Research Contribution

On the Dynamic Shortest Path Problem

CHIH-CHUNG LIN* and RUEI-CHUAN CHANG**

This paper proposes an algorithm to solve the *dynamic shortest path problem*, which is to perform an arbitrary sequence of two kinds of operations on a directed graph with edges of equal length: the *Insert* operation, which inserts an edge into the graph, and the *FindShortest* operation, which reports the shortest path between a pair of vertices if such a path exists. Each *FindShortest* operation can be done in $O(k)$ time, where $k \leq n$ is the number of edges on the shortest path, and an arbitrary sequence of at most $O(n^2)$ *Insert* operations can be done in a worst-case time of $O(n^3 \log n)$. Furtheremore, our algorithm can be extended to perform the cost-decreasing operation of the least-cost path problem, and always takes less time than previous algorithms.

1. Introduction

The design and analysis of efficient dynamic data structures for the representations of graphs has been extensively studied in the lierature [2-14, 16]. In this paper, we consider the *dynamic* shortest path problem. The problem can be formulated as follows. Let $G=(V, E)$ be a directed graph, where V is a set of vertices and E is a set of edges of equal length, say l. How is it possible to maintain a dynamic data structure under a sequence of intermixed operations of the following two kinds on G

Insert(u, v): Insert an edge between vertices u and v.
FindShortest(u, v): Report the shortest path from vertex u to vertex v if such a path exists.

Intuitively, we can represent the graph G by just keeping the edges in G. The *Insert* operation can be implemented in constant time, while the *FindShortest* operation requires $O(m)$ time when a breadth-first search algorithm is used [1]. Hence, in order to reduce the worst-case running time of *FindShortest* operations, we must maintain more path information while performing the *Insert* operation.

Many data structures have been proposed recently to support efficient update operations on graphs, and are useful for the on-line computation of graph problems [2-14, 16]. In the following, m and n denote the number of edges and vertices in a graph, respectively. For directed graphs, the on-line computation of transitive closure was considered by Ibaraki and Kato [5]. This maintains transitive closure in $O(n^3)$ and $O(n^2(m+n))$ time for any number of edge insertions and deletions, respectively. Rohnert [8] proposed algorithms for the on-line least-cost problem, which requires $O(n^2)$ worst-case time for an edge insertion and $O(mn)$ time for an edge deletion. Recently, Italiano [6, 7] discussed the path retrieval problem of how to report a path between two vertices in a graph while edges are successively inserted or deleted.

The dynamic shortest-path problem discussed in this paper is an extension to the path retrieval problem [6]. In the path retrieval problem, once a path from u to v is established, we do not have to update the path while inserting new edges. But in the shortest-path problem, a newly inserted edge may create a path shorter than the existing one, and therefore more information on the data structure must be kept.

The paper has two main results. First, we propose the data structure and algorithms that allow each *FindShortest* operation to be done in $O(k)$ time, where $k \leq n$ is the number of edges on the reported path, and an arbitrary sequence of at most $O(n^2)$ *Insert* operations can be accomplished in a worst-case time of $O(n^3 \log n)$. Furthermore, the algorithm can be extended to perform the cost-decreasing operation of the least-cost path problem [8] and always takes time less than the algorithm proposed by Rohnert [8].

The rest of the paper is organized as follows. In Section 2, we present the data structures and algorithms for the dynamic shortest-path problem. The time complexity analysis of the algorithms is discussed in Section 3. Concluding remarks are given in Section 4.

2. The Data Structure and Algorithms

In this section we present data structures and algorithms for maintaining on-line information about the shortest paths of a directed graph under a sequence of *Insert* and *FindShortest* operations. Given a direct graph $G=(V, E)$ and a node $v \in V$, a *shortest-path tree* rooted at v is a tree T that contains all the vertices

*Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.
**Institute of Information Science, Academia Sinica, Taipei, Taiwan, R.O.C.

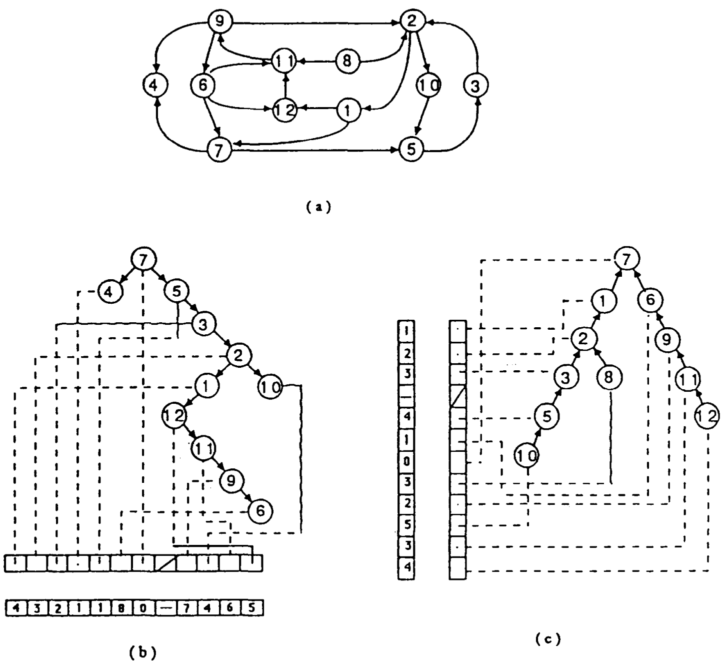

Fig. 1 An example of the data structure. (a) A directed graph G. (b) *T_Source*(7) and the seventh row of the matrices *P_Source* and *Distance*. (c) *T_Sinc*(7) and the seventh column of the matrices *P_Sink* and *Distance*.

reachable from v, and the path from root to node x in T is the shortest path from v to x.

We augment each vertex $v \in V$ with two sets:

Source(v): the set of vertices that are reachable from v.
Sink(v): the set of vertices that can reach v.

and organize them as *shortest-path trees* rooted at v, denoted by *T_Source*(v) and *T_Sink*(v), respectively.

To access each vertex in the *shortest-path trees* efficiently, we use two n-by-n matrices of pointers, defined as follows:

P_Source[u, v]: points to the vertex v in *T_Source*(u) if $v \in$ *Source*(u); otherwise, contains a null pointer.
P_Sink[u, v]: points to the vertex u in *T_Sink*(v) if $u \in$ *Sink*(v); otherwise, contains a null pointer.

We also record the distances of the shortest paths between each pair of vertices in an n-by-n matrix:

Distance[u, v]: the distance of the shortest path from u to v.

Figure 1 is an example of the data structure while $l=1$.

The *FindShortest* operation can be done easily. If *P_Sink*[u, v] contains a *null* pointer, then there is no path from u to v in G. Otherwise, we can use *P_Sink*[u, v] to access the vertex u in the shortest path tree *T_Sink*(v) and report the shortest path by the following procedure:

```
Procedure FindShortest(u, v)
  p: = P_Sink(u, v);
  while p ≠ null do
    print(P);
    p: = "the parent of P in T_Sink(v)"
  od;
end FindShortest;
```

The implementation of the *Insert* operation is more complex. Consider the graph G. Inserting an edge (u, v) of length l may change the shortest paths of some pairs of vertices to the new paths passing edge(u, v). If (x, t) is such a pair, then the following statements must hold:

(1) x may reach u (i.e. $x \in$ *Sink*(u))
(2) t is reachable from v (i.e. $t \in$ *Source*(v))
(3) *Distance*[x, t] > *Distance*[x, u] + l + *Distance*[v, t]

(1) and (2) are obvious from the fact that the newly established path must pass the edge (u, v), and (3) is the condition that causes the shortest path from x to t to be updated, where the right-hand side is the distance of the new shortest path. Hence, to implement the *Insert*(u, v) operation, we can check condition (3) for every pair (x, t), where $x \in$ *Sink*(u) and $t \in$ *Source*(v), and update the data structures if condition (3) is satisfied.

However, it is not necessary to check *all* these pairs. In fact, we may *omit* some of these checking steps by considering the properties of the *Source* and *Sink* trees. We will explain this after the definitions.

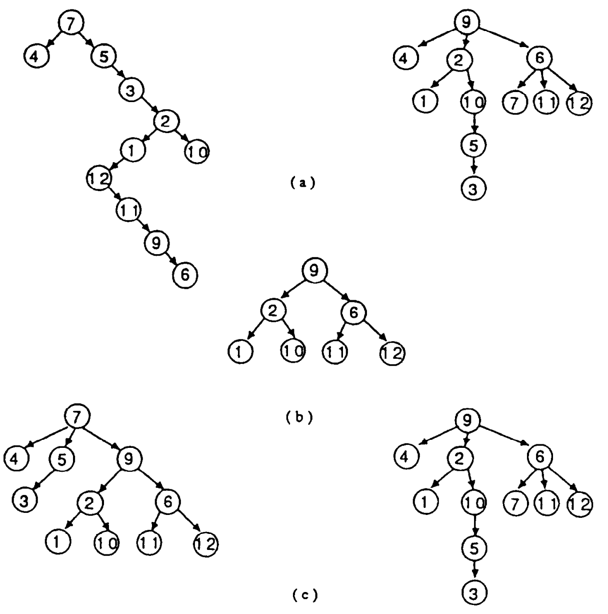

Fig. 2 (a) *T_Source*(7) and *T_Source*(9) before *Insert*(7, 9). (b) $C(7)$. (c) *T_Source*(7) and *T_Source*(9) after *Insert*(7, 9), note that $C(7)=D_7(9)$.

Definition: for $x,\ t\in V$

$D_x(t)=\{v\mid v$ is the descendant of t (including t) in *T_Source*$(x)\}$

$C(x)=\{t\mid Distance[x,t]$ changes after $Insert(u,v)\}$

$X=\{x\mid C(x)\neq\phi\}$

Clearly, $C(x)\subseteq Source(v)$ and $X\subseteq Sink(u)$ after we insert the edge (u, v) into G.

Lemma 2.1: After the operation $Insert(u, v)$, $C(x)=D_x(v)$ for every $x\in X$.

Proof: The property is obvious because if $t\in C(x)$, the new shortest path from x to t must pass the edge (u, v), i.e. $t\in D_x(v)$. On the other hand, if $t\in D_x(v)$ after the insertion, since $x\in X$, the shortest path from x to t must pass the edge (u, v) i.e. $t\in C(x)$. □

An example of Lemma 2.1 is shown in Fig. 2.

Lemma 2.2: For $Insert(u, v)$, we have $t\notin C(x)\Rightarrow w\notin C(x)$ and $t\notin C(y)$ for every $w\in D_v(t)$, and y is the descendant of x in *T_Sink*(u), where $x\in Sink(u)$ and $t\in Source(v)$.

Proof: Once we perform $Insert(u, v)$, $t\notin C(x)$ means

$$Distance[x,u]+l+Distance[v,t]\geq Distance[x,t]$$

Considering *T_Source*(v), for every $w\in D_v(t)$, v, t, and w are in the same path of *T_Source*(v), so

$$Distance[v,w]=Distance[v,t]+Distance[t,w]$$

hence

$$\begin{aligned}&Distance[x,u]+l+Distance[v,w]\\&\geq Distance[x,t]+Distance[t,w]\\&\geq Disttance[x,w]\end{aligned}$$

i.e. $w\notin C(x)$

On the other hand, we may treat *T_Sink*(u) in the same way as *T_Source*(v) by reversing edge directions, and get the result $t\notin C(y)$. □

Combining Lemmas 2.1 and 2.2, we have the following theorem, which is useful in designing an efficient algorithm for the operation $Insert(u, v)$.

Theorem 2.1: For the operation $Insert(u, v)$, if $x\in Sink(u)$ and y is the descendant of x in *T_Sink*(u), we have

(1) $C(x)=\phi\Rightarrow C(y)=\phi$

(2) $C(y)\subseteq D_x(v)\subseteq Source(v)$ after the update of *T_Source*(x) for $x\in X$.

Proof: Obvious from Lemmas 2.1 and 2.2. □

Hence, if we check condition (3) for the pair (x,t) after the edge (u, v) is inserted and find that it is not satisfied (i.e. $t\notin C(x)$), then we need not check the pair (y, w) for all y and w, the descendants of x and t in *T_Sink*(u) and *T_Source*(v), respectively.

To implement $Insert(u, v)$, we may update *T_Source*(x) for each x in *T_Sink*(u) in *depth-first* order. If $C(x)=\phi$, we need not update *T_Source*(y) for all y, the descendant of x in *T_Sink*(u). Furthermore, we update *T_Source*(x) for $x\neq u$ by checking the pair (x, t) for every t in $D_w(v)$ in depth-first order, where w is the parent of x in *T_Sink*(u). In this way we always traverse a subtree of *T_Source*(v) and the subtrees will become smaller and smaller as we update *T_Source*(x), $x\in Sink(u)$ (see Fig. 3). The complete algorithm for $Insert(u, v)$ is given below.

Procedure *DFT_Sink*(x, w, u, v) makes a *depth-first* traversal to *T_Sink*(u) to update *T_Source*(x) for each x

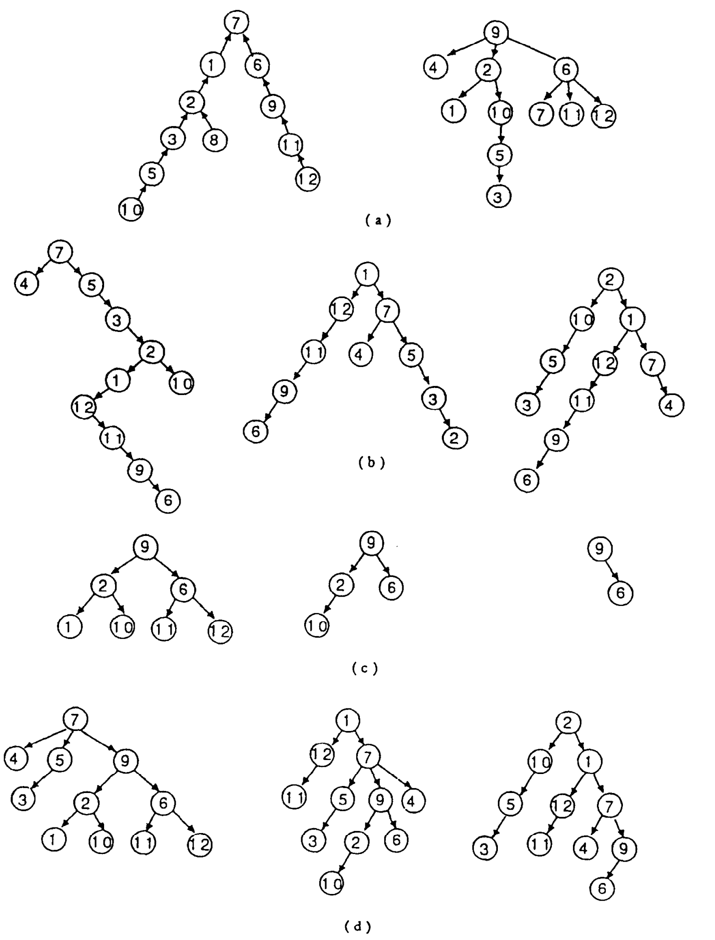

Fig. 3 Update *T_Source*(7), *T_Source*(1) and *T_Source*(2) for *Insert*(7, 9). (a) *T_Sink*(7) and *T_Source*(9). (b) *T_Source*(7), *T_Source*(1) and *T_Source*(2) before *Insert*(7, 9). (c) *C*(7), *C*(1), *C*(2) caused by traversing *T_Source*(9), $D_7(9)$ and $D_1(9)$, respectively. (d) *T_Source*(7), *T_Source*(1) and *T_Source*(2) after *Insert*(7, 9).

in *T_Sink*(*u*). Note that *w* is the parent of *x* in *T_Sink*(*u*) except in the first loop of *DFT_Sink*, in which *x* stands for *u* and *w* stands for *v*. We call *w* the *predecessor* of *x* in the following section. In fact, *w* being the *predecessor* of *x* means that we will traverse *T_Source*(*w*) (or $D_w(v)$) while updating *T_Source*(*x*).

```
Procedure Insert(u, v);
  DFT_Sink(u, v, u, v);
end Insert;
```

```
Procedure DFT_Sink(x, w, u, v);
  Update(x, w, u, v)
  if Distance[x, u] + l < Distance[x, v] then
    do for every y son of x in T_Sink(u)
      DFT_Sink(y, x, u, v)
    od
  fi
end DFT_Sink;
```

```
Procedure Update(x, w, u, v);
  if Distance[x, u] + l < Distance[x, v] then
    Distance[x, v] := Distance[x, u] + l;
    if P_Source[x, v] = null then
      create vertices pointed by
      P_Source[x, v] and P_Sink[x, v]
    fi;
    update the links of P_Source[x, v]
    to son of u in T_Source(x);
    update the links of P_Sink[x, v]
    to son of w in T_Sink(v);
    do for every t son of v in T_Source(w)
      Update(x, w, v, t)
    od
  fi
end update;
```

Remark *Insert*(*u*, *v*) can be modified simply by making

l, the length of edge (u, v), a parameter for performing edge insertion in a graph, without the restriction of all edges having equal lengths. Hence we have two versions of the *Insert* algorithm: *Insert*(u, v) and *Insert*(u, v, l). Both have the same time complexity and do exactly the same thing except that *Insert*(u, v, l) regards l as a parameter which is decided by the inserted edge. *Insert*(u, v, l) can be used to implement the cost-decreasing operation of the least-cose path problem proposed by Rohnert [8].

3. Time Complexity Analysis

In this section, we shall analyze the time complexity of *FindShortest* and *Insert*. Let n denote the number of vertices in V and m the number of edges inserted.

Obviously, the time complexity of the *FindShortest* operations if $O(k)$, where $k \leq n$ is the number of edges of the shortest path.

For the *Insert* operation, there are two nested recursive procedures, *DFT_Sink* and *Update*. Procedure *DFT_Sink*(x, w, u, v) traverses *T_Sink*(u) and procedure *Update*(x, w, u, v) traverses *T_Source*(w).

Theorem 3.1: The time complexity of *Insert*(u, v) is

$$O(\sum_{x\in X} |C(x)| + indeg_{T\_Sink(u)}(X) + \sum_{x\in X} outdeg_{T\_Source(w)}(C(x)))$$

where $indeg_{T\_Sink(u)}(X) = \Sigma_{x\in X}\, indeg_{T\_Sink(u)}(x)$, $outdeg_{T\_Source(w)}(C(x)) = \Sigma_{t\in C(x)}\, outdeg_{T\_Source(w)}(t)$, and $indeg_{T\_Sink(u)}(x)$ and $outdeg_{T\_Source(w)}(x)$ are the number of tree sons of x in *T_Sink*(u) and *T_Source*(w), respectively.

Proof: The time complexity is straightforward from the *Insert*(u, v) algorithm. The first term, $\Sigma_{x\in X}|C(x)|$, is the number of operations for the actual updating of data structures; the second term, $indeg_{T\_Sink(u)}(X)$, is the number of steps to traverse *T_Sink*(u); and the last term, $\Sigma_{x\in X}\, outdeg_{T\_Source(w)}(C(x))$, is the number of steps to traverse *T_Source*(w) for all w, the predecessor of $x\in X$. □

As mentioned in the last section, we sould like to compare the Insert algorithm with the cost-decreasing algorithm [8].

The cost-decreasing algorithm works in a similar manner and the time complexity can be expressed as:

$$O(\sum_{x\in X} |C(x)| + indeg_{T\_Sink(u)}(X) + \sum_{x\in X} outdeg_{T\_Source(v)}(C(x))).$$

Corollary: The Insert algorithm always takes time less than the cost-decreasing algorithm [8].

Proof: The only difference of the above equation and the equation shown in Theorem 3.1 is the last term. This is because the cost-decreasing algorithm updates *T_Source*(x) for each $x\in X$ by traversing *T_Source*(v), whereas our algorithm does it by traversing $D_w(v)$, which is always a subtree of *T_Source*(v) as discussed in the previous section. Obviously, $outdeg_{T\_Source(w)}(C(x)) \leq outdeg_{T\_Source(v)}(C(x))$ for every $x\in X$. Hence the corollary holds. □

In the following, we shall show that a sequence of at most $O(n^2)$ *Insert* operations can be accomplished in $O(n^3 \log n)$ time.

For convenience of analysis, we will concentrate on one of the *Source* trees, say *T_Source*(x). We measure the time it takes, in the worst case, for the updates of *T_Source*(x) over a sequence of *Insert* operations. The total time required for the sequence of operations is the update time for all the n *Source* trees.

Let us examine the *Insert*(u, v) algorithm. While updating one of the *Source* trees, say *T_Source*(x), for the insertion of edge (u, v), we need to traverse $D_w(v)$ for w, the *Predecessor* of x, to check the pair (x, t), $t\in D_w(v)$. What we want to measure is the number of vertices visited while traversing $D_w(v)$ (that is, the number of vertex pairs checked) over an arbitrary sequence of at most $O(n^2)$ *Insert* operations.

Let $C_i(x)$ denote the $C(x)$ for the ith insertion.

Definition:

$T_i(x) = \{v \mid$ vertex pair (x, v) is checked while updating *T_Source*(x) in *Insert* algorithm for the ith insertion$\}$

$N_i(x) = \{v \mid$ vertex pair (x, v) is checked, but *Distance*$[x, v]$ does not change for the ith insertion$\}$

Obviously, $T_i(x) = C_i(x) \cup N_i(x)$ and $\Sigma_i|T_i(x)| = \Sigma_i|C_i(x)| + \Sigma_i|N_i(x)|$. We bound $\Sigma_i|T_i(x)|$ by measuring $\Sigma_i|C_i(x)|$ and $\Sigma_i|N_i(x)|$.

To measure $\Sigma_i|C_i(x)|$, let us observe the changes in the "depths" of the vertices in *T_Source*(x) while updating *T_Source*(x) over a sequence of *Insert* operations. We first recursively defined the depth of a vertex as follows:

Definition:

$$Depth_x(v) = \begin{cases} 0 & \text{if } v = x \\ n & \text{if } v \notin Source(x) \\ Depth_x(u) + 1 & \text{otherwise} \end{cases}$$

where u is the parent of v in *T_Source*(x)

Note that if $v\in Source(x)$, $Depth_x(v) = Distance[x, v]$ as the length l of each edge is equal to one. If a vertex $v\in C_i(x)$, then $Depth_x(v)$ decreases *at least* one for the ith *Insert* operation. In fact, $Depth_x(v)$ will never increase over a sequence of *Insert* operations for each $v\in V$.

Lemma 3.1: For an arbitrary sequence of at most $O(n^2)$ *Insert* operations

$$\sum_i |C_i(x)| \leq O(n^2)$$

Proof: The proof is based on the sense of depth. If

$w \in C_i(x)$ for the insertion, then $Depth_x(w)$ decreases at least one for this insertion. $Depth_x(w)$ is at most n and at least 1, so $Depth_x(w)$ can decrease at most $n-1$ times. This means that w belongs to $C_i(x)$ at most $n-1$ times over a sequence of *Insert* operations. For the total n vertices, we can have at most $n(n-1)$ depth decreases. Thus we have

$$\sum_i |C_i(x)| \leq O(n^2) \qquad \square$$

Now we want to measure $\Sigma_i |N_i(x)|$, those vertices that are checked (visited) but remained unchanged while *T_Source*(x) is updated over a sequence of *Insert* operations.

In the procedure *Update*(x, w, u, v), we check whether the shortest path from x to v must be changed to the one passing through the edge (u, v) by comparing *Distance*$[x, u]+l$ and *Distance*$(x, v]$. Note that w is the *predecessor* of x and v is a child of u in *T_Source*(w) except in the initial step. Let the shortest path from x to u be $(x, t_1, t_2, \ldots, t_k=u)$ while we execute *update*(x, w, u, v). Obviously, $t_1(=w)$ is the *predecessor* of x. We call the shortest path $(x, t_1, t_2, \ldots, t_k)$ a critical path of length k induced by v if $v \in N_i(x)$ for some i. Indeed, $k=Depth_x(t_k)$. Figure 4 is an example of critical paths.

We have the following properties of critical paths:

Lemma 3.2: If there exists a critical path $(x, t_1, t_2, \ldots, t_k)$ induced by v for $v \in N_i(x)$, then the shortest path from t_1 to v after the ith *Insert* operation is $(t_1, t_2, \ldots, t_k, v)$.

Proof: Because t_1 is the *predecessor* of x, v is a son of t_k in *T_Source*(t_1). Furthermore, the shortest path from t_1 to t_k is $(t_1, t_2, \ldots, t_k)$. $\square$

Lemma 3.3: If there exists a critical path $(x, t_1, t_2, \ldots, t_k)$ induced by v for $v \in N_i(x)$, then one of the following statements must be true for the ith insertion.

(a) The inserted edge is (x, t_1), and $Depth_x(t_j)$ for $j=1 \ldots k$ decreases at least one.

(b) The inserted edge is (t_k, v), and $Depth_{t_j}(v)$ for $j=1 \ldots k$ decreases at least one.

(c) The inserted edge is (t_j, t_{j+1}) for $j=1 \ldots k-1$, $Depth_x(t_b)$, $Depth_{t_f}(t_b)$, and $Depth_{t_f}(v)$ for $f=1, \ldots j$, $b=j+1, \ldots k$ all decrease at least one.

Proof: The inserted edge is either one of the edges in the critical path or the edge (t_k, v). In case (a), $t_j \in C_i(x)$ for $j=1 \ldots k$, or else the vertex pair (x, v) will not be checked for Lemma 2.2. In case (b), $v \in C_i(t_j)$ for $j=1 \ldots k$, or else the vertex pair (x, v) will not be checked. Case (c) is obtained in the same way as (a), (b). $\square$

Lemma 3.4: For each vertex $v \in V$, all the critical paths induced by v for $v \in N_i(x)$, $i=1, 2, \ldots, O(n^2)$, over a sequence of *Insert* operations are different.

Proof: If there exists a critical path $(x, t_1, t_2, \ldots, t_k)$ induced by v twice, then $Depth_x(t_j)$ and $Depth_{t_j}(v)$ for $j=1 \ldots k$ are the same for the two insertions, and violate all the three cases in Lemma 3.3. $\square$

With Lemma 3.4, we can count the number of occurrences of $v \in N_i(x)$ for $i=1, 2, \ldots, O(n^2)$ by counting the number of possible critical paths induced by v over an arbitrary sequence of at most $O(n^2)$ *Insert* operations. We shall prove that the number of these possible critical paths is at most $O(n \log n)$, and get the bound of $O(n^2 \log n)$ for $\Sigma_i |N_i(x)|$.

Lemma 3.5: If $Depth_x(t)$ is the same for a sequence of *Insert* operations, then the shortest path from x to t remains unchanged

Proof: The shortest path from x to t is changed only when *Distance*$[x, t]$ decreases, that is, when $Depth_x(t)$ decreases. $\square$

Lemma 3.6: If two critical paths with equal length induced by v for $v \in N_i(x)$, $i=1, 2, \ldots, O(n^2)$, contain a common vertex t, then $Depth_x(t)$ is the same in these two critical paths.

Proof: Let the two critical paths be $CP_1=(x, t_1, \ldots t_{j_1}=t, \ldots, t_k)$ and $CP_2=(x, s_1, \ldots s_{j_2}=t, \ldots, s_k)$, which contain a common vertex t, with $Depth_x(t)$ equal to j_1 in CP_1 and j_2 in CP_2. By Lemma 3.2, there exist two shortest paths $SP_1=(t_1, \ldots, t, \ldots, t_k, v)$ and $SP_2=(s_1, \ldots, t, \ldots, s_k, v)$. Since $Depth_x(t)$ never increases, if CP_1 is induced before CP_2 during the *Insert* operations sequence, we have $j_1 \geq j_2$. But, using the same reasoning for $Depth_t(v)$, we have $k-j_1+1 \geq k-j_2+1$, i.e., $j_1 \leq j_2$. Hence $j_1=j_2$, and thus $Depth_x(t)$ will be the same. $\square$

Lemma 3.7: The critical paths with equal length induced by v for $v \in N_i(x)$, $i=1, 2, \ldots$, have no common vertex except the root x.

Proof: If two critical paths

$$CP_1=(x, t_1, \ldots, t, \ldots, t_k),$$
$$CP_2=(x, s_1, \ldots, t, \ldots, s_k)$$

of equal length k contain a common vertex t, then from Lemma 3.2, there exists two shortest paths $SP_1=(t_1, \ldots, t, \ldots, t_k, v)$ and $SP_2=(s_1, \ldots, t, \ldots, s_k, v)$. From Lemma 3.6, $Depth_x(t)$ will be the same in CP_1 and CP_2, and thus $Depth_t(v)$ remains unchanged. From Lemma 3.5, $Depth_x(t)$ keeps the same value, which implies that

$$(x, t_1, \ldots, t)=(x, s_1 \ldots t)$$

and $Depth_t(v)$ remains unchanged, which implies that

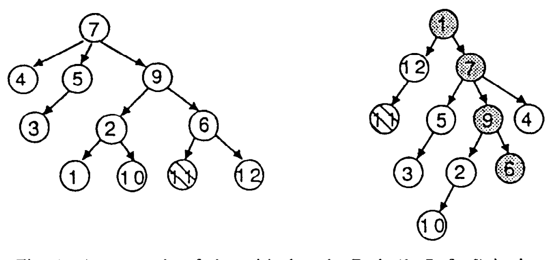

Fig. 4 An example of the critical path. Path (1, 7, 9, 6) is the critical path of length 3 induced by vertex 11 for $11 \in N(1)$ while *Update*(1, 7, 6, 11) is executed.

$(t, \ldots, t_k, v) = (t, \ldots, s_k, v)$.

Hence, $CP_1 = CP_2$, which violates Lemma 3.4. □

Lemma 3.8: The number of possible critical paths induced by v for $v \in N_i(x)$, $i = 1, 2, \ldots$, over an arbitrary sequence of at most $O(n^2)$ *Insert* operations is no more than $O(n \log n)$.

Proof: From Lemma 3.7, for $v \in V$, the maximum number of critical paths with length k induced by v is $\lfloor n/k \rfloor$. Thus, the total number of possible critical paths is

$$\begin{aligned} & n + \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \cdots + 1 \\ & \leq n(1 + 1/2 + 1/3 + \cdots + 1/n) \\ & \leq n \left(1 + \int_1^n 1/x dx\right) \\ & = O(n \log n) \end{aligned}$$

□

Theorem 3.2: For an arbitrary sequence of at most $O(n^2)$ *Insert* operations, $\Sigma_i |N_i(x)| \leq O(n^2 \log n)$.

Proof: From Lemma 3.8, the number of occurrences of $v \in N_i(x)$ for $i = 1, 2, \ldots$ is at most $O(n \log n)$ for every vertex $v \in V$. Hence, $\Sigma_i |N_i(x)| \leq O(n^2 \log n)$, since V has at most n vertices. □

From Lemma 3.1 and Theorem 3.2, we know that for some Source trees, say Source(x), the numbers of steps it takes for an arbitrary sequence of Insert operations (i.e. $\Sigma_i |T_i(x)|$) is $O(n^2 + n^2 \log n) = O(n^2 \log n)$. Hence, for the total number n of Source trees, it takes at most $O(n^3 \log n)$ time. This leads to the following theorem:

Theorem 3.3: An arbitrary sequence of at most $O(n^2)$ *Insert* operations takes no more than $O(n^3 \log n)$ time.

4. Remarks and open Problems

In this paper we discussed the *dynamic shortest-path problem*, which searches for and reports the shortest path between two vertices on a directed graph with n vertices while edges of equal lengths are inserted one by one. We have proposed efficient algorithms by presenting a data structure that supports each *FindShortest* operation in $O(k)$ time, where $k \leq n$ is the number of edges of the reported path, and an arbitrary sequence of at most $O(n^2)$ *Insert* operations in a worst-case time of $O(n^3 \log n)$. The structure requires $O(n^2)$ space. Furthermore, our *Insert* algorithm can be extended to perform the cost-decreasing operation of the least-cost path problem [8] and always takes a time less than or equal to that in [8].

Our result may be generalizable in various directions. first, the question of whether there is an algorithm that takes less than $O(n^2)$ worst-case time per operation remains open. Furthermore, the operations of edges deletion are not considered in this paper. Both the case in which only edge deletions are allowed and that in which both edges insertions and edge deletions are allowed deserve further study.

Finally, this problem can be extended to graphs whose edges have arbitrary length without negative distance cycles, such as the least-case path problem [8]. In contrast to the case in which the edges are of equal length, the distance of a shortest path between two vertices can have at most n values, namely, n, $n-1, \ldots,$ 1. In this case however, the distance of a shortest path may be any value. Hence, the analysis in Section 3 is no longer valid.

References

1. AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. The Design and Analysis of Computer Algorithms, Reading, Mass.; Addison-Wesley, 1974.
2. EVEN, S. and SHILOACH, Y. An on-line edge deletion problem, *J. ACM*, **28** (1981), 1-4.
3. FREDERICKSON, G. N. Data structures for on-line updating of minimum spanning trees with applications, *SIAM J. Comput.* **14** (1985), 781-798.
4. HAREL, D. On-line maintenance of the connected components of dynamic graphs, Unpublished manuscript, 1982.
5. IBARAKI, T. and KATOH, N. On-line computation of transitive closures for graphs, *Inf. Process. Lett.* **16** (1983), 95-97.
6. ITALIANO, G. F. Amortized efficiency of a path retrieval data structure, *Theor. Comput. Sci.*, **48** (1986), 273-281.
7. ITALIANO, G. F. Finding paths and deleting edges in directed acyclic graphs, **Inf. Process. Lett. 28** (1988), 5-11.
8. ROHNERT, H. A dynamization of the all-pairs least-cost path problem, Proc. 2nd Ann. Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, **182** (Springer, Berlin, 1985), 279-286.
9. SLEATOR, D. D. and TARJAN, R. E. A data structure for dynamic trees, *J. Comput. Sys. Sci.*, **24** (1983), 362-381.
10. SLEATOR, D. D. and TARJAN, R. E. Self-adjusting binary search trees, *J. ACM.*, **32** (1985), 652-686.
11. SLEATOR, D. D. and TARJAN, R. E. Self-adjusting heaps, *SIAM J. Comput.*, **15** (1986), 52-69.
12. TARJAN, R. E. Efficiency of a good but not linear set union algorithm, *J. ACM.*, **22** (1975), 215-225.
13. TARJAN, R. E. and LEEUWEN, J. VAN. Worst-case analysis of set union algorithms, *J. ACM.*, **31** (1984), 245-281.
14. TARJAN, R. E. Data Structures and Network Algorithms, CBMS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
15. TARJAN, R. E. Amortized computational complexity, *SIAM J. Alg. Disc. Meth.*, **6** (1985), 306-318.
16. TSAKALIDIS, A. K. The nearest common ancestor in a dynamic tree, *Acta Inf.*, **25** (1988), 37-54.

(Received March 5, 1990; revised June 21, 1990)