ELSEVIER

Information Processing Letters 58 (1996) 149-155

Information Processing Letters

NC algorithms for dynamically solving the all pairs shortest paths problem and related problems

Weifa Liang [a,*], Brendan D. McKay [a,1], Hong Shen [b,2]

[a] *Department of Computer Science, Australian National University, Canberra, ACT 0200, Australia*

[b] *School of Computing and Information Technology, Griffith University, Nathan, QLD 4111, Australia*

Received 23 March 1995

Communicated by K. Ikeda

Abstract

We consider the problem of dynamically maintaining a solution of all pairs shortest paths in a directed weighted graph $G = (V, E)$ undergoing a sequence of edge insertions and/or the edge cost decreases, and present a simple data structure to support the above operations. The proposed algorithm for maintaining the data structure requires $O(\log n)$ time and $O(n^2)$ processors for each of the operations above. Furthermore, our algorithm is able to answer n^2 queries concerning the lengths of all pairs shortest paths in $O(1)$ time, and to find n shortest paths in $O(\log n)$ time. The same bounds for similar operations can be achieved for other problems such as dynamically maintaining all pairs longest paths in a directed acyclic graph (DAG), the topological order of a DAG, and the transitive closure of a directed graph. To the best of our knowledge, no partially dynamic NC algorithm using $O(n^2)$ processors for these problems is known yet, despite the existence of several efficient sequential algorithms. All known NC algorithms for these problems are based on matrix multiplication which requires $M(n)$ processors at least. Currently the best result for $M(n)$ is $n^{2.376}$ (Coppersmith and Winograd, 1987). Unless otherwise specified, our computational model is a CREW PRAM in which concurrent read to the same memory cell is allowed, but concurrent write to the same memory cell is forbidden.

Keywords: Parallel algorithms; Partially dynamic graph algorithms; The all pairs shortest paths problem; The longest path problem; The transitive closure problem; Topological sorting

1. Introduction

A *fully dynamic* graph algorithm is one that allows edge insertions and deletions, and recomputes some desired graph property quickly after each such update. A *partially dynamic* graph algorithm is one that allows either edge insertions or edge deletions but not both. In this paper we consider the edge insertion case only. Let $G(V, E)$ be a weighted or unweighted directed (acyclic) graph. Initially, $G(V, E) = G(V, \emptyset)$ and $V = \{1, 2, \ldots, n\}$. An important problem in the algorithm area is how to maintain *on-line* all pairs shortest paths in G after undergoing dynamic updates. The problem consists of (1) maintaining a distance matrix *Dist*, where an entry $Dist[i, j]$ in *Dist* contains the length of the shortest path from vertex i to vertex j; (2) finding such paths and answering queries about the lengths of such paths for the edge insertions, $1 \leqslant$

* Corresponding author. Email: wliang@cs.anu.edu.au.

[1] Email: bdm@cs.anu.edu.au.

[2] Email: hong@cit.gu.edu.au.

PII S0020-0190(96)00037-3

$i, j \leqslant n$. In detail, in this paper we consider how to maintain all pairs shortest paths in G in parallel after undergoing an intermixed sequence of the following operations:

- *Add*(x, y, w): insert an edge of cost w from vertex x to vertex y;
- *LengthShortestPath*(x, y): return the length of the shortest path from x to y if it exists; $+\infty$ otherwise;
- *FindShortestPath*(x, y): return the shortest path from x to y if it exists.

The problem above has been studied in the sequential context previously [1,9]. Ausiello et al. considered a special case of this problem in which case the edge cost is an integer bounded by C. For this case, their algorithm requires $O(Cn^3 \log n)$ amortized time for an intermixed sequence of edge insertions and edge cost decreases, $O(1)$ time for a query *LengthShortestPath*(x, y) and $O(l)$ time for *FindShortestPath*(x, y), where l is the number of edges of the shortest path from x to y. Independently Lin and Chang [9] also gave the same results for this special case. As for the problem of fully dynamic maintenance of all pairs shortest paths in a directed graph, i.e., maintaining a solution to the all pairs shortest paths problem during edge insertions, edge deletions and edge cost updates, Even and Gazit [4] have shown that it requires $O(n^2)$ time for an edge insertion and/or an edge cost decrease, and $O(mn + n^2 \log n)$ time for an edge deletion and/or an edge cost increase respectively, where m is the number of edges in the current graph. Rohnert [11] and Yellin [14] also considered the fully dynamic maintenance of this problem, giving similar results. However, to the best of our knowledge, there is no known NC algorithm for this problem using $O(n^2)$ processors only, in spite of several efficient sequential algorithms for it. Without exception, all existing NC algorithms for this problem are based on matrix multiplication which requires $M(n)$ processors at least. Currently the best result for $M(n)$ is $n^{2.376}$ [3]. In this paper we suggest an NC algorithm for this problem. Our algorithm requires

(i) $O(\log n)$ time and $O(n^2)$ processors for an edge insertion and/or an edge cost decrease;
(ii) $O(1)$ time and $O(n^2)$ processors for the queries of the lengths of n^2 pairs shortest paths; and
(iii) $O(\log n)$ time and $O(n^2)$ processors for finding n shortest paths.

The proposed NC algorithm for the partially dynamic maintenance of the all pairs shortest paths problem can also be extended to other dynamic update problems such as the all pairs longest paths problem in a directed acyclic graph (DAG), the topological sorting problem on a DAG, and the transitive closure problem in a directed graph. The all pairs longest paths problem is polynomially solvable on DAGs, but is NP-complete on general graphs [5]. Ausiello et al. [1] extended their results on all pairs shortest paths to all pairs longest paths in a DAG, and presented an algorithm. Their algorithm for this latter problem has the same time complexity as that for the former problem.

The topological sorting problem and the transitive closure problem have wide applications in distributed computing and compilers [14,12]. For the on-line topological sorting problem which was raised in complier, Spaccamela et al. [12] proposed an amortized $O(nm)$ time algorithm for m edge insertions. Note that there exists a topological ordering of the vertices in a directed graph only if the graph is a *DAG*. For the transitive closure problem, there are several sequential algorithms for it [6,8,7]. The first fully dynamic algorithm was given by Ibaraki and Katoh [6]. Their algorithm requires $O(n^3)$ time for a sequence of edge insertions, and $O(n^2(m + n))$ time for m edge deletions. Later La Poutré and van Leeuwen [8] considered a special case of this problem in which case a directed graph can be augmented to a DAG. For this case, their algorithm requires $O(nm_t)$ amortized time for either m edge insertions or m edge deletions but not for both of them, where m_t is either the number of edges in the original graph for deletion or the number of edges in the resulting graph for insertion respectively. Independently Italiano [7] also presented an algorithm for the DAG with the same time bound. If the maximum vertex degree of the augmented graph is bounded by Δ, for this special case Yellin [14] proposed an algorithm which is still the best known. His algorithm requires $O(m\Delta)$ amortized time where m is either the number of edges of the transitive closure in the final graph for edge insertions or the number of edges in the initial graph for edge deletions.

We extend our results for the all pairs shortest paths problem to all other problems mentioned above, and solve them using similar techniques. In detail, our algorithms for these problems require $O(\log n)$ time and $O(n^2)$ processors for an edge insertion, and $O(\log n)$

time and $O(n^2)$ processors for n path queries such as finding the longest paths in a DAG, finding n directed paths in the transitive closure problem etc. Unless specified, our computational model is a CREW PRAM in which concurrent read and exclusive write are allowed.

2. The algorithm for the all pairs shortest paths problem

The data structure for this problem comprises two matrices *Dist* and *PT*. An entry $Dist[i,j]$ in *Dist* is the length of the shortest path from vertex i to vertex j if such a path exists, $+\infty$ otherwise. Consider *the single source shortest path problem* starting from i. It consists of computing the shortest paths from i to all other vertices. Denote by $T(i)$, a shortest path tree rooted at i, where the parent $F_{T(i)}(j)$ of vertex j is j's immediate ancestor on the shortest path from i to j if j is reachable from i, $F_{T(i)}(j)$ is assigned *NULL* otherwise. An entry $PT[i,j]$ in *PT* is defined as $PT[i,j] := F_{T(i)}(j)$.

Having the above data structure, we now consider inserting an edge (x,y) into the graph G. Let $Dist_{\text{old}}[i,j]$ and $Dist_{\text{new}}[i,j]$ be the length of the shortest path from i to j before and after inserting the edge (x,y) respectively. Then, by the definition of the shortest paths, for all $i \in V$ and $j \in V$, we have

$$Dist_{\text{new}}[i,j] := \min\{Dist_{\text{old}}[i,j], Dist_{\text{old}}[i,x] + Dist_{\text{old}}[y,j] + w(x,y)\}, \quad (1)$$

where $w(x,y)$ represents the cost of the edge (x,y).

Using Eq. (1), in the following we give a simple dynamic update algorithm for the all pairs shortest paths problem.

Consider inserting an edge (x,y) into G. Our algorithm proceeds as follows. For each $T(i)$ we first assigned a copy $T_{i,y}$ of tree $T(y)$ to it. Then for each tree $T(i)$ and every vertex $j \in V$, we check whether

$$Dist_{\text{old}}[i,j] > Dist_{\text{old}}[i,x] + Dist_{\text{old}}[y,j] + w(x,y).$$

If yes, we assign $Dist[i,j]$ by

$$Dist_{\text{old}}[i,x] + Dist_{\text{old}}[y,j] + w(x,y),$$

and delete the edge $(j, F_{T(i)}(j))$ from $T(i)$, i.e., set $F_{T(i)}(j) := j$. Otherwise we delete the edge $(j, F_{T_{i,y}}(j))$ from the tree $T_{i,y}$. Let $T'(i)$ and $T'_{i,y}$ be the resulting graphs after deleting the edges from $T(i)$ and $T_{i,y}$ respectively. We now compute the connected components of graphs $T'(i)$ and $T'_{i,y}$. Denote by $CC_G(q)$ the connected component containing vertex q in a graph G. Let $F_T^{\text{old}}(i)$ and $F_T^{\text{new}}(i)$ be the parent of vertex i in a tree T before and after inserting the edge (x,y) respectively. Now we check whether $Dist_{\text{old}}[i,y] > Dist_{\text{new}}[i,y]$. If yes, vertex x is the parent of y in $T(i)$ after inserting edge (x,y), and we set $F^{\text{new}}_{T(i)}(y) := x$. Finally we merge two connected components $CC_{T'(i)}(i)$ and $CC_{T'_{i,y}}(y)$ (here actually two subtrees) into a connected component, i.e., we update the tree $T(i)$. The description for this update is as follows. Initially we set $F^{\text{new}}_{T(i)}(v) := NULL$ for all $v \in V$. Then for each vertex $k \in CC_{T'(i)}(i)$, we set $F^{\text{new}}_{T(i)}(k) := F^{\text{old}}_{T(i)}(k)$, and for each $k \in CC_{T'_{i,y}}(y)$ we set $F^{\text{new}}_{T(i)}(k) := F_{T_{i,y}}(k)$.

The algorithm above can be easily modified to process the case of edge cost decrease case. We omit it here. In the following we show that the proposed algorithm is correct.

Lemma 1. *For any vertex $k \neq y$ in $T(y)$, $k \in CC_{T'_{i,y}}(y)$ if and only if*

$$Dist_{\text{old}}[i,k] > Dist_{\text{old}}[i,x] + Dist_{\text{old}}[y,k] + w(x,y). \quad (2)$$

Proof. For any $k \in V$ which is reachable from y, if k does not satisfy Inequality (2), the edge $(k, F_{T_{i,y}}(k))$ will be deleted from $T_{i,y}$, and hence k is excluded from $CC_{T'_{i,y}}$. So the vertices remaining in $CC_{T'_{i,y}}$ must satisfy Inequality (2).

Let k be such a vertex that $Dist_{\text{old}}[i,k] > Dist_{\text{old}}[i,x] + Dist_{\text{old}}[y,k] + w(x,y) = Dist_{\text{new}}[i,k]$. In order to prove $k \in CC_{T'_{i,y}}(y)$, we just need to show that, for every ancestor p of k in $T(y)$,

$$Dist_{\text{old}}[i,p] > Dist_{\text{old}}[i,x] + Dist_{\text{old}}[y,p] + w(x,y). \quad (3)$$

We prove this by contradiction. Assume that vertex p_1 is the first ancestor vertex (starting from k) of k which does not satisfy Inequality (3). Then

$$Dist_{\text{old}}[i, p_1] \leqslant Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, p_1] + w(x, y).$$

However, we already knew that

$$\begin{aligned} Dist_{\text{old}}[i, k] &\leqslant Dist_{\text{old}}[i, p_1] + Dist_{\text{old}}[p_1, k] \\ &\leqslant (Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, p_1] + w(x, y)) \\ &\quad + Dist_{\text{old}}[p_1, k] \\ &= Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, k] + w(x, y) \\ &= Dist_{\text{new}}[i, k]. \end{aligned}$$

Contradiction! So, for every ancestor p of k in $T(y)$, we have $Dist_{\text{old}}[i, p] > Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, p] + w(x, y)$. From this we derive that the vertices k and y are in the same connected component of $T'_{i,y}$, i.e., $k \in CC_{T'_{i,y}}(y)$. □

Using the same argument we have the following lemma.

Lemma 2. *For any vertex $k \neq i$ in $T(i)$, $k \in CC_{T'(i)}(i)$ if and only if*

$$Dist_{\text{old}}[i, k] \leqslant Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, k] + w(x, y). \quad (4)$$

Lemma 3. *The proposed algorithm is correct.*

Proof. Let k be a vertex in $CC_{T'(i)}(i)$. By Lemma 2, we know that $Dist_{\text{old}}[i, k] \leqslant Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, k] + w(x, y)$. So, the edge (x, y) added into G does not affect the shortest path from i to k. If y already changed its parent to x in $T(i)$ and k in $CC_{T'_{i,y}}(y)$, by Lemma 1, $Dist_{\text{old}}[i, k] > Dist_{\text{old}}[i, x] + Dist_{\text{old}}[y, k] + w(x, y)$. This means that the shortest path from i to k must pass through the edge (x, y). The parent of k in $T(y)$ therefore is also the parent of k in $T(i)$ after inserting the edge (x, y). □

Lemma 4. *The update algorithm for inserting an edge (x, y) into G requires $O(\log n)$ time and $O(n^2)$ processors.*

Proof. Note that the computation of connected components in a forest with n vertices can be done in $O(\log n)$ time using $O(n)$ processors. Therefore, the entire update for an edge insertion requires $O(\log n)$ time and $O(n^2)$ processors because there are n shortest path trees. For each such a tree $T(i)$, it may need to merge with a subtree of $T(y)$ including y. The merge needs $O(\log n)$ time and $O(n)$ processors, $1 \leqslant i \leqslant n$. So, the algorithm for maintaining all pairs shortest paths requires $O(\log n)$ time and total $O(n^2)$ processors. □

Now consider operation *LengthShortestPath*(u, v). It can be handled easily by retrieving the value of the entry $Dist(u, v)$. So, we are able to answer as many as n^2 such queries in $O(1)$ time because there are n^2 processors available. The query for finding the shortest path from u to v can be done by finding the directed path from v to the root u in the tree $T(u)$ which can be obtained easily in $O(\log n)$ time using $O(n)$ processors by *path doubling technique* in parallel processing. So, we are able to find n shortest paths in $O(\log n)$ time. Therefore we have the following theorem.

Theorem 5. *Given a weighted directed graph $G(V, E)$ provided that the edge insertions introduce no negative cycles in G, there exists a data structure which supports n^2 LengthShortestPath(x, y) operations in $O(1)$ time, n FindShortestPath(x, y) operations in $O(\log n)$ time, and an Add(x, y, w) operation in $O(\log n)$ time, using $O(n^2)$ processors at most.*

3. The algorithms for other problems

3.1. Finding all pairs longest paths in a DAG

Following the similar discussion by Ausiello et al. in [1], we can modify the algorithm in Section 2 to make it capable of performing the following operations on a DAG. Assume that the augmented graph is still a DAG.

- *Add*(x, y, w): insert an edge of cost w from vertex x to y;
- *LengthLongestPath*(x, y): return the length of the longest path from x to y if it exists; $+\infty$ otherwise;
- *FindLongestPath*(x, y): return the longest path from x to y if it exists.

The data structure for this problem is a modification of that for the all pairs shortest paths problem. That is,

for each $i \in V$ we establish a directed tree $T(i)$ rooted at vertex i. Define the parent $F_{T(i)}(j)$ of a vertex j in $T(i)$ by u, where u is the immediate ancestor of j on the longest path from i to j. The tree $T(i)$ is stored in the ith row of the matrix PT defined in Section 2. Meanwhile we also establish a matrix $Long$ in which an entry $Long[i,j]$ represents the length of the longest path from i to j if it does exist, $Long[i,j]$ is $-\infty$ otherwise.

When inserting an edge (x,y), we first check whether it introduces a cycle, which can be done easily by testing whether $Long[y,x] \neq -\infty$. If yes, the edge (x,y) cannot be added to G, because it will form a cycle in G consisting of the longest path from y to x and the edge (x,y) itself. Otherwise we should re-compute some longest paths after inserting (x,y). Assume that there is no cycle introduced by adding (x,y) into the current graph. Let $Long_{old}[i,j]$ and $Long_{new}[i,j]$ be the length of the longest path from i to j before and after inserting (x,y) respectively. Then by the definition of the longest paths, for all $i \in V$ and $j \in V$, we have

$$Long_{old}[i,i] := Long_{new}[i,i] := 0, \tag{5}$$

$$Long_{new}[i,j] := \max\{Long_{old}[i,j], Long_{old}[i,x] + Long_{old}[y,j] + w(x,y)\}. \tag{6}$$

The processing for this problem is similar to that for the all pairs shortest paths problem. Therefore we have

Theorem 6. *Given a DAG $G(V,E)$ provided that the edge insertions introduce no cycles, there exists a data structures which supports n^2 LengthLongestPath(x,y) operations in $O(1)$ time, n FindLongestPath(x,y) operations in $O(\log n)$ time, and an Add(x,y,w) operation in $O(\log n)$ time. The number of processors used is $O(n^2)$.*

3.2. Topological sorting in a DAG

A topological order of a partially ordered set (V,Ω) is a total order ord on V such that if $(x,y) \in \Omega$, then $ord(x) < ord(y)$. It is straightforward to represent a partially ordered set (V,Ω) by using a DAG $G(V,E)$ such that $(x,y) \in E$ if and only if $(x,y) \in \Omega$. A topological order ord of vertices in G is to map each vertex v to an integer $ord(v)$ between 1 and n such that, for each $(x,y) \in E$, $ord(x) < ord(y)$. The integer $ord(u)$ is called the topological numbering of vertex u. A simple sequential algorithm for this problem requires $O(m+n)$ time by using *depth-first searching* technique [10]. But all known parallel algorithms for this problem [2,13] are based on matrix multiplication which require $O(\log^2 n)$ time and $M(n)$ processors at least on a CREW PRAM.

In this paper we deal with how to maintain the topological numbering of vertices dynamically after a sequence of edge insertions operations. The data structure for this problem include two matrices $Dist$ and PT, and two one-dimensional arrays ord and Vex, where an entry $Dist[i,j]$ in $Dist$ is the length of the shortest path from i to j if j is reachable from i, $Dist[i,j] := +\infty$ otherwise. An entry $ord(i)$ in ord represents vertex i's topological numbering, and an entry $Vex(i)$ in Vex represents the vertex whose topological numbering is i. For each vertex $i \in V$ we build a reachability tree $T(i)$ which actually is a shortest path tree starting at i. The tree $T(i)$ is stored in the ith row of the matrix PT.

Consider inserting an edge (x,y) to a DAG G. First we check whether the augmented graph is still a DAG, i.e., whether this operation introduces any cycle in G. This can be done by checking if $Dist[y,x] \neq +\infty$. If $Dist[y,x] \neq +\infty$, we cannot add this edge to G, because otherwise a directed cycle will be introduced. From now on, assume that inserting an edge (x,y) introduces no cycles. We proceed as follows. We first check whether the old topological order is still valid, i.e., we check whether $ord(x) < ord(y)$. If so, we just update the matrices $Dist$ and PT by the algorithm in Section 2, and do nothing about the topological order update. Otherwise we must update ord and Vex too, i.e., recompute the topological numbering of vertices. Our algorithm for this latter case depends on the following lemma proven by Spaccamela et al.

Lemma 7 (see [12]). *Given a DAG $G(V,E)$ and a topological order ord of its vertices, let $G'(V,E \cup \{(x,y)\})$ be the graph obtained from G by inserting edge (x,y). If G' is acyclic, then there is a topological order ord_{new} of G' such that $ord_{new}(w) = ord_{old}(w)$ for all w with $ord_{old}(w) < ord_{old}(x)$ or $ord_{old}(w) > ord_{old}(y)$, where $ord_{old}(w)$ is the topological numbering of vertex w before inserting the edge (x,y).*

Let $X = \{x_1, x_2, \ldots, x_p\}$ be a vertex set in which x is reachable from each vertex in X and $ord(y) \leqslant ord(x_i) \leqslant ord(x)$ for $1 \leqslant i \leqslant p$, where $ord(x_i) < ord(x_{i+1})$ for all $i \leqslant p-1$. The construction of X can be done in $O(\log n)$ time using $O(n)$ processors by testing each u to see if $Dist(u,x) \neq +\infty$ and $ord(y) \leqslant ord(u) \leqslant ord(x)$ and then by sorting.

Let $Y = \{y_1, y_2, \ldots, y_q\}$ be a vertex set in which each vertex is reachable from y and $ord(y) \leqslant ord(y_j) \leqslant ord(x)$ for $1 \leqslant j \leqslant q$, where $ord(y_j) < ord(y_{j+1})$ for all $j \leqslant q-1$. Obviously Y can be obtained by checking each vertex u in $T(y)$ if it satisfies $ord(y) \leqslant ord(u) \leqslant ord(x)$.

It is easy to show that $X \cap Y = \emptyset$ when $X \neq \{x\}$ and $Y \neq \{y\}$, since otherwise the augmented graph is not a DAG any more. Let A and B be two disjoint vertex sets. Define $ord(A) < ord(B)$ if $ord(u) < ord(v)$ for every $u \in A$ and every $v \in B$.

From Lemma 7, we know that the topological numbering $ord(u)$ of vertex u may need to be updated if $ord(y) \leqslant ord(u) \leqslant ord(x)$. Let $U = \{u \mid ord(y) \leqslant ord(u) \leqslant ord(x)\}$. Obviously $X \subset U$ and $Y \subset U$. Let $W = U - Y$. Then $x \in W$ and $y \in Y$. In order to compute the new topological order after inserting the edge (x,y), it is necessary that each of the following conditions must be satisfied:

(i) $ord_{new}(x) < ord_{new}(y)$ and $ord_{new}(W) < ord_{new}(Y)$ because $ord_{old}(x) = \max\{ord_{old}(v) \mid v \in W\}$, $ord_{old}(y) = \min\{ord_{old}(u) \mid u \in Y\}$ and $ord_{old}(x) > ord_{old}(y)$;

(ii) for any two vertices $u, v \in W$, $ord_{new}(u) < ord_{new}(v)$ if $ord_{old}(u) < ord_{old}(v)$;

(iii) for any two vertices $u, v \in Y$, $ord_{new}(u) < ord_{new}(v)$ if $ord_{old}(u) < ord_{old}(v)$.

Based on the discussion above we now update the topological numbering of vertices in U. We proceed as follows. First mark all vertices in Y. Then define a 1-dimensional array *count* in which the entry $count(i) := 1$ if $Vex(i)$ is marked; $count(i) := 0$ otherwise. Then compute the prefix $sum(i) := \sum_{j=1}^{i} count(j)$ of array *count* for all $1 \leqslant i \leqslant n$. Finally we assign new topological numberings to all vertices. For each $u \in V-U$, $ord_{new}(u) := ord_{old}(u)$ and $Vex_{new}(ord_{old}(u)) := u$. Let $l = ord_{old}(y)$, $h = ord_{old}(x)$ and $k = \max\{sum(i) \mid 1 \leqslant i \leqslant n\}$. For each i between l and h, if $Vex(i) \in Y$ then $Vex_{new}(h - k + sum(i)) := Vex_{old}(i)$ and $ord_{new}(Vex_{old}(i)) := h - k + sum(i)$; if $Vex(i) \in W$ then $Vex_{new}(i - sum(i)) := Vex_{old}(i)$ and $ord_{new}(Vex_{old}(i)) := i - sum(i)$.

It is not difficult to verify that the newtopological numbering of vertices satisfies the above conditions (i)–(iii). The computation of all prefixes of *count* can be done in $O(\log n)$ time using $O(n/\log n)$ processors. Thus we have the following theorem.

Theorem 8. *Given a DAG $G(V,E)$ provided that the edge insertions introduce no cycles, there exists a data structure which supports maintaining the topological order of vertices in G when there is an edge insertion. The proposed algorithm requires $O(\log n)$ time and $O(n^2)$ processors.*

3.3. The dynamic transitive closure problem

The problem is to compute the transitive closure of the augmented graph dynamically undergoing an intermixed sequence of the following operations:

- *Add*(x,y): insert an edge from vertex x to y;
- *Reachability*(x,y): return TRUE if y is reachable from x, return FALSE otherwise;
- *ReachabilityPath*(x,y): return a directed path from x to y if such one exists.

By a similar discussion as in the previous section, we maintain two matrices *reach* and *PT*, where an entry *reach*$[i,j]$ in *reach* is TRUE if vertex j is reachable from the vertex i, and FALSE otherwise. The ith row of *PT* stores a directed rooted tree $T(i)$, where $T(i)$ is the shortest path tree from i to all other vertices, and these vertices are reachable from i. Note that the cost of each edge here is one. So, the transitive closure problem is a special case of the all pairs shortest paths problem. Thus we have

Theorem 9. *Given a directed graph $G(V,E)$, there exists a data structure which supports n^2 Reachability(x,y) operations in $O(1)$ time, n ReachabilityPath(x,y) operations in $O(\log n)$ time, and an Add(x,y) operation in $O(\log n)$ time. The number of processors used is $O(n^2)$.*

Remark. The parallel algorithms described above require the same time for processing a single query and multiple queries. It should be noticed that it is more efficient for parallel dynamic algorithms to process a batch of queries rather than each query individually. The possibility for processing multiple queries simul-

taneously is provided by the fact that queries only retrieve information from the given data structures but do not modify the underlying data structures.

4. Conclusions

In this paper we discuss the partially dynamic update problem related to directed (acyclic) graphs such as the all pairs shortest paths problem, the all pairs longest paths problem, the topological sorting problem, and the transitive closure problem. Our algorithm requires $O(\log n)$ time for an edge insertion, $O(1)$ time for n^2 queries of graph's simple property such as the length of the shortest (longest) paths (in a DAG) and the reachability between a pair of vertices, and $O(\log n)$ time for both finding n pairs shortest (longest) paths and maintaining the topological order of vertices. The number of processors used is at most $O(n^2)$. The open problem is whether there is any NC algorithm for the problems above by using $O(n^2)$ processors or $O(nm)$ processors for *edge deletions* only.

References

[1] G. Ausiello, G.F. Italiano, A.M. Spaccamela and U. Nanni, Incremental algorithms for minimal length paths, *J. Algorithms* **12** (1991) 615–638.

[2] P. Chandhui and R.K. Ghosh, Parallel algorithms for analyzing activity networks, *BIT* **26** (1986) 418–429.

[3] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, in: *Proc. 19th ACM Symp. on Theory of Computing* (1987) 418–429.

[4] S. Even and H. Gazit, Updating distances in dynamic graphs, *Methods Oper. Res.* **49** (1985) 371–387.

[5] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W.H. Freeman, New York, 1979).

[6] T. Ibaraki and N. Katoh, On-line computation of transitive closure for graphs, *Inform. Process. Lett.* **16** (1983) 371–387.

[7] G.F. Italiano, Finding paths and deleting edges in directed acyclic graphs, *Inform. Process. Lett.* **28** (1988) 371–387.

[8] J.A. La Poutré and J. van Leeuwen, Maintenance of transitive closure and transitive reduction of graphs, in: *Proc. Internat. Workshop on Graph-Theoretic Concepts Computer Science*, Lecture Notes in Computer Science **314** (Springer, Berlin, 1988) 106–120.

[9] C.C. Lin and R.C. Chang, On the dynamic shortest path problem, *J. Inform. Process.* **13** (1990) 470–476.

[10] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness* (Springer, Berlin, 1984).

[11] H. Rohnert, A dynamization of the all pairs cost path problem, in: *Proc. 2nd Ann. Symp. on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **182** (Springer, Berlin, 1985) 279–286.

[12] A.M. Spaccamela, U. Nanni and H. Rohnert, On-line graph algorithms for incremental compilation, in: *Proc. 19th Internat. Workshop on Graph-Theoretic Concepts Computer Science*, Lecture Notes in Computer Science **790** (Springer, Berlin, 1993) 70–86.

[13] C. Tang and W. Liang, Parallel computation for AOE problems, *Math. Numerica Sinica* **13** (1991) 112–120.

[14] D.M. Yellin, Speeding up dynamic transitive closure for bounded degree graphs, *Acta Inform.* **30** (1993) 369–384.