A Space Saving Trick for Directed Dynamic Transitive Closure and Shortest Path Algorithms

Valerie King[1] and Mikkel Thorup[1,2]

[1] Department of Computer Science,
University of Victoria, Victoria, BC.
val@csr.uvic.ca, mthorup@research.att.com
[2] AT&T Labs–Research,
Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932.
mthorup@research.att.com

Abstract. We present a simple space saving trick that applies to many previous algorithms for transitive closure and shortest paths in dynamic directed graphs. In these problems, an update can change all edges incident to a node. The basic queries on reachability and distances should be answered in constant time, but also paths should be produced in time proportional to their length. For:

Transitive closure of Demetrescu and Italiano (FOCS 2000)
Space reduction from $O(n^3)$ to $O(n^2)$, preserving an amortized update time of $O(n^2)$.

Exact all-pairs shortest dipaths of King (FOCS 1999)
Space reduction from $\tilde{O}(n^3)$ to $\tilde{O}(n^2\sqrt{nb})$, preserving an amortized update time of $\tilde{O}(n^2\sqrt{nb})$, where b is the maximal edge weight.

Approximate all-pairs shortest dipaths of King (FOCS 1999)
Space reduction from $\tilde{O}(n^3)$ to $\tilde{O}(n^2)$, preserving an amortized update time of $\tilde{O}(n^2)$.

Several authors (Demetrescu and Italiano, FOCS 2000, and Brown and King, Oberwolfach 2000) had discovered techniques to give a corresponding space reduction, but these techniques could be used to show only the existence of a desired dipath, and could not be used to produce the actual path.

1 Introduction

This paper is on saving space for dynamic transitive closure and shortest paths in dynamic directed graphs. For example, the space saving for transitive closure is from $O(n^3)$ to $O(n^2)$. Papers on algorithm are often more concerned with time than space, but a bad space complexity hits like a wall, whereas time only kills slowly. Ignoring for a moment the O-notation, if a computer has gigabyte of memory, with n^3 space, you can only run on graphs with up to $1,000$ nodes, whereas with n^2 space, you can deal with more than $30,000$ nodes. Our space savings techniques are simplifying and tend to decrease rather than increase the constants hidden in the O-notation, both in time and in space.

J. Wang (Ed.): COCOON 2001, LNCS 2108, pp. 268–277, 2001.

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 graph algorithm should process queries quickly and must perform update operations faster than computing from scratch (as performed by the fastest "static" algorithm).

In this paper, we give dynamic algorithms for weighted directed graphs. The problems considered are as follows, where the queries range over all pairs of vertices u and w:

Transitive closure:
Reachability query: Is there a path from u to w in the current graph?
Path query: Return a path from u to w.

Exact all-pairs shortest paths:
Distance query: What is the distance from u to w?
Path query: Return a path from u to w.

Approximate all-pairs shortest paths:
Distance query: What is an upper bound on the distance from u to w within a factor of $(1+\epsilon)$?
Path query: Return a path from u to w of length within a factor $(1+\epsilon)$ of the distance.

The following update operations are allowed:

- multi-insert(E_v): inserts a set of edges incident to the same vertex v.
- delete(e): deletes any edge currently in the graph.

In this paper, we are only considering amortized time bounds. Deleting an edge is free in that it is paid for by the preceding insertion/initialization of the edge.

We present a simple space saving trick that for all the above problems reduce the space substantially in the current fastest algorithm. More precisely, assuming that the graphs start with an empty edge set, we achieve:

Transitive closure of Demetrescu and Italiano [3,4] Space reduction from $O(n^3)$ to $O(n^2)$, preserving an amortized multi-insert time of $O(n^2)$.

Exact all-pairs shortest dipath of King [9] Space reduction from $\tilde{O}(n^3)$ to $\tilde{O}(n^2\sqrt{nb})$, preserving an amortized multi-insert time of $\tilde{O}(n^2\sqrt{nb})$, where b the maximal edge weight.

Approximate all-pairs shortest dipaths of King [9] Space reduction from $\tilde{O}(n^3)$ to $\tilde{O}(n^2)$, preserving an amortized multi-insert time of $\tilde{O}(n^2)$.

The above update times should be compared with the best static algorithms. Transitive closure and approximate shortest paths can be computed essentially as fast matrix multiplication [8,10,13], which takes $\tilde{O}(n^{2.376})$ time, and for exact all pairs shortest paths, can be computed in $\tilde{O}(n^{2.575})$ time, again using fast

matrix multiplication. Thus, the amortized update times may not seem that impressive. However, fast matrix multiplication is considered rather slow, and it is an open problem to construct a combinatorial algorithm just for transitive closure with sub-cubic running time. Our algorithms are purely combinatorial, and in this view, the dynamic speed-up is optimal in that we can construct a graph from scratch with a multi-insert from each vertex, paying a total cost of $n \times O(n^2) = O(n^3)$ to construct the transitive closure, thus matching the best static combinatorial time bound. Our reduction of the space from $O(n^3)$ to $O(n^2)$ adds optimality with respect to space.

Our ideas also apply to the older single source algorithms of Even and Shiloach [5] and of Ramalingam and Reps [11], improving their internal space from $O(m)$ to $O(n)$. Hence, if used for all pairs shortest paths, the improvement is from $O(mn)$ to $O(n^2)$. We are here particularly interested in Ramalingam and Reps' algorithms because it has proved efficient in practice [7,6]. Our idea also simplifies the implementation which typically reduces the worked performed.

A common theme of all the above algorithms is that they keep tables of distance information along with witnesses. A table entry can only increase if all its witnesses are lost. This theme goes back to Even and Shiloach's decremental single shortest path algorithm [5], and is also used in the fully-dynamic single source shortest path algorithm of Ramalingam and Reps [11]. Unfortunately, there may be many witnesses for each entry, and it is the storage of these witnesses that cause a prohibitive space-overhead.

Recently space improvement similar to ours were reported by Brown and King [2] and of Demetrescu and Italiano [4]. Their idea was to maintain a counter of the number of witnesses for each entry. Again, the entry can only increase if its counter goes to 0. The problem in their solution is that they loose track of witnesses, and then they cannot answer path queries.

Our challenge of identifying witnesses rather than just knowing their existence is similar in spirit and motivation to that of finding witnesses for Boolean matrix multiplication as done by Alon et al. [1], though the techniques we apply here are combinatorial and completely different.

Our contribution is to observe that if we are careful about the order in which we scan for witnesses, then we only need to store the first witness that we meet. When a witness is lost, we just continue the scan for the next witness. Our problem is then to identify scanning orders and properly define witnesses so that the same potential witnesses are not scanned repeatedly.

Contents First, in §2, we present the simplest version of the trick, applying it to Even and Shiloach's classic algorithm [5]. In §2, we also point out the modification needed for Ramalingam and Reps' algorithm [11]. Next, in §3, we show how to tailor the trick to King's exact shortest paths and transitive closure algorithms [9]. For space reasons we defer the treatment of King's exact and approximate shortest paths and Ranmalingam and Reps shortest paths algorithm to the journal version of this paper.

Notation We working on a dynamic directed weighted graph $G = (E, V)$ with edge weight function $\ell : E \to \mathbb{N}$. For any subgraph H of G, $V(H)$ is its vertex set and $E(H)$ its edge set. For any pair of vertices (v, w), $\mathsf{dist}(v, w)$ denotes their distance in G. For each vertex v, $in(v)$ and $out(v)$ denotes its incoming and outgoing edges in G.

2 Even and Shiloach's Algorithm

Our basic idea is most easily presented in terms of Even and Shiloach's decremental single shortest path algorithm [5]. Their result is that we can maintain distances up to some threshold Δ in $O(m\Delta)$ total time. In [5] they just consider unweighted graphs and threshold distance $\Delta = n$, giving them an $O(mn)$ bound for maintaining all distances from the source. We generalize this to weighted graphs.

For a given source node s, the Even-Shiloach algorithm maintains a *shortest path graph* H_s which is the union of all shortest paths from s of length at most Δ. For each vertex v, we maintain its distance $d_s(v)$ from s, and, for each distance $i = 0, ..., \Delta$, we maintain the set of vertices v with $d_s(v) = i$. During our response to an edge deletion, $d_s(v)$ will be incremented by one at a time until it reaches the new distance from s to v.

In the shortest path graph H_s, each edge (u, v) is a *witness* of the current distance to v, in the sense that $\mathsf{dist}(s, v) = d_s(v) = d_s(u) + \ell(u, v) = \mathsf{dist}(s, u) + \ell(u, v)$. Hence, when an edge (u, v) is deleted, it affects distances only if (u, v) is in H_s, and there is no other edge (x, v) to v in H_s.

Suppose the deleted edge (u, v) is a last witness to v in H_s. This implies that $d_s(v)$ has increased, and the outgoing edges (v, w) are no longer valid witnesses of $d_s(w)$. A cascading effect is started: when a node w loses all its incoming edges in H_s, we must delete all its outgoing edges in H_s.

To update H_s efficiently, the cascade is performed one distance i at the time. Let (u, v) be the original edge deleted. Starting with $i \leftarrow d_s(v)$, we do as follows. For each vertex w with $d_s(w) = i$ and no incoming edges in H_s, we remove all its outgoing edges in H_s, and increment $d_s(w)$ by one. This preserves $d_s(w) \leq \mathsf{dist}(s, w)$, and now, for all u with $d_s(u) \leq i$, we know that $d_s(u) = \mathsf{dist}(s, u)$. Next, for each w with $d_s(w)$ increased, we consider all its incoming edges (x, w) in G. The edge (x, w) witnesses $d_s(w) = \mathsf{dist}(s, w)$ if $d_s(x) \leq i$ and $d_s(x) + \ell(x, v) = d_s(w)$, and if so, (x, w) is added to H_s. If no such witness is found, we still have $d_s(w) < \mathsf{dist}(s, v)$. We now set $i \leftarrow i + 1$ and repeat. We terminate when $i = \Delta + 1$ or when we get to a distance i where all nodes have incoming edges in H_s.

For a detailed description and analysis of the above algorithm, the reader is referred to [5]. The running time is dominated by the scanning for witnesses; each time we increment $d_s(w)$, we scan all its incoming edge to see if they witness the new distance. Since this can happen at most Δ times for each w, the total cost is $O(\Delta \cdot \sum_{w \in V(G)} |\{(x, w) \in E(G)\}|) = O(\Delta m)$.

2.1 Our Space Saving Trick

The internal space of the Even-Shiloach algorithm, i.e. the space beyond the input graph G, is $O(m)$ for storing the shortest path graph H_s. We will reduce this to $O(n)$. If we want a decremental data structure for each node, then the space is reduced from $O(mn)$ to $O(n^2)$.

For each vertex v, its list $in(v)$ of incoming edges in G stores the incoming edges in some order. Instead of storing all of H_s, for each v, we store only the edge (x, v) from H_s which is first in $in(v)$. We then have the unique shortest path tree T_s from s, spanning all nodes within distance Δ and such that the parent pointer of each node v is the first edge in $in(v)$ which is on a shortest path from the source to v.

The deletion of an edge (u, v) only has a consequence if (u, v) is in T_s. In that case, we search for a replacement witness. Starting from the successor of (u, v), we scan the edges in $in(v)$, stopping if we reach a new witness (x, v) with $d_s(x) + \ell(x, v) = d_s(v)$. If that happens , we replace (u, v) with (x, v) in T_s, and stop. Else we conclude that v has also lost its last incoming edge in H_s. As in the Even-Shiloach algorithm, each incoming edge to each vertex v is considered at most once for each value of $d_s(v)$. Hence, we preserve the running time of $O(\Delta m)$. However, since T_s is a tree, we have reduced the internal space from $O(m)$ to $O(n)$. Also we save work when edges from $H_s \setminus T_s$ are deleted.

3 King's Algorithms

As the basis for all the fully-dynamic algorithms of King in [9], we want to maintain all distances up to some threshold Δ. Besides edge deletions, we have multi-inserts that for each vertex v can add any set of edges incident to v. The amortized cost per multi-insert is $O(\Delta n^2)$. This also pays for all deletions. The space is $O(n^3)$, but we reduce it to $O(\Delta n^2)$.

The algorithm uses the Even-Shiloach algorithm to maintains $2n$ deletions-only data structures for distances up to Δ. For each vertex v, we have one $Out(v)$ for distances from v, i.e. with v as source, and one $In(v)$ with distances to v, i.e. with v as a sink. For each pair (u, w) of vertices, $\mathsf{d}ist_v(u, w)$ is the distance from u to v in $In(v)$ plus the distance from v to w in $Out(v)$.

We start with a graph with no edges. Whenever we make a multi-insert of edges incident to v, we reinitialize the pair $(In(v), Out(v))$ at amortized cost $O(\Delta m)$. When an edge is deleted, it is deleted from all pairs $(In(v), Out(v))$ created after it was inserted.

Lemma 1. *If the distance from u to w is at most Δ, $\mathsf{d}ist(u,w) = \min_v \mathsf{d}ist_v(u,w)$.*

Proof. Obviously, $\mathsf{d}ist(u, w) \leq \mathsf{d}ist_v(u, w)$ for all v. Consider an arbitrary shortest path from u to w and let v be the last vertex updated on this path. Then all the edges of this path are contained in $(In(v), Out(v))$ so $\mathsf{d}ist_v(u, w) = \mathsf{d}ist(u, w)$.

Thus, our problem is to keep track of the vertex v minimizing $\mathsf{d}ist_v(u,w)$ for each pair (u,w) of vertices. King's idea is, for each possible distance $d \in \{1,...,\Delta\}$, to maintain the list $L(u,w,d)$ of vertices v with $\mathsf{d}ist_v(u,w) = d$.
The King algorithm: Whenever a vertex u increases its distance to v in $In(v)$, this increases all distances $\mathsf{d}ist_v(u,w) \leq \Delta$. If d and d' are the old and the new distance of v, v has to be moved from $L(u,w,d)$ to $L(u,w,d')$. This move takes constant time. Each vertex u increases its distance at most Δ times in $In(v)$, and each time, we have to perform a move between lists for each vertex w with $\mathsf{d}ist_v(u,w) \leq \Delta$. Thus, u pays a total cost of $O(\Delta n)$ for increases to its distance in $In(v)$. The cost for increases to its distance in $Out(v)$ is the same. Thus, for each pair $(In(v), Out(v))$, the total cost of distance increases is $O(\Delta n^2)$, which is attributed to the multi-insert creating $(In(v), Out(v))$.

To get the distance matrix, for each (u,w), we maintain the smallest d for $L(u,w,d) \neq \emptyset$. Then $d = \mathsf{d}ist(u,w)$. For any vertex v in $L(u,w,d)$, the pair $(In(v), Out(v))$ provides a shortest path from u to w. When doing a multi-insert, the above is trivially done by scanning all lists in $O(n^2\Delta)$ time. When doing deletions, our amortization argument allows the multi-inserts to pay for all distance increases, even if these are considered to be done one at the time. If $L(u,w,d)$ becomes empty because some vertex v is moved to $L(u,w,d')$, we can afford to scan all the lists $L(u,w,e)$, $e = d+1,...,d'$ by charging the cost to the multi-inserts.

Each pair $(In(v), Out(v))$ takes $O(m)$ space. For each (u,v,w), v is in $L(u,w,\mathsf{d}ist_v(u,w))$, so the total number of elements in the lists is $O(n^3)$. Moreover, there are $O(\Delta n^2) = O(n^3)$ lists. Hence, the total size of the data structure is $O(n^3)$.

3.1 Improving the Space

To reduce the space of the above data structure to $O(\Delta n^2)$, we need to address both the pairs $(In(v), Out(v))$ and the witness lists $L(u,w,d)$.

$(In(v), Out(v))$ For each pair $(In(v), Out(v))$, we wish to reduce the space to $O(n)$ using the trick from §2.1. However, that trick assumed for each vertex w, that we could traverse $in(w)$ in some fixed order. That is, if (x,w) was the witness of w in T_v, and if (x,w) was deleted, we could continue the traversal for witnesses from the successor of (x,w) in $in(w)$ knowing that the same edge would never be considered twice for the same distance $d_v(w)$.

To deal with the differing incidence lists, we spend $O(\Delta n^2)$, rather than $O(\Delta m)$, to maintain each pair $(In(v), Out(v))$. This does not affect our asymptotic cost, since we have already incurrred this cost for each multi-insert. Hence we can spend $O(n)$ time to traverse $in(w)$ in $Out(v)$.

We fix one arbitrary global ordering $\prec$ of all the vertices. For each vertex w its list $in(w)$ in $Out(v)$ will be ordered as $\prec$ orders its predecessors. However, $in(w)$ will be stored only implicitly. We keep a global $n \times n$ incidence matrix which for each $(x,y) \in V^2$ which stores its length $\ell(x,y)$ or ∞ if (x,y) is not an edge.

We traverse $in(w)$ in $Out(v)$ by going through all the vertices $x \in V$ in the order $\prec$ and keeping a pointer to our current location in the order. For each x, we check if (x, w) is an appropriate witness, i.e., if $\mathsf{d}ist_v(v, x)+\ell(x, w) = dist_v(v, w)$. For each edge (x, w) which serves as a witness, we keep a pointer back to v, so that when (x, w) or $\mathsf{d}ist_v(v, x)$ is changed, we know to look for another witness.

We only increment $\mathsf{d}ist_v(v, w)$ after the order is traversed. We only decrement $\mathsf{d}ist_v(v, w)$ when $(In(v), Out(v))$ is reinitialized. At botht times, we reset the pointer to the beginning of the order. Note that in this space-saving version, we may inadvertently add a witness which was created after the last time $Out(v)$ was initialized.

We maintain a shortest path tree of edge witnesses as described in Section 2.1, so that nodes affected by a distance increase can be easily determined.

As claimed, the total time to traverse the order is $O(n)$, giving the amortized cost of $O(\Delta n^2)$ for each pair $(In(v), Out(v))$. Our total space for all the pairs $(In(v), Out(v))$, including the global incidence matrix, is now $O(n^2)$.

***L*(*u*, *w*, *d*)** Instead of the lists $L(u, w, d)$ of vertices v with $\mathsf{d}ist_v(u, w) = d$, we maintain only the oldest witness v with $\mathsf{d}ist_v(u, w) \leq d$. This reduces the space to a constant per list, hence to to $O(\Delta n^2)$, as desired.

The switch from '=' to '$\leq$' is crucial to our amortization. The point is that if v is not a witness for $\mathsf{d}ist_v(u, w) \leq d$, it will not become so before $(In(v), Out(v))$ is replaced by a multi-insert.

We maintain a historical list of the vertices ordered according to dates. Note that this order has nothing to do with the order $\prec$ used previously. When we do a multi-insert of edges around v, we move v to the end of the historical list.

Suppose $d = \mathsf{d}ist_v(u, w)$ is increased. If v was the oldest witness of (u, w, d), we scan the historical list starting from the successor of v, stopping as soon as we find a younger vertex v' with $\mathsf{d}ist_{v'}(u, w) = d$. Then v' is the new oldest witness of (u, w, d), and otherwise, there is no witness. If $\mathsf{d}ist_v(u, w)$ was increased by more than one, we repeat the above scanning if v was also the oldest witness for $(u, w, d + 1)$. As soon as we reach some value e so that v was not the oldest witness for (u, w, e), we stop; if $v' \neq v$ is the oldest witness for (u, w, e), it is also an older witness for all (u, w, f) with $f > e$.

Since we always scan younger and younger witnesses, we see that a vertex v, between multi-inserts around v, can only be scanned once for each triple (u, w, d). So, the amortized cost of each multi-insert is $O(\Delta n^2)$. By getting rid of most pointers, we also reduce the constants hidden in the O-notation.

Below, we briefly describe how the above techniques are applied by King [9] to transitive closure, and leave the discussion of approximate and exact shortest paths to the full paper. We show that our space improvements remain valid for these applications.

3.2 Transitive Closure

King maintains the transitive closure of a digraph, in a manner similar to the classical method of repeated squaring of matrices. For $i = 0, 1, ..., h = \lceil \log n \rceil$,

we maintain a digraph G^i that is a subgraph of the transitive closure, and which is guaranteed to have an edge (v, w) if there is a dipath from v to w in G of length at most 2^i, i.e., $G^0 = G$, and G^h is the transitive closure.

For $i > 0$, and for each vertex $v \in V$, we maintain the pair $(In^{i+1}(v'), Out^{i+1}(v'))$, with $\Delta = 2$. For $i > 0$, the edges of G^{i+1} are defined to be all pairs (u, w) such that there is a path from u to w in $(In^i(v), Out^i(v))$ of length 3 or less.

Lemma 2 (9). *If a dipath P from u to w has length at most 2^{i+1} and the youngest vertex in the dipath is v, there is a path from u to w in $(In^{i+1}(v), Out^{i+1}(v))$ of length 3 or less.*

Proof. Let v' be a middle vertex in P. Assume v' is in $Out^{i+1}(v)$. Then $P[u, v]$, $P[v, v']$, $P[v', w]$ are all of length at most 2^i. Hence (u, v), (v, v') and (v', w) were all in G^i when $(In^{i+1}(v), Out^{i+1}(v))$ was last reinitialized.

To save space, we implement the $(In^{i+1}(v'), Out^{i+1}(v'))$ pairs as in the previous section, with incidence matrices for $i = 1, ..., h$. When a multi-insert of edges around v occurs, and G^0 has been updated, then for $i = 1, ..., h$, we reinitialize only $(In^{i+1}(v), Out^{i+1}(v))$ using all edges currently in G^i. A the multi-insert may create edges in G^i not incident to v which do not affect older pairs $(In^{i+1}(v'), Out^{i+1}(v'))$.

If an edge is deleted from G, it is deleted from the incidence matrix for G^0. For $i = 1, ..., h$, as edges in G^{i-1} are deleted, this may result in the destruction of paths in $(In^{i+1}(v'), Out^{i+1}(v'))$ which in turn may result in the deletion of edges in G^{i+1}.

Given any edge $(u, w) \in G^{i+1}$, to expand it to a path, we consider the witness v for $\mathsf{dist}_v^{i+1}(u, w) \leq 3$. Then $(In^{i+1}(v), In^{i+1}(v))$ provide us a path of length ≤ 3 from (u, w) using edges from G^i, and each of these edges can be expanded recursively.

To get simple paths, we combine with a linear time computation of strongly connected components [12]. For each strongly connected component, using an in-tree and an out-tree from an arbitrary vertex, we provide a simple path between any pair of vertices in the same strongly connected component. In linear time, we also label each vertex with the strongly connected component it is in. Now we only need to reconstruct segments between strongly connected components. So, if G^{i+1} provides us with a path of edges from G^i, we only expand edge (x, y) if x and y are in different connected components.

Since there are $\log n$ levels, we maintain the transitive closure in $O(n^2 \log n)$ amortized time per multi-insert and $O(n^2 \log n)$ space.

4 Demetrescu-Italiano Transitive Closure

The Demetrescu-Italiano transitive closure algorithm represents the graph as an adjacency matrix and achieves a $\log n$ speed-up by basing its algorithm on the static recursive scheme [8,10] for computing transitive closure, rather than

repeated squaring. The adjacency matrix of the graph is subdivided into four sub-matrices. The transitive closure can be shown to be a polynomial over these four sub-matrices. [4,3] show that the problem of updating a polynomial over Boolean matrices can be reduced to updating the product of degree 2 polynomials, with some errors allowed, as described below. Solving this problem in $O(n^2)$ space gives an $O(n^2)$ space bound for the whole algorithm.

Let M_1 and M_2 be two $n \times n$ Boolean matrices. In computing their product, there are two kinds of updates allowed to either matrix in which 0's are changed to 1's: one where any entry can be set to 1 and one where just entries in a particular row i and column i can be set to 1. The first is done by LazySet, the second by SetRow(i, M) and SetCol(i, M), respectively. In addition, 1's can be changed to 0's by ReSet. The "error" allowed is that in certain circumstances, the bits set by LazySet can be ignored.

When entries (x, y) in M_1 and (y, z) in M_2 are both 1, we call these entries a witness pair for (x, z), and y a witness for (x, z). A witness pair (x, z) cannot be ignored if its two entries (x, y) and (y, z) exist currently and they existed since a time when one of them was contained in a row or column updated by a SetRow or SetCol. That is, the problem is to maintain for each entry in the matrix product a witness pair (x, y, z) if one exists and was created at or before the most recently executed $SetRow(x, M_1)$, $SetCol(y, M_1)$, $SetRow(y, M_2)$, or $SetCol(z, M_2)$.

The goal is to spend no more than $O(n^2)$ time per SetCol and SetRow. Essentially, LazySet does nothing except change bits in M_1 and M_2; SetCol and SetRow do the work. The cost of ReSet is amortized against the operations which change the 0's to 1's.

The implementation of this in $O(n^3)$ space is straightforward, and involves the keeping of lists of relevant witnesses.

Our result: We can accomplish the multiplication of two matrices (with the allowable errors)more easily while maintaining witnesses. We do this with no change to the asymptotic time, maintaining a cost of $O(n^2)$ time per SetRow and SetCol, and space $O(n^2)$. We use the technique of the previous sections, of keeping a pointer to the next available witness. We keep:

– For each x, z, a pointer $P1(x, z)$ which is initially set to the first column of y in M_1 such that $M_1(x, y)$ AND $M_2(y, z)$; these two entries are a witness pair for (x, z)

– For each x, z, a pointer $P2(x, z)$ which is initially set to the first row y in M_2 such that $M_1(x, y)$ AND $M_2(y, z)$; these are a witness pair for (x, z).

– A queue of y 's ordered by order of last execution of either SetCol(y, M_1) or SetRow(y, M_2).

– For each (x, z), a pointer $P3(x, z)$ which is initially set to the first y in the queue such that $M_1(x, y)$ AND $M2(y, z)$

– Back pointers from the entries which are witnesses back to the (x, z) for which they are a witness.

To do updates:

When SetRow(x) in M_1 is executed, we reset $P1(x, z)$ to the beginning for all z.

When SetCol(z) in M_2 is executed, we reset $P2(x, z)$ to the beginning for all x. When $M_1(x, y)$ or $M_2(y, z)$is set to 0 during a ReSet:
– If $P1(x, z)$ points to y, then we move $P1(x, z)$ forward until another witness is found or the end of the row.
– If $P2(x, z)$ points to y then we move $P2(x, z)$ forward until another witness is found or the end of the column.
– If $P3(x, z)$ points to y, then we advance $P3(x, z)$ until another witness is found.
Running time analysis: Each time a pointer is set back to the beginning of a row or column, there may be another n checks at a cost of $O(1)$ each. $P3(x, z)$ is never set back but there may be an added cost of O(1) for every pair (x, z) for each queue change. We charge the SetCol or SetRow to cover this.

References

1. N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proceedings of the 33rd IEEE Annual Symposium on Foundations of Computer Science*, pages 417–426, 1992.
2. G. Brown and V. King. Space-efficent methods for maintaining shortest paths and transitive closure: Theory and practice, August 2000. Presentation at Workshop on Efficient Algorithms at Oberwolfach.
3. C. Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs.* PhD thesis, Dip. Informatica e Sistemistica Universita' di Roma "La Sapienza", 2000.
4. C. Demetrescu and G. Italiano. Fully dynamic transitive closure: Breaking through the $o(n^2)$ barrier. In *Proc. 41st IEEE Symp. on Foundations of Computer Science*, pages 381–389, 2000.
5. S. Even and Y. Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.
6. Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proc. 19th IEEE INFOCOM - The Conference on Computer Communications*, pages 519–528, 2000.
7. D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single-source shortest path problem. *ACM Jounal of Experimental Algorithmics*, 3, article 5, 1998.
8. M. E. Furman. Application of a method of rapid multiplication of matrices to the problem of finding the transitive closure of a graph. *Soviet Math. Dokl.*, 11(5):1250, 1970.
9. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 81–89, 1999.
10. I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.
11. G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Jounal of Algorithms*, 21(2):267–305, 1996.
12. R. E. Tarjan. Deep-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.
13. U. Zwick. All pairs shortest paths in weighted directed graphs-exact and almost exact algorithms. In *Proceedings of the 39rd IEEE Annual Symposium on Foundations of Computer Science*, pages 310–319, 1998.