A Fully Dynamic Algorithm for Maintaining the Transitive Closure

Valerie King[1] and Garry Sagert[2]
Department of Computer Science, University of Victoria, Victoria, BC, Canada

E-mail: val@csr.uvic.ca; gsagert@alumni.uvic.ca

This paper presents an efficient fully dynamic graph algorithm for maintaining the transitive closure of a directed graph. The algorithm updates the adjacency matrix of the transitive closure with each update to the graph, hence, each reachability query of the form "Is there a directed path from i to j?" can be answered in $O(1)$ time. The algorithm is randomized and has a one-sided error; it is correct when answering yes, but has $O(1/n^c)$ probability of error when answering no, for any constant c. In acyclic graphs, worst case update time is $O(n^2)$. In general graphs, the update time is $O(n^{2+\alpha})$, where $\alpha = \min\{.26$, maximum size of a strongly connected component$\}$. The space complexity of the algorithm is $O(n^2)$.

---

1. INTRODUCTION

This paper presents a fully dynamic graph algorithm for maintaining the transitive closure of a directed graph. A *fully dynamic graph algorithm* is a data structure for a graph which implements an on-line sequence of update operations that insert and delete edges in the graph, and answers queries about a given property of the graph. A dynamic algorithm should process queries quickly, and must perform update operations faster than computing the graph property from scratch (as performed by the fastest "static" algorithm). A graph algorithm is said to be *partially* dynamic if it allows only insertions or only deletions.

Researchers have been studying dynamic graph problems for over twenty years. The study of dynamic graph problems on undirected graphs has met with much success, for a number of graph properties. For a survey of the earlier work, see [5]. For more recent work, see [8, 7, 9]. Directed graph problems have proven to be much tougher and very little is known, especially for fully dynamic graph algorithms. Yet maintaining the transitive closure of a changing directed graph is

[1] Work partially done at the University of Copenhagen and Hebrew University.
[2] Work partially done at the University of Copenhagen.

a fundamental problem, with applications to such areas as compilers, data bases, and garbage collection.

This paper offers a breakthrough in the area of fully dynamic algorithms for directed graphs, using a surprisingly simple technique. The algorithm presented here is the first to process reachability queries quickly, indeed, to maintain the adjacency matrix of the transitive closure after each update, while assuming no lookahead, i.e., no knowledge about future updates. Unlike other dynamic graph algorithms, in one update operation, it can insert an arbitrary set of edges incident to a common vertex (in acyclic graphs, or graphs with strongly connected components containing less than $n^{2.26}$ vertices) and delete an arbitrary set of edges incident to a common vertex (in general graphs). In addition, unlike other algorithms, in acyclic graphs, it can answer *sensitivity queries* of the form "Is there a path from i to j not containing edge e?" in $O(1)$ time.

Let $|SCC|$ denote the number of vertices in the largest strongly connected component in the graph, and let E_v denote an arbitrary set of edges incident to a common vertex v. For graphs containing n vertices, our results are as follows:

1. For acyclic graphs:

   - Update: Insert or delete an edge set E_v in $O(n^2)$ worst case time.
   - Reachability query: "Can vertex i reach vertex j?" in $O(1)$ time.
   - Sensitivity query: "Is there a path from i to j not containing edge e?" in $O(1)$ time.

2. For graphs such that $|SCC| \leq n^{\varepsilon}$, for $\varepsilon \in 0 \ldots 1$:

   - Update: Insert or delete an edge set E_v in $O(n^{2+\varepsilon})$ worst case time.
   - Reachability query: "Can vertex i reach vertex j?" in $O(1)$ time.

3. For general graphs:

   - Update: Insert an edge, or delete an edge set E_v in $O(n^{2.26})$ amortized time.
   - Reachability query: "Can vertex i reach vertex j?" in $O(1)$ time.

Algorithm (3.) uses a subroutine which computes the product of two rectangular matrices. Our update time depends on the method by which the product is computed. If square matrix multiplication is used, the amortized cost of executing updates on a graph is $O(n^{2+\frac{\omega-2}{\omega-1}})$, where n^{ω} is the cost of multiplying two $n \times n$ matrices. Note that as long as $\omega > 2$, the cost per update is less than the cost of multiplying two $n \times n$ matrices. For $\omega = 3$ (simple matrix multiplication), the update time is $O(n^{2.5})$; for $\omega = 2.81$ (Strassen [17]), the update time is $O(n^{2.45})$; for $\omega = n^{2.38}$ (Coppersmith and Winograd [2]), the update time is $O(n^{2.28})$. If the fast rectangular matrix multiplication is used (Huang and Pan [10]), the update time is $O(n^{2.26})$.

These update times are improved if the graph is sparse. For $m < n^{1.54}$, an improved update time is given by $O(n^{1.5+\log m/2\log n})$. More improvements are possible if the size of the transitive closure is $o(n^2)$.

For acyclic graphs, and graphs with small strongly connected components, initialization requires the insertion of the edges incident to each vertex into the data structure, for total costs of $O(n^3)$ and $O(n^{3+\varepsilon})$ respectively.

For general graphs, some of the costs for processing the initial graph may be incurred after the start of the sequence of update operations. The total cost of the initialization procedure is $O(n^{2.26}\ n \lg n)$, which when amortized over a update sequence of length $\Omega(n \lg n)$, costs $O(n^{2.26})$ per update.

We assume a unit cost RAM model with wordsize $O(\log n)$. The algorithm is randomized and has a one-sided error. If the answer to a reachability query is "yes", then the answer is always correct. If the answer is "no", then it is incorrect with probability $O(1/n^c)$, for any fixed constant c. That is, there is a small chance a path $u \leadsto v$ may exist if the answer to the query "Is there a path from u to v?" is negative. The randomization is used only to reduce wordsize. If wordsize n is permitted, or all paths in the graph are of constant length, then the algorithm becomes deterministic.

1.1. Related work

There are only two previously known fully dynamic reachability algorithms for directed graphs, even for the restricted class of acyclic graphs. Both of the algorithms permit only a single edge insertion or deletion in a given update. Neither gives improved running times for the special case of acyclic graphs, nor do they provide sensitivity queries.

It can be argued that the most recent algorithm, by Khanna, Motwani and Wilson (1996)[14], is not a fully dynamic algorithm in that it assumes some knowledge of future update operations at the time of each update. This algorithm uses matrix multiplication as a subroutine. If fast matrix multiplication [2] is used (i.e. $\omega = 2.38$), then the amortized cost of an update is $O(n^{2.18})$. However, a lookahead of $\Theta(n^{.18})$ updates is required. This algorithm is deterministic, but depends heavily on the use of lookahead information.

The other by Henzinger and King (1995)[7] also uses matrix multiplication as a subroutine. This algorithm has an amortized update time of $\tilde{O}(nm^{(\omega-1)/\omega})$, or $\tilde{O}(nm^{0.58})$ if fast square matrix multiplication [2] is used. However, a cost as high as $\Theta(n/\log n)$ may be incurred for each reachability query. Consequently, the adjacency matrix of the transitive closure cannot be updated with each update to the graph. While this algorithm is also randomized with a one-sided error, its techniques are quite different from the algorithm presented here.

Other related work includes *partially* dynamic algorithms. The best result for updates allowing edge insertions only is $O(n)$ amortized time per inserted edge, and $O(1)$ time per query by Italiano (1986)[12], and by La Poutre and van Leeuwen (1987) [16]. This improved upon Ibaraki and Katoh's (1983)[11] algorithm with a total cost of $O(n^3)$ for an arbitrary number of insertions. There is also Yellin's (1993)[19] algorithm, with a total cost of $O(m^*\Delta)$ for any number of insertions, where m^* is the number of edges in the transitive closure and Δ is the out-degree of the resulting graph.

The best deletions-only algorithm for general graphs is by La Poutre and van Leeuwen (1987)[16]. Their algorithm requires $O(m)$ amortized time per edge deletion and $O(1)$ per query. This improved upon the deletions-only algorithm of Ibaraki and Katoh (1983)[11], which can delete any number of edges in $O(n^2(m+n))$ total time.

For acyclic graphs, Italiano (1988)[13] has a deletions-only algorithm which requires $O(n)$ amortized time per edge deletion, and $O(1)$ per query. There is also Yellin's (1993)[19] deletions-only algorithm with a total cost of $O(m^*\Delta)$ for any number of deletions, where m^* is the number of edges in the transitive closure and Δ is the out-degree of the initial graph .

There is also a randomized algorithm with a one-sided error of E. Cohen's (1997) [1] for computing the transitive closure from scratch. This algorithm is based on her linear time algorithm for estimating the size of the transitive closure. However, the cost is $O(mn)$ in the worst case.

The only known lower bound is that for undirected reachability. The lower bound of $\Theta(\log n/\log\log n)$ per update [6] has almost been matched by the upper bound for undirected reachability. Khanna, Motwani and Wilson [14] give some evidence to suggest that dynamic graph problems on directed graphs are intrinsically harder. They show that the certificate complexity of reachability and other directed graph problems is $\Theta(n^2)$, as opposed to $\Theta(n)$ for undirected graphs. This implies that sparsification, a technique for speeding up undirected graphs algorithms, is not applicable to directed graph problems.

2. DEFINITIONS

The following definitions are used throughout the entire paper. Other definitions are introduced as needed.

Definition 2.1. A directed path p is a sequence $p = v_1, v_2, \ldots, v_k$ of distinct vertices $\in V$, such that for every $v_i, v_{i+1} \in p$, $(v_i, v_{i+1}) \in E$. If there exists a directed path beginning with vertex u and ending with vertex v, we say there is a path $u \leadsto v$.

Definition 2.2. In a graph $G = (V, E)$, vertex u can *reach* vertex v, and vertex v is *reachable from* vertex u iff there exists a *directed path* $u \leadsto v$.

Definition 2.3. A *strongly connected component* (SCC) is a maximal subset of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, there exists a path $u \leadsto v$, and a path $v \leadsto u$. The *size* of a strongly connected component is the number of vertices in the component. A vertex which is contained in no cycles is a strongly connected component of size one.

Definition 2.4. For $V = \{1, 2, \ldots, n\}$, the *(reflexive) transitive closure* for G is represented by an $n \times n$ matrix M^*_G such that $M^*_G(i,j) = 1$ if there exists a directed path $i \leadsto j$, or $i = j$; else $M^*_G(i,j) = 0$.

Definition 2.5. A dynamic graph algorithm is a graph algorithm which maintains a graph property during a sequence of *graph updates.* We consider updates of the form:

- insertion (deletion) of an edge set E_v incident to vertex $v \in V$ into (from) G;
- insertion (deletion) of a vertex into (from) V.

The graph property we are interested in is the transitive closure matrix M_G^*.

DEFINITION 2.6. During the sequence of updates, *queries* about the graph property can be made. We consider queries of the form:

- reachability: "Can vertex i reach vertex j?" and
- sensitivity: "Is there a path from i to j not containing edge e?"

DEFINITION 2.7. A *unit cost RAM with wordsize x* is a machine which can process each arithmetic operation in $O(1)$ time using a wordsize x.

3. ACYCLIC GRAPHS

For now, let us assume we are working with a unit cost RAM with wordsize n. In Section 3.4 we show how to reduce wordsize using a randomization technique.

The following algorithm makes use of a very simple idea. We maintain $Paths(i, j)$ to be equal to the number of distinct directed paths $i \leadsto j$ in the current graph, or 1 if $i = j$. Two paths are distinct if their edges differ. The $n \times n$ adjacency matrix M_G^* representing the transitive closure of G is thus given by $M_G^*(i, j) = 1$ if $Paths(i, j) \neq 0$, else $M_G^*(i, j) = 0$.

A sensitivity query of the form "Is there a path from i to j which doesn't contain edge (u, v)?" is answered true if $Paths(i, u) * Paths(v, j) \neq Paths(i, j)$, else the answer is false.

The following lemma for updating a single edge is easy to see, as illustrated by Figure 1.

LEMMA 3.1. *Let G be a directed acyclic graph containing vertices i, j, u and v. If edge (u, v) is inserted (deleted) such that no cycles are created, the number of paths $i \leadsto j$ increases (decreases) by $Paths(i, u) * Paths(v, j)$.*

It is not much more difficult to insert or delete a set of edges incident to a common vertex, as illustrated by Figure 2 and the following lemma.

LEMMA 3.2. *Let G be an acyclic directed graph and E_v be a set of edges incident to a common vertex v. If the edges in E_v are inserted (deleted) such that no cycles are created, then the number of paths $i \leadsto j$ increases (decreases) by [(number old paths $i \leadsto v$) * (number new paths $v \leadsto j$)] + [(number new paths $i \leadsto v$)* (number old paths $v \leadsto j$)] + [(number new paths $i \leadsto v$)*(number new paths $v \leadsto j$)].*

3.1. Update Procedures

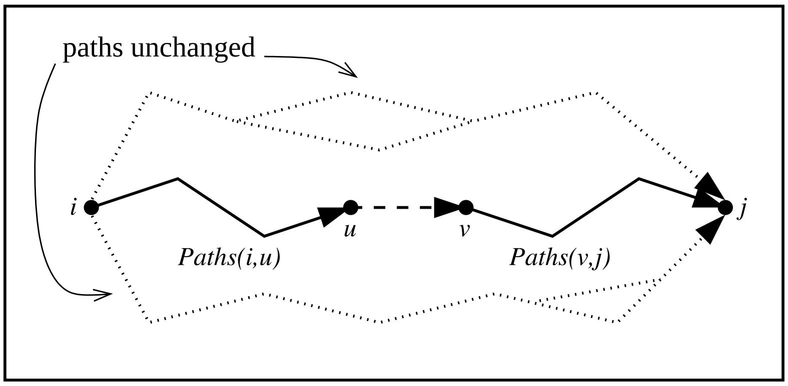

FIG. 1. Inserting a single edge

The number of paths from $i \rightsquigarrow j$ changes only if some path $i \rightsquigarrow j$ contains the updated edge (u, v).

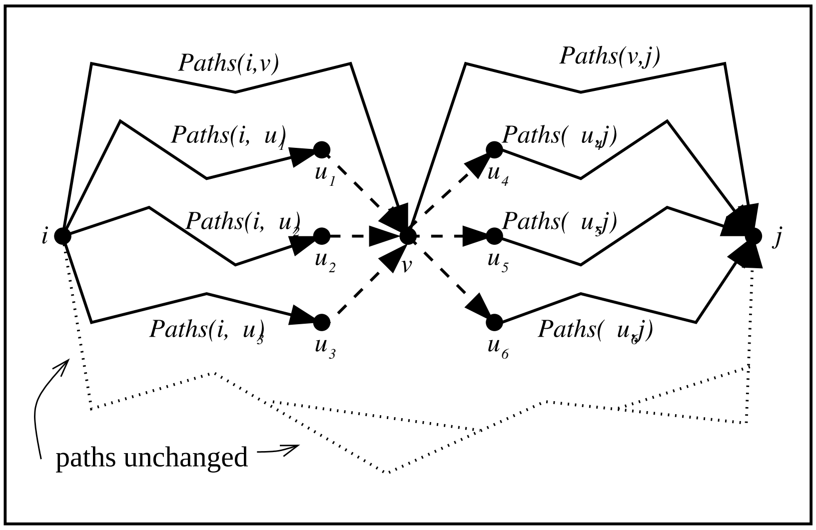

FIG. 2. Inserting an edge set

The number of paths from $i \rightsquigarrow j$ changes only if some path $i \rightsquigarrow j$ contains an updated edge (u_k, v), or an updated edge (v, u_k).

It follows from Lemma 3.2 that the INSERT_ACYCLIC routine in ALGORITHM 1 correctly updates $Paths$ when E_v is inserted into G. The arrays $From$ and To are used as temporary storage, allowing each summation to be carried out only once per vertex per update. Similarly, the DELETE_ACYCLIC routine in ALGORITHM 2 correctly updates $Paths$ when E_v is deleted from G.

ALGORITHM 1 (INSERT_ACYCLIC(E_v, G)).

for all $i \in V$ **do**
    $From(i) \leftarrow \sum_{(u,v)\in E_v} Paths(i,u)$
for all $j \in V$ **do**
    $To(j) \leftarrow \sum_{(v,u)\in E_v} Paths(u,j)$
for all $i, j \in V \setminus \{v\}$ such that $From(i) \neq 0 \bigvee To(j) \neq 0$ **do**
    $Paths(i,j) \leftarrow Paths(i,j) + [Paths(i,v)To(j)$
    $+From(i)Paths(v,j)$
    $+From(i)To(j)]$
$j \leftarrow v$
for all $i \in V \setminus \{v\}$ such that $From(i) \neq 0 \bigvee To(j) \neq 0$ **do**
    $Paths(i,j) \leftarrow Paths(i,j) + [Paths(i,v)To(j)$
    $+From(i)Paths(v,j)$
    $+From(i)To(j)]$
$i \leftarrow v$
for all $j \in V \setminus \{v\}$ such that $From(i) \neq 0 \bigvee To(j) \neq 0$ **do**
    $Paths(i,j) \leftarrow Paths(i,j) + [Paths(i,v)To(j)$
    $+From(i)Paths(v,j)$
    $+From(i)To(j)]$
for all $i, j \in V$ such that $Paths(i,j)$ has increased from 0 **do**
    $M^*_G(i,j) \leftarrow 1$

ALGORITHM 2 (DELETE_ACYCLIC(E_v, G)).

for all $i \in V$ such that $Paths(i,u) \neq 0$ **do**
    $From(i) \leftarrow \sum_{(u,v)\in E_v} Paths(i,u)$
for all $j \in V$ such that $Paths(u,j) \neq 0$ **do**
    $To(j) \leftarrow \sum_{(v,u)\in E_v} Paths(u,j)$
$j \leftarrow v$
for all $i \in V \setminus \{v\}$ such that $From(i) \neq 0 \bigvee To(j) \neq 0$ **do**
    $Paths(i,j) \leftarrow Paths(i,j) - [Paths(i,v)To(j)$
    $+From(i)Paths(v,j)$
    $+From(i)To(j)]$
$i \leftarrow v$
for all $j \in V \setminus \{v\}$ such that $From(i) \neq 0 \bigvee To(j) \neq 0$ **do**
    $Paths(i,j) \leftarrow Paths(i,j) - [Paths(i,v)To(j)$
    $+From(i)Paths(v,j)$
    $+From(i)To(j)]$
for all $i, j \in V \setminus \{v\}$ such that $From(i) \neq 0 \bigvee To(j) \neq 0$ **do**
    $Paths(i,j) \leftarrow Paths(i,j) - [Paths(i,v)To(j)$

```
                    +From(i)Paths(v, j)
                    +From(i)To(j)]
for all i, j ∈ V such that Paths(i, j) has decreased to 0 do
    M*_G(i, j) ← 0
```

To initialize the data structures, we first set $Paths(i,j) = M^*_G(i,j) = 0$ for all i,j. We then apply INSERT_ACYCLIC(E_v, G) for each $v \in V$ and each set E_v of outgoing edges incident to v.

3.2. Analysis

If the algorithm is implemented so that for each vertex u, there is a list of vertices v such that $Paths(v,u) \neq 0$, and a list of vertices w such that $Paths(u,w) \neq 0$, then the algorithm can perform updates in time proportional to the number of edges in the transitive closure. The algorithm requires the computation of a sum of up to $|E_v| = O(n)$ numbers for each vertex, for a total cost of $O(n^2)$ per update. The initialization requires up to n applications of INSERT_ACYCLIC, for a total cost of $O(n^3)$.

3.3. Extensions to Multigraphs and Vertex Insertions and Deletions

It is not difficult to see that this algorithm also works for multigraphs, i.e., graphs with more than one edge between a pair of vertices. In this case, we view each instance of an edge as a distinct path between its endpoints.

If $|E_v| > n$, then the sums can still be computed in $O(n)$ time per vertex by first determining the *multiplicity* of each edge in E_v. The multiplicity of edge (u,v) is equal to the number instances of $(u,v) \in E$. Let $\#(u,v)$ and $\#(v,u)$ denote the multiplicities of (u,v) and (v,u) respectively. Then the computations of the $From$ and To become respectively:

$$From(i) \leftarrow \sum_{(u,v)\in E_v} \#(u,v) \cdot Paths(i,u)$$

and

$$To(j) \leftarrow \sum_{(v,u)\in E_v} \#(v,u) \cdot Paths(u,j)$$

Inserting and deleting isolated vertices can be done easily. Depending upon the implementation details, this can be done in $O(1)$ time.

3.4. Reducing wordsize

The algorithm as stated above may produce numbers as large as 2^n, as there may be 2^n distinct paths in an acyclic graph (without multiple edges). If we are to assume that arithmetic operations can be done in unit time, then it is usual to assume a wordsize of $O(\log n)$ [18].

To reduce the wordsize to $2c \lg n$, for $c \geq 5$, the algorithm begins by randomly picking a prime p of value between n^c and n^{c+1}. All operations (additions, subtractions, multiplications) are performed modulo p and can be done in constant time with wordsize $2c \lg n$. As the number of computations involving a particular prime increases, so do the chances of getting a "false zero", i.e., the result of some computation may equal 0 *mod* p with wordsize $2c\lg n$ when the result is not equal to 0 when computed with wordsize n. To keep the probability of false zeroes low, the algorithm chooses a new prime every n update operations and reinitializes the data structures. To preserve $O(n^2)$ worst case time, the steps of reinitialization with a new prime may be interleaved with the operations involving the current prime.

We observe the following:

LEMMA 3.3. *If $O(n^k)$ arithmetic computations involving numbers of value $\leq 2^n$ are performed modulo a random prime p of value $\Theta(n^c)$, then the probability of a false zero arising is $O(1/n^{c-k-1})$.*

Proof. There are $O(n/\log n)$ prime divisors of value $\Theta(n^c)$ which divide a number of value $\leq 2^n$, and therefore $O(n^{k+1}/\log n)$ prime divisors of any of the numbers generated. By the Prime Number Theorem, see [3], there are approximately $\Theta(n^c/\log n)$ primes of value $\Theta(n^c)$. Hence the probability that a random prime of value $\Theta(n^c)$ divides any of the numbers generated is $O(1/n^{c-k-1})$. ■

COROLLARY 3.1. *After any update in a sequence of n updates taking time $O(n^{2+\beta})$, the probability that there is a pair of vertices i, j such that $M^*_G(i,j) = 0$ while there exists a path $i \rightsquigarrow j$ is $O(1/n^{c-4-\beta})$.*

In particular, for the algorithms just shown, $\beta = 0$. For the remainder of the paper, we assume all operations are done modulo a random prime.

4. GRAPHS WITH SMALL STRONGLY CONNECTED COMPONENTS

In this section we describe a fully dynamic transitive closure algorithm which works quickly when the strongly connected components in the graph are small. In Section 5 we show how to deal with big strongly connected components.

The idea is to represent each strongly connected component in the original graph G by a single "super" vertex in the *compressed graph* G^c. We use the acyclic algorithm from Section 3.1 to maintain the transitive closure $M^*_{G^c}$ for G^c, which is acyclic. When a strongly connected component in G splits into smaller pieces due to a deletion, we "expand" the corresponding super vertex in G^c. As a convention, we use lowercase letters to denote vertices in V and uppercase letters to denote vertices in V^c. We formalize these ideas:

DEFINITION 4.1. Given a general graph $G = (V, E)$, let $c(v)$ denote the (maximal) strongly connected component containing v, for all $v \in V$.

DEFINITION 4.2. For any graph $G = (V, E)$ and subset of edges $E' \subseteq E$, we define the *compressed multiset* E'^c with respect to E' as follows: Edge (I, J) appears with multiplicity k in E'^c, where $k = |\{(u, v) \in E' \mid c(u) = I, c(v) = J \text{ and } c(u) \neq c(v)\}|$.

DEFINITION 4.3. The graph $G^c = (V^c, E^c)$ is the *compressed acyclic multigraph* with respect to G, where V^c is the set of strongly connected components of G, and E^c is the compressed multiset with respect to E. Figure 3 illustrates this.

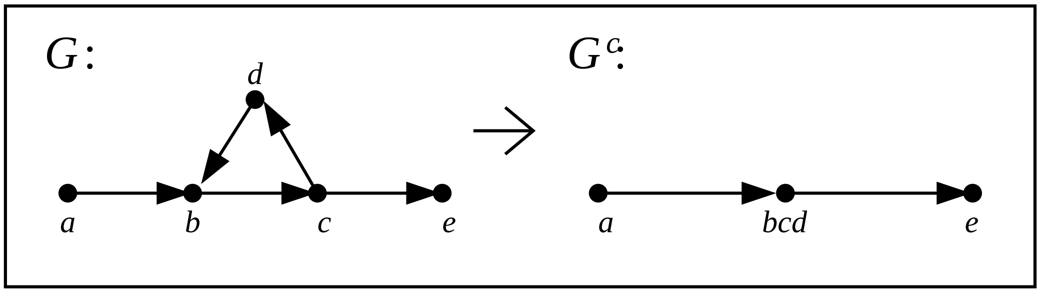

FIG. 3. The compressed acyclic multigraph: The vertices b, c and d which make up the strongly connected component in G are "compressed" into one vertex in G^c.

We define $Paths(I, J)$ for $I, J \in V^c$ as the number of paths from I to J in G^c. The answer to a reachability query: "Is vertex j reachable from vertex i?" is given by $M^*_G(I, J)$, where $c(i)$ is represented by I in G^c and $c(j)$ is represented by J.

4.1. Update Procedures

The update routines maintain representations of the original graph G, and the compressed graph G^c.

4.1.1. Subroutines

- INSERT_ACYCLIC(E_v, G): From Section 3.1.
- DELETE_ACYCLIC(E_v, G): From Section 3.1.
- FIND_SCCS(G): Return the strongly connected components of G. Also determine $c(v)$ for each vertex v. See [3].

4.1.2. Main Routines

The INSERT_SMALL routine in ALGORITHM 3 inserts a set E_v of edges incident to vertex v into G, as illustrated by Figure 4.

ALGORITHM 3 (INSERT_SMALL(E_v, G)).

$C_{old} \leftarrow c(v)$
$E \leftarrow E \cup E_v$
Run FIND_SCCS(G) to find $c(v)$ for each vertex v.
if $C_{old} = c(v)$ /* *no new SCCs have been formed by the insertions* */
  then INSERT_ACYCLIC(E^c_v, G^c) to insert compressed multiset E^c_v into G^c.
  else let $C_1, \ldots, C_k$ be the old SCCs which now compose $c(v)$.
    for all $i = 1, \ldots, k$ **do**
      Let E_{C_i} be the set of edges in E^c with at least one endpoint in C_i.
      DELETE_ACYCLIC(E_{C_i}, G^c) to remove these edges from G^c.
      Remove C_i from V^c.
    Add a vertex labeled $c(v)$ to V^c.

Let E_C be the set of edges in E with one endpoint in C.
INSERT_ACYCLIC(E_C^c, G^c) to add the compressed multiset E_C^c to E^c.
for all i, j such that $Paths(c(i), c(j))$ is changed from 0 **do**
$M_G^*(i, j) \leftarrow 1$

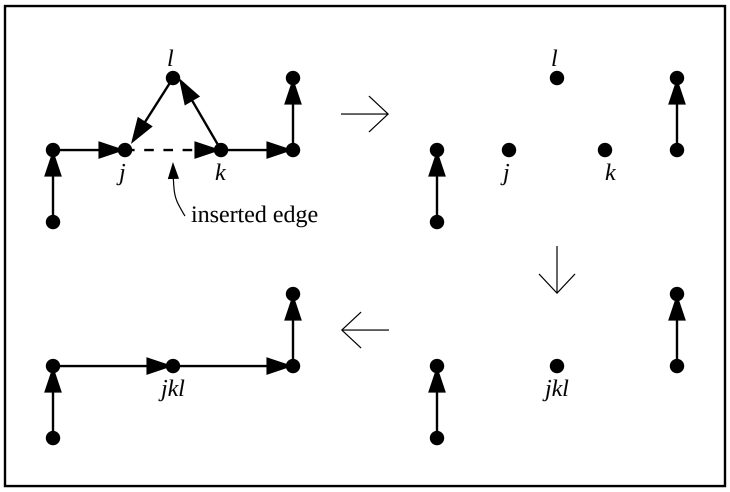

FIG. 4. Compressing a strongly connected component: When an insertion creates a strongly connected component, the essential steps of compression in G^c are: 1. Remove all edges incident to strongly connected vertices; 2. Replace the strongly connected vertices with a super vertex; 3. Add the incident edges back in. Expanding a super vertex is essentially the same procedure, but executed in reverse order.

The DELETE_SMALL routine in ALGORITHM 4 deletes a set E_v of edges incident to vertex v from G. This routine performs the same steps as INSERT_SMALL in ALGORITHM 3, but in reverse order.

ALGORITHM 4 (DELETE_SMALL(E_v, G)).

$C_{old} \leftarrow c(v)$
$E \rightarrow E \setminus E_v$
Run FIND_SCCs(G) to find $c(v)$ for each vertex v.
if $C_{old} = c(v)$ */* no SCCs have split apart due to the deletions */*
then DELETE_ACYCLIC(E_v^c, G^c) to delete E_v^c from G^c.
else Let $C_1, C_2, .., C_k$ be the new SCCs into which C_{old} has decomposed.
Let $E_{c(v)} \subseteq E^c$ be the set of all edges incident to $c(v)$ in G^c.
DELETE_ACYCLIC($E_{c(v)}, G^c$) to remove $E_{c(v)}$ from G^c.
Remove $c(v)$ from G^c.
for all $i = 1, \ldots, k$ **do**
Insert a vertex labelled C_i into V_c.
Let E_{C_i} be the set of edges in E with one end in C_i.
INSERT_ACYCLIC($E_{C_i}^c, G^c$) to add $E_{C_i}^c$ to E^c.
for all i, j such that $Paths(c(i), c(j))$ is changed to 0 **do**
$M_G^*(i, j) \leftarrow 0$

To initialize the data structures, we simply insert the outgoing edges E_v incident to each vertex v using INSERT_SMALL(E_v, G).

4.2. Analysis

The cost of the algorithms is dominated by the calls to INSERT_ACYCLIC and DELETE_ACYCLIC, each of which cost $O(n^2)$. If no strongly connected components change, then there is a single such call.

A deletion may cause a vertex of V^c, which represents a strongly connected component consisting of s vertices of V, to be removed and replaced by up to s new vertices in V^c. Similarly, an insertion which results in the creation of a new strongly connected component containing s vertices in V may require the deletion of up to s vertices from V^c. Each insertion (deletion) of a vertex into (from) V^c requires a call to INSERT_ACYCLIC and DELETE_ACYCLIC, respectively, resulting in a total cost of $O(n^2 s)$ per update. If we assume an upper bound of n^ε on the size of any strongly connected component, then each update costs at most $O(n^{2+\varepsilon})$. Initialization requires up to n applications of INSERT_SMALL, for a total cost of $O(n^{3+\varepsilon})$.

5. GENERAL GRAPHS

The main idea is to maintain a graph $G' = (V, E')$ such that G' contains no big strongly connected components (see Figure 5). The edge set $E' \subseteq E$ contains all edges between vertices which are not both in the same big component, and possibly other edges. This property ensures that there is a path from i to j in G' if there is a path from i to j in G which does not include any intermediate vertices contained in big strongly connected components. Then a path in G can be described by a concatenation of paths in G', where two such paths are joined if and only if the end of the first is contained in the same big component as the start of the second. The transitive closure of G' is maintained using the INSERT_SMALL and DELETE_SMALL routines of the previous section. After each update, paths in G' are concatenated as described to determine all paths of G.

Initially, E' contains only edges of E whose endpoints are not both contained in the same big component. An edge in $E \setminus E'$ is added to E' if and when both its endpoints cease to be in the same big component. Once an edge is added to E', it is not removed unless it is deleted from G. If a big component forms again due to an edge insertion, only the newly inserted edge which causes the big component to form is omitted from E'.

5.1. Definitions

DEFINITION 5.1. A strongly connected component is *big* if it contains more than n^ε vertices.

DEFINITION 5.2. Let $M^*_{G'}$ be the $n \times n$ matrix representing the transitive closure of G', as maintained by our algorithm for small components.

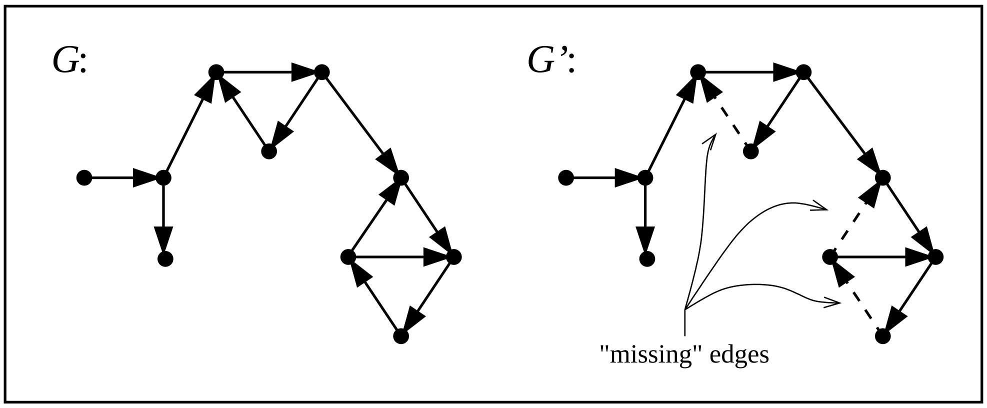

FIG. 5. The graph G' with "missing" edges: The edges which would cause a big strongly connected component are simply not inserted into G'.

Definition 5.3. Let $c(v)$ denote the strongly connected component in G containing vertex v.

Definition 5.4. Let $b(v)$ denote the big strongly connected component in G containing vertex v, provided v is in a big component.

Definition 5.5. Let $V_B = \{B_1, B_2, \ldots, B_k\}$ be the set of big strongly connected components of G. Since each has size greater than n^ε, we have $k \leq n^{1-\varepsilon}$.

Definition 5.6. Let B be a $k \times k$ Boolean matrix such that $B(i,j) = 1$ iff there exists a pair of vertices $u \in B_i$ and $v \in B_j$ such that $M^*_{G'}(u,v) = 1$. That is, B is the adjacency matrix for the graph whose vertices are the big strongly connected components of G; there is an edge from big strongly connected component B_i to B_j iff some vertex in B_i can reach some vertex in B_j in G'.

Definition 5.7. Let $M^*_{G'}B$ be the $n \times k$ Boolean matrix such that $M^*_{G'}B(i,j) = 1$ iff $M^*_{G'}(i,v) = 1$ for $i \in V$ and any $v \in B_j$, or $i \in B_j$, i.e., if there is a path in G' from vertex i to some vertex v in big component B_j, or i is in B_j itself.

Definition 5.8. Let $BM^*_{G'}$ be the $k \times n$ Boolean matrix such that $BM^*_{G'}(i,j) = 1$ iff $M^*_{G'}(v,j) = 1$ for $j \in V$ and any $v \in B_i$, or $j \in B_i$, i.e., if there is a path in G' from some vertex v in big component B_i to vertex j, or j is in B_i itself.

5.2. Update Procedures

5.2.1. Subroutines

- Insert_Small(E_v, G): From Section 4.1.
- Delete_Small(E_v, G): From Section 4.1.
- Find_Big_SCCs(G): Return the big strongly connected components of G. Also determine $b(v)$ for each vertex v in a big strongly connected component. See [3].
- Approx_Vertex_Cover(V): Return a vertex cover on E which is no more than twice the size of the optimal vertex cover (i.e. a vertex cover of minimum size). See [3].
- Generate_Matrices(V_B): Generate B, $M^*_{G'}B$, $BM^*_{G'}$, and the transitive closure M^*_B of B. Then return $M^*_G = M^*_{G'} \bigvee (M^*_{G'}B \times M^*_B \times BM^*_{G'})$, assuming $M^*_{G'}$ is correct. See Algorithm 5.

Algorithm 5 (Generate_Matrices(V_B)).

if $V_B = \emptyset$ /* *there exist no big components* */
  then $M^*_G \leftarrow M^*_{G'}$
  else Initialize B, $M^*_{G'}B$ and $BM^*_{G'}$ to 0.
    for all big components $B_i, B_j \in V_B$ **do**
      for all $u \in B_i, v \in B_j$ **do**
        if $M^*_{G'}(u,v) = 1$ **then** $B(i,j) \leftarrow 1$
    for all vertices $i \in V$ and big components $B_j \in V_B$ **do**
      for all $v \in B_j$ **do**
        if $M^*_{G'}(i,v) = 1 \vee i \in B_j$ **then** $M^*_{G'}B(i,j) \leftarrow 1$
    for all big components $B_i \in V_B$ and vertices $j \in V$ **do**
      for all $v \in B_i$ **do**
        if $M^*_{G'}(v,j) = 1 \vee j \in B_i$ **then** $BM^*_{G'}(i,j) \leftarrow 1$
    Compute the transitive closure M^*_B of B with a static method.
    $M^*_G \leftarrow M^*_{G'} \bigvee (M^*_{G'}B \times M^*_B \times BM^*_{G'})$

5.2.2. *Main Routines*

The Insert(u,v) routine in Algorithm 6 calls Insert_Small($\{(u,v)\}, G'$) to insert (u,v) into G' if u and v are not both in the same big component. If u and v are in the same big component, then edge (u,v) is marked and will be inserted into G' by Delete if and when u and v cease to be in the same big component.

Algorithm 6 (Insert(u,v)).

$E \leftarrow E \cup \{(u,v)\}$
$V_B \leftarrow Find\_Big\_SCCs(G)$ /* *find $b(v)$ for each vertex v* */
if $b(u) \neq b(v)$
  then Insert_Small($\{(u,v)\}, G'$) to insert (u,v) into G'.
    Generate_Matrices(V_B)
  else Mark (u,v).
    Generate_Matrices(V_B)

The Delete routine in Algorithm 7 calls Delete_Small to delete edges between small components in G'. It then finds the set E_{insert} of marked edges in $E \setminus E'$ whose endpoints have ceased to be in the same big component. APPROX_VERTEX_COVER(E_{insert}) is then called to find a vertex cover on E_{insert}. Finally, INSERT_SMALL is called for each vertex in the approximate vertex cover, to insert E_{insert} into G'.

ALGORITHM 7 (DELETE(E_v)).

```
E ← E \ E_v
V_B ← Find_Big_SCCs(G)                    /* find b(v) for each vertex v */
DELETE_SMALL(E_v ∩ E', G')
Let E_insert be the set of marked edges {(u, w)| such that b(u) ≠ b(w)}.
Unmark all edges in E_insert.
V_{E_Insert} ← Approx_Vertex_Cover(E_Insert)
for all u ∈ V_{E_Insert} do
    Let E_u be the edges in E_insert incident to u.
    INSERT_SMALL(E_u, G') to insert E_u into G'.
    Remove E_u from E_insert.
GENERATE_MATRICES(V_B)
```

To efficiently insert the initial edges, the INITIALIZE routine in ALGORITHM 8 finds a vertex cover on the edges E_{init}which are ready for insertion into G' (i.e. the endpoints of each edge lie in different big components) by calling APPROX_VERTEX_COVER on E_{init}. The remaining edges are marked, and will be inserted by DELETE if and when both endpoints cease to be in the same big component.

ALGORITHM 8 (INITIALIZE(G)).

```
G' ← (V, ∅)
V_B ← Find_Big_SCCs(G)                    /* find b(v) for each vertex v */
Let E_init be the set of edges {(u, w)| such that b(u) ≠ b(w)} .
Mark all edges in E \ E_init.
V_{E_init} ← Approx_Vertex_Cover(E_init)
for all u ∈ V_{E_init} do
    Let E_u be the edges in E_init incident to u.
    INSERT_SMALL(E_u, G') to insert E_u into G'.
    Remove E_u from E_init.
GENERATE_MATRICES(V_B)
```

5.3. Analysis

To proceed with the analysis, we observe that the edges in the current graph can be partitioned into three subsets.

• The set of unmarked edges. These edges were inserted into G' at some point when their endpoints were not in the same big component. The cost of inserting these edges into G' has already been realized.

• The set of marked initial edges which were inserted into G by the INITIALIZE routine. These edges have not yet been inserted into G'. The cost of inserting marked initial edges into G' is charged to the Initialize routine. An initial edge which is deleted and then reinserted becomes a non-initial edge.

• The set of marked non-initial edges which were inserted into G by the Insert routine after initialization. These edges have not yet been inserted into G'. The cost of inserting a marked non-initial edge (u, v) into G' is charged to Insert(u, v).

We use an amortized analysis. We first establish the cost of each subroutine on a graph with m edges and n vertices, and then use these costs to determine how much the Insert, Delete, and Initialize routines must be charged to maintain a non-negative balance.

- Let FBS be the cost of Find_Big_SCCs. This is $O(n + m)$ from [3].
- Let AVC be the cost of Approx_Vertex_Cover. This is $O(n + m)$ from [3].
- Let IS be the cost of Insert_Small. This is $O(n^{2+\varepsilon})$ from Section 4.2.
- Let DS be the cost of Delete_Small. This is $O(n^{2+\varepsilon})$ from Section 4.2.
- Let GM be the cost of Generate_Matrices. This requires more reasoning.

Initially, the matrices B, $BM^*_{G'}$ and $M^*_{G'}B$ are generated in $O(n^2)$ time. The next step is finding the transitive closure M^*_B of B. The transitive closure of an $n \times n$ matrix can be computed in $O(n^\omega)$ time, where where $O(n^\omega)$ is the cost of multiplying two $n \times n$ matrices [15]. Since there are no more than $n^{1-\varepsilon}$ big components, finding the transitive closure of B takes at most $O(n^{\omega(1-\varepsilon)})$ time.

The cost of computing $M^*_{G'}B \times M^*_B \times BM^*_{G'}$ depends on the technique used. The asymptotically fastest way is with fast rectangular matrix multiplication [10] in time $n^{\omega(1,1-\varepsilon,1)}$, where

$$\omega(1,1-\varepsilon,1) = \left(\frac{1}{(1-\beta)\log q}\right) \times \log \frac{\begin{pmatrix} ((1-\varepsilon)\beta)^{((1-\varepsilon)\beta)} \\ \times(2(1-\beta))^{2(1-\beta)} \\ \times((1-\varepsilon)(1-\beta)+2\beta)^{((1-\varepsilon)(1-\beta)+2\beta)} \\ \times(q+2)^{(2+(1-\varepsilon))} \end{pmatrix}}{(2+(1-\varepsilon))^{(2+(1-\varepsilon))}}$$

The variable q is an integer, and β is a small real value, typically between 0.005 and 0.05. These values are chosen such that the equation is minimized.

To use square matrix multiplication instead, we observe that multiplying an $n \times k$ matrix by a $k \times n$ matrix can be done with $(n/k)^2$ multiplications of $k \times k$ matrices. Here, $k \leq n^{1-\varepsilon}$, so the multiplication costs no more than $O((n^\varepsilon)^2 n^{\omega(1-\varepsilon)})$, or $O(n^{2\varepsilon+\omega-\omega\varepsilon})$.

Matrix multiplication is not necessary if the graph is sparse. Every pair of vertices joined by a path which runs through some B_i can be found by using one depth-first search to determine all vertices which can reach big component B_i, and another depth-first search to determine all vertices which can be reached by B_i. To do this for each B_i costs $O(m)$ per B_i, or $O(mn^{1-\varepsilon})$.

We now determine how much each update routine must be charged to maintain a non-negative balance.

LEMMA 5.1. *The required charge to* INSERT *is* $\leq FBS + IS + GM$.

Proof. Each call to INSERT makes exactly one call to FIND_BIG_SCCs and one call to GENERATE_MATRICES. The routine INSERT_SMALL is called from within INSERT only if the inserted edge does not have both endpoints in the same big component. Otherwise it may be inserted by a call to INSERT_SMALL from within the DELETE routine. In any case, it suffices to charge INSERT for all calls to INSERT_SMALL which may be made on its behalf. ■

LEMMA 5.2. *The required charge to* INITIALIZE *is* $\leq FBS + AVC + 2n\lceil\log_2 n\rceil \cdot IS + GM$.

Proof. Each call to INITIALIZE makes exactly one call to each of FIND_BIG_SCCs, APPROX_VERTEX_COVER and GENERATE_MATRICES. In addition, INSERT_SMALL may initially be called once for each vertex to insert edges whose endpoints are not in the same big component. The remaining initial edges may be inserted by calls to INSERT_SMALL from within the DELETE routine. From Lemma 5.3, the number of calls to INSERT_SMALL from within the DELETE routine for the purpose of deleting initial edges is bounded by $2n\lceil\log_2 n\rceil$. ■

LEMMA 5.3. *The* DELETE *routine performs at most* $2n\lceil\log_2 n\rceil$ *calls to* INSERT_SMALL *for the purpose of inserting marked initial edges into* G'.

Proof. We recursively define a set of equivalence classes $C_1, C_2, \ldots, C_k$ at step t (i.e. after t updates) on the vertex set of the current graph. At step 0 (i.e. at initialization), we define $x \equiv y$ iff x and y are in the same big strongly connected component. At step t, we define $x \equiv y$ iff $x \equiv y$ at step $t-1$, and x and y are contained in a big strongly connected component. Note that if (x, y) is a marked initial edge in the current graph, then $x \equiv y$. Without loss of generality, the equivalence classes form a partially ordered set, where $|C_i| \geq |C_{i+1}|$. We observe that marked initial edges in E_{insert} are inserted into G' only when an equivalence class C splits into smaller equivalence classes $C'_1, C'_2, \ldots, C'_k$. We also observe that each marked initial edge to be inserted has its endpoints in different members of $\{C'_1, C'_2, \ldots, C'_k\}$. These edges can be inserted using the vertices in $C'_2, \ldots, C'_k$ as a vertex cover, at a cost proportional to $|C'_2| + \ldots + |C'_k|$. That is, when an equivalence class C splits into $C'_1, C'_2, \ldots, C'_k$, we charge the vertices in the smaller resulting equivalence classes for the insertions. After a number of steps, it may be the case that each equivalence class contains exactly one vertex. At this point there are no more marked initial edges in the graph. Now each vertex v can be charged at most $\lceil\log_2 n\rceil$ times for inserting marked initial edges, because the equivalence class containing v decreases in size by a factor of at least 2 each time v is charged. We also know that after each split, the DELETE routine uses a vertex cover which is no larger than twice the size of the optimal vertex cover for inserting the marked edges [3]. Using the optimal vertex cover to insert the marked edges can be no less efficient than using the set of vertices in the smaller equivalence classes, so it follows that no more than $2n\lceil\log_2 n\rceil$ calls to INSERT_SMALL are made for the purpose of inserting marked initial edges into G'. ■

LEMMA 5.4. *The required charge to* DELETE *is* $\leq FBS + DS + AVC + GM$.

Proof. Each call to DELETE makes exactly one call to each of FIND_BIG_SCCS, DELETE_SMALL, APPROX_VERTEX_COVER and GENERATE_MATRICES. All calls to INSERT_SMALL are on behalf of INITIALIZE or INSERT, and have been paid for by INITIALIZE or INSERT respectively. ■

THEOREM 5.1. *If fast rectangular matrix multiplication [10] is used, the total cost of the initialization is* $O(n^{2.26}(n\lg n))$*, which when amortized over an update sequence of length* $\Omega(n\lg n)$*, costs* $O(n^{2.26})$ *per update. Furthermore, the amortized cost of inserting an edge, or deleting an edge set* E_v *incident to a common vertex* v *is* $O(n^{2.26})$*.*

Proof. The amounts which INITIALIZE, INSERT and DELETE must be charged can be minimized by finding an appropriate value for ε. It follows from Lemmas 5.1, 5.2 and 5.4 that that the smallest value for ε can be found by minimizing $O(n^{2+\varepsilon} + x)$, where x depends on the technique used for obtaining $M^*_{G'}B \times M^*_B \times BM^*_{G'}$ within GENERATE_MATRICES. If fast rectangular matrix multiplication [10] is used, $O(n^{2+\varepsilon} + n^{\omega(1,1-\varepsilon,1)})$ must be minimized. Experimentally we found that this value is minimized when $\varepsilon = .257..$, $q = 10$ and $\beta = 0.0226$, yielding a running time of $O(n^{2.26})$. To find these numbers, we wrote a computer program which iterated over a range of integer values for $q \in 0,\ldots,30$, and real values for $\beta \in 0.005,\ldots,0.05$ and $\varepsilon \in 0,\ldots,1$. The program calculated $2+\varepsilon-\omega(1,1-\varepsilon,1)$ for every combination and kept track of the best results encountered. The program used an increment size of 0.0001 for non-integers. If square matrix multiplication is used, $O(n^{2+\varepsilon}+n^{2\varepsilon+\omega-\omega\varepsilon})$ must be minimized. Setting ε to $\frac{\omega-2}{\omega-1}$ yields a sum of $O(n^{2+\frac{\omega-2}{\omega-1}})$. If the graph is sparse and the depth-first search technique is used, $O(n^{2+\varepsilon} + mn^{1-\varepsilon})$ must be minimized. Setting $\varepsilon = \log(m/n)/2\log n$ yields a sum of $O(n^{2+\log(m/n)/2\log n})$. If $m < n^{1.54}$, the charge per update is less than $O(n^{1.5+\log m/2\log n})$ in this case. ■

REFERENCES

1. E. Cohen, "Size-Estimation Framework with Applications to Transitive Closure and Reachability", *J. of Computer and System Sciences 55*, 1997, pp. 441–453.
2. D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation 9*, 1990, pp. 251–280.
3. T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
4. S. Even and Y. Shiloach, "An On-Line Edge-Deletion Problem", *J. ACM 28*, 1981, pp. 1–4.
5. J. Feigenbaum and S. Kannan, "Dynamic Graph Algorithms", in *Handbook of Discrete and Combinatorial Mathematics*, pp. 583-591.
6. M. L. Fredman and M. Rauch Henzinger, "Lower Bounds for Fully Dynamic Connectivity Problems in Graphs", *Algorithmica 22*, 1998, pp. 351–362.
7. M.R. Henzinger and V. King, "Fully Dynamic Biconnectivity and Transitive Closure", *Proc. 36th Symp. on Foundations of Computer Science (FOCS)*, 1995, pp. 664–672.

8. J. Holm and K. de Lichtenberg and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity", *Proc. 30th Symp. on Theory of Computing (STOC)*, 1998, pp. 79-89.

9. M. R. Henzinger and M. Thorup, "Sampling to Provide or to Bound: With Applications to Fully Dynamic Graph Algorithms," *Random Structures and Algorithms 11*, 1997, pp. 369–379.

10. X. Huang and V. Y. Pan, "Fast Rectangular Matrix Multiplication and Applications" *Journal of Complexity 14* (1998), pp. 257-299.

11. T. Ibaraki and N. Katoh, "On-line computation of transitive closure of graphs, *Information Processing Letters*, 1983, pp. 95-97.

12. G. F. Italiano, "Amortized efficiency of a path retrieval data structure", *Theoretical Computer Science 48*, 1986, pp. 273-281.

13. G. F. Italiano, "Finding Paths and Deleting Edges in Directed Acyclic Graphs" *Information Processing Letters*, 1988, pp. 5-11.

14. S. Khanna, R. Motwani and R. Wilson, "Graph Certificates and Lookahead in Dynamic Directed Graph Problems, with Applications", *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, 1996.

15. D. Kozen, *The Design and Analysis of Algorithms*, Springer-Verlag, 1992, pp. 26-7.

16. H. La Poutré and J. van Leeuwen, "Maintenance of transitive closure and transitive reduction of graphs", *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 314, Springer Verlag, Berlin, 1987, pp. 106–120.

17. V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik 14*, 1969, pp. 354–356.

18. R. E. Tarjan *Data Structures and Network Algorithms*, CMBS 44, SIAM, 1993.

19. D. M. Yellin, "Speeding up dynamic transitive closure for bounded degree graphs" *Acta Informatica 30*, 1993, pp. 369–384.