enqueued in the list $\mathcal{Q}$ with their shortest distances, iterating through steps 2–4 of the basic algorithm will result in $\hat{T}$ representing an SPT.

Proof: According to Lemma 5, the nodes in $\mathcal{D}$ (as explained in Definition 2) will be consolidated with their shortest distances. Since all distance attributes are distances of real paths (Lemma 1), no distance attribute can be less than the optimal distance. Therefore, the distance attributes of nodes not in $\mathcal{D}$ are their shortest distances, and they are already consolidated with their shortest distances. Hence, when all nodes are consolidated, $\hat{T}$ represents an SPT. ■

Any germinal set of nodes serves as "seeds" to (re)construct an entire SPT. The correctness of the basic algorithm then follows by proving that the initialization of the algorithm enqueues a germinal set of nodes $\mathcal{A}$ with their shortest distances in the list $\mathcal{Q}$.

Lemma 7: The initialization of the basic algorithm enqueues a germinal set of nodes with their shortest distances in the list $\mathcal{Q}$.

Proof: When the algorithm starts from scratch (static version), the initialization enqueues only $Source(\mathcal{G})$ (with the shortest distance being 0). Since any node is a descendent of the source node $Source(\mathcal{G})$ with respect to any SPT tree, any set containing $Source(\mathcal{G})$ is a complete germinal set.

Now consider the dynamic version of the initialization. Suppose an edge e in $\hat{T}$ (the original SPT) increases its weight by Δ and suppose that the distance of each node $\mathtt{B}_{\max}(\mathtt{E}(\mathtt{e}), \hat{T})$ has been updated with an increase of Δ. After the update, let $\mathcal{D}_{\hat{T}}$ be the set defined in Lemma 6. Note that $\mathcal{D}_{\hat{T}}$ is a subset of $\mathtt{B}_{\max}(\mathtt{E}(\mathtt{e}), \hat{T})$ containing those nodes each of which can find a shorter path from the source without passing through e. These alternative paths must pass through nodes in $\mathcal{R} = \mathtt{B}_{\max}(\mathtt{E}(\mathtt{e}), \hat{T})$ that can decrease their distance label in $\hat{T}$ by choosing a parent $p \notin \mathcal{R}$. The set $\hat{\mathcal{A}}$ of all such nodes is a complete germinal set and will be enqueued with their shortest distances after the initialization procedure.

On the other hand, suppose the weight of an edge e in $\hat{T}$ decreases. The set $\mathcal{D}_{\hat{T}}$ then consists of nodes each of which can find a shorter path from the source through the edge e. The nodes in $\mathcal{R} = \mathtt{B}_{\max}(\mathtt{E}(\mathtt{e}), \hat{T})$ are first removed from $\mathcal{D}_{\hat{T}}$ by updating their new distance attributes (the parent attribute remains the same). The remaining nodes in $\mathcal{D}$ are not in $\mathcal{R}$, but their shortest paths must pass through nodes in $\mathcal{R}$. The set of nodes not in $\mathcal{R}$ that can decrease their distance label by selecting a parent in $\mathcal{R}$ is therefore a complete germinal set and will be enqueued with their shortest distances after the initialization procedure. ■

Theorem 1: Upon termination of the basic algorithm, $\hat{T}$ represents an SPT with all nodes in $\mathcal{G}$ which are reachable from $Source(\mathcal{G})$.

Proof: From Lemma 3 we know that the basic algorithm will terminate. From Lemma 7 we know that after initialization, $\mathcal{Q}$ will contain a complete germinal set. According to Lemma 6 the basic algorithm will compute an SPT. ■

To show that the basic algorithm also works when multiple edge weights are changed at once (see end of Section IV), we need to show that the set of enqueued nodes after the extended initialization procedure also constitutes a complete germinal set.

For the case where there are multiple edge weight increments, during initialization, every node in $\mathcal{D}_{\hat{T}}$ (same meaning as in Lemma 7) is first updated with a larger distance attribute. Then, all the nodes in $\mathcal{D}_{\hat{T}}$ that can get a smaller distance attribute through a parent outside of the updated area $\mathcal{R}$ will be enqueued in $\mathcal{Q}$. Since all the new alternative paths must pass through these enqueued nodes, they form a complete germinal set.

For the case where there are multiple edge weight decrements, it is sufficient to observe that any node that is affected by the changes (ultimately decrements its distance from the source) must be a descendent (in one of the new SPT tree) of the end node of one of the edges whose weights have been reduced. Since the set of these end nodes is a complete germinal set, the set which includes all their direct descendents that can improve their distance (and are enqueued in $\mathcal{Q}$) must also be a complete germinal set.

B. Algorithmic Complexity

1) First Incremental Method:

Theorem 2: The First Incremental Bellman–Ford algorithm has a complexity of $O(D_{\max} \cdot \delta_d^2)$.

Proof: Let $\mathcal{R}_d$ denote the set of δ_d nodes whose distance attributes must change. The nodes in $\mathcal{R}_d$ are extracted in cycles. After extraction cycle i, each visited node in $\mathcal{R}_d$ has a distance attribute that is not more than the length of the shortest of all paths from the source that have at most i hops inside in $\mathcal{R}_d$. Since the shortest path has at most $(\delta_d - 1)$ hops in $\mathcal{R}_d$, the distance attribute of each nodes in $\mathcal{R}_d$ cannot change after extraction cycle $(\delta_d - 1)$. Each extraction cycle involves at most δ_d nodes (only nodes in $\mathcal{R}_d$ will enter $\mathcal{Q}$). After a node extraction, at most $D_{\max}$ edges will be visited. Therefore, the complexity of this algorithm is $O(D_{\max} \cdot \delta_d^2)$. ■

To derive the complexity for the dynamic Dijkstra algorithms, we first prove the following lemmas.

Lemma 8: Given any SPT T, let $\mathcal{T}_{\min}^N$ be any tree containing nodes with the N smallest distance attributes. Also let $\mathcal{N}_{\text{new}}^N$ denote the set of nodes not in $\mathcal{T}_{\min}^N$ but adjacent to nodes in $\mathcal{T}_{\min}^N$. Using the Dijkstra algorithm, every node $n \in \mathcal{N}_{\text{new}}^N$ chooses a parent node $p \in \mathcal{T}_{\min}^N$ such that its distance attribute $\mathtt{D}(\mathtt{p}, \mathcal{T}_{\min}^{\mathtt{N}}) + \mathtt{W}(\mathtt{n}, \mathtt{p})$ is minimized. The node n with the smallest distance attribute in $\mathcal{N}_{\text{new}}^N$ has the $(N+1)$ smallest distance attribute in any SPT.

Proof: The distance attribute of every node in any SPT T is larger than that of any of its ancestor nodes. If n_{N+1} is the node with the $(N+1)$ smallest distance attribute for any $T \in S$, the parent of n_{N+1} must be in $\mathcal{T}_{\min}^N$. Therefore, n_{N+1} must be in $\mathcal{N}_{\text{new}}^N$ and its distance attribute in N_{new}^N must be the same as that in $T \in \mathcal{S}$. The other nodes in $\mathcal{N}_{\text{new}}^N$ will have a distance attribute

that is at least their distance attribute in $\mathcal{T}$. Therefore, n_{N+1} is the node in $\mathcal{N}_{\text{new}}^{N}$ with the smallest distance attribute. ■

Lemma 9: In the Dijkstra algorithms, if a node n is extracted from $\mathcal{Q}$, it will not re-enter $\mathcal{Q}$ before the termination of the algorithm.

Proof: The proof is by induction. It can be shown that after the initialization, the first node to be extracted from $\mathcal{Q}$ has its distance attribute equal to its shortest distance, and thus it will not re-enter $\mathcal{Q}$. We now assume that the set of nodes $\mathcal{N}_{\text{ext}}$ extracted previously have shortest distance attributes. At any time, $\mathcal{Q}$ includes all the nodes adjacent to nodes in $\mathcal{N}_{\text{ext}}$. Each node $n \in \mathcal{Q}$ has the parent in $\mathcal{N}_{\text{ext}}$ such that its distance attribute is minimized. According to Lemma 8, the next node extracted will have a shortest distance attribute. By induction, every node extracted must have a shortest distance, and therefore cannot re-enter $\mathcal{Q}$.

Theorem 3: The First Incremental Linear Dijkstra algorithm has a complexity of $O(\delta_d^2 + D_{\max} \cdot \delta_d)$, the First Incremental Heap Dijkstra has a complexity of $O(D_{\max} \cdot \delta_d \cdot \lg\delta_d)$, and the First Incremental Fibonacci Dijkstra has a complexity of $O(\delta_d \cdot \lg\delta_d + D_{\max} \cdot \delta_d)$.

Proof: After the initialization, $\mathcal{Q}$ contains all the nodes that can improve their distance attribute by choosing a parent with a shortest distance attribute. From Lemma 9, we know that when a node is extracted from $\mathcal{Q}$ it will never re-enter $\mathcal{Q}$. Therefore there will be a maximum of δ_d extractions. After each extraction all the edges leaving the extracted node are visited, which in turn will update the distance attribute of some node in $\mathcal{Q}$.

Since there are at most δ_d nodes in $\mathcal{Q}$, extracting a node from $\mathcal{Q}$ takes $O(\delta_d)$ complexity with linear search and $O(\lg\delta_d)$ complexity with binary and Fibonacci heaps. Updating a distance attribute in $\mathcal{Q}$ takes $O(1)$ complexity with a regular list and a Fibonacci heap, while requiring $O(\lg\delta_d)$ complexity with a binary heap. Therefore, the complexity of First Incremental Linear Dijkstra is $O(\delta_d^2 + D_{\max} \cdot \delta_d)$, that of First Incremental Heap Dijkstra is $O(D_{\max} \cdot \delta_d \cdot \lg\delta_d)$, and that of First Incremental Fibonacci Dijkstra is $O(\delta_d \cdot \lg\delta_d + D_{\max} \cdot \delta_d)$. ■

2) Second Incremental Method:

Theorem 4: The Second Incremental Bellman–Ford algorithm has a complexity of $O(D_{\max} \cdot \delta_d^3)$.

Proof: As in the proof of Theorem 2, we will denote as $\mathcal{R}_d$ the set of δ_d nodes whose distance attributes must change. After extraction cycle i, each visited node in $\mathcal{R}_d$ has a distance attribute that is not more than the length of the shortest of all paths from the source that have at most i hops inside in $\mathcal{R}_d$. As in the first incremental case at most δ_d^2 nodes will be extracted from $\mathcal{Q}$. However in the second incremental case, after extracting one node, the algorithm might visit at most all the edges $(D_{\max} \cdot \delta_d)$ in $\mathcal{R}_d$. Therefore the complexity of this algorithm is $O(D_{\max} \cdot \delta_d^3)$. ■

Lemma 10: In the Second Incremental (Linear or Heap) Dijkstra algorithm, only the nodes (δ_{pd} of them) whose parent attribute must necessarily change are extracted exactly once from $\mathcal{Q}$.

Proof: Every node inserted in $\mathcal{Q}$ (step 4) must be outside the branch $\mathrm{B}(\mathrm{y}, \hat{T})$, while its parent in $\mathcal{Q}$ is inside the branch. Therefore, any node n inserted in $\mathcal{Q}$ must have a parent attribute in $\mathcal{Q}$ that is different from that in $\hat{T}$.

Let $\mathcal{P}$ be the set of nodes that have been consolidated with optimal distance attributes in $\hat{T}$. At any time after initialization, $\mathcal{Q}$ contains all nodes adjacent to nodes in $\mathcal{P}$ that can improve their distance attribute by changing the parent attribute in $\hat{T}$ to some node in $\mathcal{P}$. According to Lemma 8, the nodes extracted from $\mathcal{Q}$ will be consolidated with incrementing distance attributes.

Furthermore, when node n is extracted from $\mathcal{Q}$ with a shortest distance attribute d, every other node with shortest distance attribute less than d has already been consolidated. Since n was not dequeued from $\mathcal{Q}$, the segment between n and $\mathrm{P}(\mathrm{n}, \hat{T})$ cannot be part of an SPT. Therefore, node n *must* change its parent attribute to the new one contained in $\mathcal{Q}$. ■

Theorem 5: The Second Incremental (Linear or Heap) Dijkstra algorithm obtains a new valid SPT in $\hat{T}$, while modifying the minimum number of parent attributes in $\hat{T}$.

Proof: During initialization, the only nodes that change parent attributes are the descendent of the modified edge and the descendents of the incrementing edge (in $\hat{T}$) that can maintain the same distance attribute by adopting an alternative parent.

During the rest of the algorithm, a parent attribute can be changed only when a node is extracted from $\mathcal{Q}$. According to Lemma 10, only the nodes that have a parent in $\hat{T}$ different from every parent in $T \in S$ will change the parent attributes. ■

Theorem 6: The complexity of the Second Incremental Linear Dijkstra algorithm is $O(\delta_{pd}\delta_d + \gamma \cdot D_{\max} \cdot \delta_d)$, that of Second Incremental Heap Dijkstra is $O(\gamma \cdot D_{\max} \cdot \delta_d \cdot \lg\delta_d)$, and that of Second Incremental Fibonacci Dijkstra is $O(\delta_d \cdot \lg\delta_d + \gamma \cdot D_{\max} \cdot \delta_d)$.

Proof: According to Lemma 10, only δ_{pd} nodes will be extracted from Q. After a node extraction at most δ_d nodes will be visited. Therefore the total number of edges visited is $\gamma \cdot D_{\max} \cdot \delta_d$, where γ is at worst δ_{pd}.

For Second Incremental Linear Dijkstra, an extraction requires $O(\delta_d)$ complexity while an update or removal requires $O(1)$ complexity. Therefore the total complexity is $O(\delta_{pd}\delta_d + \gamma \cdot D_{\max} \cdot \delta_d)$. For Second Incremental Heap Dijkstra, every operation on $\mathcal{Q}$ has cost $O(\lg\delta_d)$. In the worst case, some operations might be done on $\mathcal{Q}$ every time a node is visited. Therefore the complexity is $O(\gamma \cdot D_{\max} \cdot \delta_d \cdot \lg\delta_d)$. For the Second Incremental Fibbonacci Dijkstra, an extraction or removal requires $O(\lg\delta_d)$ while an update only requires $O(1)$ complexity. Therefore, its total complexity is $O(\delta_d \cdot \lg\delta_d + \gamma \cdot D_{\max} \cdot \delta_d)$. ■

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," *Q. Appl. Math.*, vol. 16, pp. 87–90, 1958.

[3] D. Bertsekas, *Linear Network Optimization: Algorithms and Codes*, MA, Cambridge: MIT Press, 1991.

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

[5] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[6] S. Deering and D. Cheriton, "Multicast routing in datagram internetworks and extended LANs," *ACM Trans. Comput. Syst.*, vol. 8, no. 2, pp. 85–110, May 1990.

[7] E. Dijkstra, "A note two problems in connection with graphs," *Numerical Math.*, vol. 1, pp. 269–271, 1959.

[8] E. Feuerstein and A. Marchetti-Spaccamela, "Dynamic algorithms for shortest paths in planar graphs," *Theoretical Comput. Sci.*, vol. 116, pp. 359–371, 1993.

[9] P. Franciosa, D. Frigioni, and R. Giaccio, "Semi-dynamic shortest paths and breadth-first search in digraph," in *Proc. 14th Annu. Symp. Theoretical Aspects of Computer Science*, Mar. 1997, pp. 33–46.

[10] 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*, Atlanta, GA, 1998, pp. 212–221.

[11] ——, "Incremental algorithms for single-source shortest path trees," in *Proc. Foundations of Software Technology and Theoretical Computer Science*, Dec. 1994, pp. 113–124.

[12] J. J. Garcia-Luna-Aceves and S. Murthy, "A path finding algorithm for loop-free routing," *IEEE/ACM Trans. Networking*, pp. 148–160, Feb. 1997.

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

[14] G. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Incremental algorithms for minimal length paths," *J. Algorithms*, vol. 12, pp. 615–638, 1991.

[15] C. Labovitz, G. Malan, and F. Jahanian, "Internet routing instability," in *Proc. SIGCOMM'97*, Sept., pp. 115–126.

[16] J. Moy, "OSPF version 2," Cascade Communications Corp., Internet Draft, RFC 2178, July 1997.

[17] ——, *OSPF: Anatomy of an Internet Routing Protocol*. Reading, MA: Addison-Wesley, 1998.

[18] J. McQuillan, I. Richer, and E. Rosen, "The new routing algorithm for the ARPANET," *IEEE Trans. Commun.*, vol. COM-28, pp. 711–719, May 1980.

[19] P. Narvaez, K.-Y. Siu, and H.-Y. Tzeng. (1998, May) New dynamic algorithms for shortest path tree computation. Bell Labs, Lucent Technologies. [Online]. Available: http://perth.mit.edu/ pnarvaez/publications.html

[20] V. Paxson, "End-to-end routing behavior in the Internet," *IEEE/ACM Trans. Networking*, vol. 5, pp. 601–615, Oct. 1997.

[21] R. Perlman and G. Varghese, "Pitfalls in the design of distributed routing algorithms," in *Proc. SIGCOMM'88*, Aug., pp. 43–54.

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

[23] B. Rajagopalan and M. Faiman, "A responsive distributed shortest-path routing algorithm within autonomous systems," *Internetworking: Research and Experience*, vol. 2, no. 1, pp. 51–69, Mar. 1991.

[24] G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-path problem," *J. Algorithms*, vol. 21, pp. 267–305, 1996.

[25] Y. Rekhter and T. P. Gross, "Applications of the border gateway protocol in the Internet," DDN Network Information Center, RFC 1772, Mar. 1995.

[26] M. Schwartz and T. Stern, "Routing techniques used in computer communication networks," *IEEE Trans. Commun.*, vol. 28, pp. 539–552, Apr. 1980.

[27] P. Spira and A. Pan, "On finding and updating spanning trees and shortest paths," *SIAM J. Computing*, vol. 4, no. 3, pp. 375–380, Sept. 1975.

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

[29] R. Tarjan, "Amortized computational complexity," *SIAM J. Algorithms and Discrete Methods*, vol. 6, no. 2, pp. 306–318, Apr. 1985.

[30] *BGP-4 Protocol Analysis*: DDN Network Information Center, Mar. 1995.

[31] L. Wei. Random topology generator (RTG). Univ. Southern California, Los Angeles, CA. [Online]. Available: http://netweb.usc.edu/daniel/research/sims/topology/

Paolo Narvácz received the S.B., M.Eng. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, in 1996, 1997, and 2000, respectively.

As a Research Assistant at MIT, he worked on flow control algorithms, TCP dynamics, and routing algorithms. Between 1997 and 1999, he was with the High Speed Networking Department, Bell Labs, Lucent Technologies, Holmdel, NJ. His work at Bell Labs involved algorithms for fast routing recomputation, multiple path routing, and protocols for fast local routing restoration. He is currently a Systems Architect at Sycamore Networks, Chelmsford, MA.

Kai-Yeung Siu received the B.S. degree (*summa cum laude*) in mathematics and computer science from New York University, New York, and the B.Eng. degree (*summa cum laude*) in electrical engineering from The Cooper Union, New York, both in 1987. He received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1988 and 1991, respectively.

From 1989 to 1990, he was a Research Student Associate at the IBM Almaden Research Center, San Jose, CA. From 1991 to 1995, he was Assistant Professor of electrical and computer engineering at the University of California, Irvine. He joined the Massachusetts Institute of Technology (MIT), Cambridge, MA, in 1996, and is currently Associate Professor and holder of the d'Arbeloff Career Development Chair at MIT. He is with the d'Arbeloff Laboratory for Information Systems and Technology of Mechanical Engineering and also affiliated with the Laboratory for Information and Decision Systems of Electrical Engineering and Computer Science. He is also the Research Director of the MIT Auto-ID Center, an industry-funded center which develops next-generation automatic identification systems with e-commerce applications. He has published over 100 research papers in the areas of optical networking, wireless communications, Internet routing and congestion control protocols, parallel and distributed algorithms, and computational complexity theory. He has also served as a Consultant for major Internet equipment vendors and service providers.

Hong-Yi Tzeng, photograph and biography not available at time of publication.