Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs

Valerie King
Department of Computer Science
University of Victoria
P.O. Box 3055
Victoria, BC, Canada
V8W 3P6
email: val@csr.uvic.ca *

Abstract

This paper presents the first fully dynamic algorithms for maintaining all-pairs shortest paths in digraphs with positive integer weights less than b. For approximate shortest paths with an error factor of $(2+\epsilon)$, for any positive constant ϵ, the amortized update time is $O(n^2 \log^2 n/ \log\log n)$; for an error factor of $(1+\epsilon)$ the amortized update time is $O(n^2 \log^3(bn)/\epsilon^2)$. For exact shortest paths the amortized update time is $O(n^{2.5}\sqrt{b \log n})$. Query time for exact and approximate shortest distances is $O(1)$; exact and approximate paths can be generated in time proportional to their lengths.

Also presented is a fully dynamic transitive closure algorithm with update time $O(n^2 \log n)$ and query time $O(1)$. The previously known fully dynamic transitive closure algorithm with fast query time has one-sided error and update time $O(n^{2.28})$.

The algorithms use simple data structures, and are deterministic.

1. The problem

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 from scratch (as performed by the fastest "static" algorithm).

We give fully dynamic algorithms for the following problems on directed graphs: transitive closure, and approximate and exact all-pairs shortest paths, on graphs with edge weights which are positive integers bounded by b. Another consequence of our work is an all-pairs shortest path algorithm for edge deletions only.

Our data structures implements the following update operations:

- $insert(E_v)$: inserts a set of edges incident to the same vertex v.
- $delete(E')$: deletes any arbitrary subset E' of edges currently in the graph.

For transitive closure, the query operation is of the form:

- (A) Is there a path from u to w in the current graph?

For all-pairs shortest paths, the queries are of the form:

- (A) What is the shortest distance from u to w?
- (B) Generate a shortest path from u to w.

For approximate all-pairs shortest paths, the queries are of the form:

- (A) What is an upper bound on the shortest distance from u to w which is within a factor of $(1+\epsilon)$ of the shortest distance?
- (B) Generate a path from u to w whose distance is within a factor of $(1+\epsilon)$ of the shortest distance.

Let n be the number of vertices in the graph and m be the number of edges initially in the graph. All update times are amortized over a worst case sequence of operations of length $\Omega(m/n)$. All query times are proportional to the length of the output; in particular, type (A) queries run in $O(1)$ time. All algorithms are deterministic.

*This work was done while the author was visiting U.C. Berkeley and ICSI in Berkeley.

For transitive closure, the amortized update time is $O(n^2 \log n)$ per update. The only previously known fully dynamic transitive closure algorithm [10] with fast query time is randomized with one-sided error and has update time $O(n^{2+\frac{\omega-2}{\omega-1}})$ where ω is the usual exponent of matrix multiplication. If the method of Coppersmith and Winograd [3] is used, this is $O(n^{2.28})$.

For exact all-pairs shortest paths, the amortized update time is $O(n^{2.5}\sqrt{b \log n})$. There are no previously known fully dynamic algorithms for general graphs. The fastest static algorithm [15] for exact distances in a directed graph runs in time $O(b^{.681} n^{2+\mu})$ where μ satisfies the equation $\omega(1, \mu, 1) = 1 + 2\mu$ and $\omega(1, \mu, 1)$ is the exponent of the multiplication of an $n \times n^\mu$ matrix by an $n^\mu \times n$ matrix. The smallest value known is $\mu = .575$, if the method of [3] is used.

The approximate all-pairs shortest paths algorithms maintain paths whose distances are bounded above by the product of the error factor and the actual shortest distance. The amortized update time in graphs with unweighted edges and error factor $2 + \epsilon$ for any positive ϵ is $O(n^2 \log^2 n / \log \log n)$. For weighted edges and error factor $1 + \epsilon$, the amortized update time is $O(n^2 \log^3(bn)/\epsilon^2)$. The fastest static algorithm for approximate all-pairs shortest paths with error factor $1 + \epsilon$ has running time $\tilde{O}((n^\omega/\epsilon) \log(b/\epsilon))$.

For all-pairs shortest paths with edge deletions only, the total cost is $O(mn^2 b)$, or $O(n^2 b)$ per deletion if there are $\Omega(m)$ deletions.

1.1. The techniques: an overview

In Section 2, we first show that a single-source shortest path tree up to distance d can be maintained during a sequence of any number of edge deletions in time $O(md)$, in a graph with positive integer edge weights.

For the exact all-pairs shortest path algorithm with edge deletions only, we simply maintain a forest of n single-source shortest path trees of depth nb.

We then show how to maintain exact all-pairs shortest paths with insertions and deletions for distance up to d. For each vertex v, we maintain a single-source shortest path tree of depth d of vertices which reach v (In_v) and another tree of vertices which are reached by v (Out_v). We call this a "forest of In and Out trees of depth d". For each pair of vertices u, w, we keep a count, $count(u, w, j)$, of the number of In_v and Out_v such that there is a path from u to v to w of length j. For each operation $insert(E_v)$ we rebuild In_v and Out_v.

In Section 3, we show how to maintain the transitive closure. We keep a hierarchy of $\lg n$ forests of In and Out trees of depth 2, where the edges used to construct a forest on one level depend on the paths in the forest of the previous level. In Section 4, we maintain approximate all-pairs shortest paths by keeping a hierarchy of $O(\log n / \log \log n)$ forests of In and Out trees of depth $\lg n$ or more depending on the error factor.

We maintain exact shortest paths in Section 4, by maintaining one forest of In and Out trees of depth $\sqrt{nb}$. After each update, we stitch the paths in these trees together to generate the shortest paths.

1.2. Related Work

In 1981, Even and Shiloach showed how to maintain a breadthfirst search tree, which could process any number of deletions in time $O(mn)$ for m the number of edges, and n the number of vertices. In 1995, Henzinger and King [7] recognized that this data structure could be adapted to directed graphs, to maintain reachability from a single vertex for distances (for unweighted edges) of up to d, for any number of edge deletions, in time $O(md)$. They use a forest of such data structures as part of a fully dynamic transitive closure algorithm with amortized update time $\tilde{O}(nm^{\omega-1/\omega}) = \tilde{O}(nm^{0.58})$ for $\omega = 2.38$. Like our exact shortest paths algorithm, that algorithm involves stitching together short paths, but uses fast matrix mulitplication to do so. It is randomized, with one-sided error, and has a slow query time of $O(n/\log n)$.

In 1999, King and Sagert designed a dynamic transitive closure algorithm with $O(1)$ query time, and also one-sided error. They maintain a count of the number of distinct paths for each pair of vertices, modular a random prime, for acyclic graphs. Non-acyclic graphs are reduced to the acyclic case. The cost per update is $O(n^2\alpha)$ where $\alpha = \min\{\text{max size of a strongly connected component}, n^{.28}\}$. An advantage of this algorithm is that if the size of the strongly connected component is no greater than $n^{.28}$, then its worst case update time matches its amortized update time. Using the techniques of King and Sagert, Kapron and King have devised an exact shortest path algorithm for unweighted graphs for distances up to d which has worst case update time of $O(n^2 2^d)$.

Here, we incorporate the King-Sagert idea of keeping a count with the forest data structure of Henzinger-King.

The best amortized update times for partially dynamic problems are as follows: for maintaining transitive closure with insertions, $O(n)$[8, 11]; with deletions, $O(m)$ [11]; with deletions in acyclic graphs, $O(n)$ [9]; for maintaining shortest paths with insertions and positive integer weights no greater than b, $O(nb \log n)$ [1].

Klein *et.al.* give a fully dynamic algorithm for the all-pairs shortest path problem on planar graphs. If the sum of the absolute values of the edge-lengths is D then the time per operation is $O(n^{9/7} \log nD)$.

G. Ramalingham and T. Reps consider the problem of maintaining shortest paths in a different model of complexity in which running time is given in terms of a parameter different from input size [12, 14]. Their algorithm for maintaining a single source shortest path is similar to ours for the short distances, deletions-only case. It has been experimentally analyzed by D. Frigioni *et. al.* [4].

Lower bounds for dynamic transitive closure and shortest paths problems have been considered by several researchers, but in general, the models assumed are too restrictive to imply a lower bound for our algorithms. See [14] for a discussion of these works. The only relevant lower bound is the $\Omega(\log n/\log\log n)$ bound for dynamic connectivity for undirected graphs in the cell probe model by Henzinger and Fredman [6].

1.3. Applications

In a 1990 survey of the application of graph algorithms to data bases, Yannakakis discusses the application of partially dynamic transitive closure to problems in data bases. The "regular path problem" in data bases corresponds to computing transitive closure in a directed graph where each relation in the data base corresponds to k sets of k edges with a common endpoint in a directed graph. Here, k is the number of states of a finite automaton for a regular language. With the new fully dynamic transitive closure algorithm presented here, each new relation can be inserted in $O(k(nk)^2\log(kn))$ time. Each deletion of a relation can be accomplished in $O((nk)^2\log(kn))$ time. See also [2]. In [5], the question of maintaining distances between objects in a very large data base is posed.

Reps and others have investigated the application of dynamic shortest paths and transitive closure algorithms to data flow analysis and compilers. See [13].

2. Exact all-pairs shortest paths for small distances

2.1. Single-source shortest path trees of depth d–deletions-only

In this section, we give a deletions-only algorithm for maintaining a single-source shortest path tree from a vertex s of depth up to d, in a graph whose edges weights are positive integers.

When a tree edge (u,v) is deleted, the algorithm mimics Dijkstra's single source shortest path algorithm to reinsert the vertices in the subtree rooted at v. That is, a vertex v is added to the shortest path tree when its distance to s is minimal among the vertices not in the tree. The goal is to spend no more time than that proportional to 1 + the sum of the degrees of those vertices whose distance from s have been changed by the deletion.

We call a vertex is *changed* if we've determined that the distance from s to w has increased. A vertex is *settled* if it is joined to the source by tree edges. A vertex is *uncertain* if it has been examined but it is not yet determined if its distance from s has changed.

For each vertex w, for distances no greater than d, $l(w)$ denotes the distance of w from s before the deletion, until $l(w)$ is revised, in which case it is revised to the distance of w from s after the deletion. If the distance is greater than d, then $l(w)=\infty$.

Each vertex w maintains a set $predlist(w)$ containing all vertices z which are either settled or uncertain and such that $(w,z)\in E$ and (w,z) is not a tree edge. For each w, we maintain $d(w)=\min_{z\in predlist(w)} l(z)+weight(z,w)$ and let $minpred(w)$ be a vertex z which minimizes the expression.

The data structure is initialized by computing a shortest path tree and forming $predlist(w)$ for each vertex w.

When a tree edge (u,v) is deleted from the graph, (u,v) is removed from the shortest path tree. Vertex v becomes uncertain. All other vertices in v's subtree are unexamined and not in the tree.

If any vertex w is uncertain, w is stored in a heap H with $key(w)=l(w)$, the old distance of w from s. If w is changed and unsettled, then w is also stored in H with $key(w)=d(w)$. H is empty between runs of Delete.

If $key(w)$ is minimal over all keys stored in H and w is uncertain, the algorithm decides if w has changed, based on whether $d(w)=key(w)$ (i.e. $l(w)$). If w is not changed, it can be settled. When w is settled, all its descendants (which are unexamined) are automatically reinserted into the tree. (The edges between them are intact.) If $d(w)\neq key(w)$ then w is changed, and is unsettled. If $key(w)$ is minimal over all keys in H and w is changed, this means that $key(w)=d(w)$ and w can be settled.

Delete(u,v)
Make_uncertain(v);
Repeat until H is empty.
  $w\leftarrow delete\_min(H)$
  if $key(w)>d$ then for all w in H, $l(w)\leftarrow\infty$; STOP.
  if $uncertain(w)$ then do
    if $key(w)=d(w)$ then Settle(w);
    else do
      Make_changed(w);
      for each tree edge (w,z) do
        remove (w,z) from tree;
        Make_uncertain(z);
  else Settle(w).

Make_changed(w)
For each nontree edge $(w,z)\in E$, remove w from $predlist(z)$.
Add w to H with $key(w)\leftarrow d(w)$.

$uncertain(w) \leftarrow false$

Settle(w)
Add edge $(minpred(w), w)$ to tree.
Remove $minpred(w)$ from $predlist(w)$.
If $not(uncertain(w))$ then do
for all (w, z), insert w into $predlist(z)$.
$l(w) \leftarrow d(w)$.

Make_uncertain(z)
$uncertain(z) \leftarrow true$; add z to H with $key(z) = l(z)$.

Proof of correctness: The proof follows that of Dijkstra's algorithm, with some minor differences. We note that the only vertices that need to be examined are v and those vertices in v's subtree which have an ancestor which has changed. We also note that to follow Dijkstra's algorithm, we wish to choose the *settled* vertex in $predlist(w)$ which minimizes $d(w)$. But the vertices in $predlist$ are not necessarily settled; they may be uncertain or unexamined. Hence one needs to argue that the minimality of $key(w)$ when $w \leftarrow delete\_min(H)$ implies that $minpred(w)$ is settled. We leave the details to the reader.

Analysis: Each $predlist$ and H may be implemented as a heap of up to d keys, one for each distance represented in the list. Hence, Make_certain runs in time $O(\log d)$, Settle(w) runs in $O(\log d)$ if w is uncertain, and $O(\log d *$ degree of $w)$ if w is changed. Make_changed runs in time $O(\log d *$ degree of $w)$.

The cost of running Delete is then $O(\log d)$ if no distances change (but the total number of edges stored is reduced by 1), or $O(degree(w) \log d)$ for each w such that $l(w)$ is increased. Thus the worst case time for the deletion of a single edge is no more than $O(\sum_v degree(v) \log d)$ or $O(m \log d)$.

Over a sequence of deletions, $l(w)$ can increase no more than d times, for a total charge of $O(d * degree(w) \log d)$ or $(dm \log d)$ for all vertices. Also no more than $O(m \log d)$ is incurred by those deletions where no distances change. This gives a total cost of $O(md \log d)$.

2.1.1 A faster implementation

We eliminate the use of heaps and save a factor of $\log d$, in the amortized time. We sketch that variation here. Each $predlist(w)$ is represented by an array such that for $i = 1, ..., d$, $predlist(w)[i]$ is the set of all settled and uncertain vertices z such that $l(z) + weight(z, w) = i$. The heap H is replaced by the set of all changed, unsettled vertices plus an array H' such that for $i = 1, ..., d$, $H'[i]$ is a set of uncertain vertices w with $l(w) = i$. Note that in the algorithm, the sequence of minimum keys extracted from the heap is nondecreasing. We keep a pointer L to the next possible value for the minimum key. When Delete(u, v) is run, L is initially set to $l(v)$.

For each value of L, each changed and unsettled vertex w is examined to see if it can be settled or not, determined by whether $predlist(w)[L]$ is nonempty or not. After these vertices are examined, the uncertain vertices w' with $l(w') = L$ are then either settled or become changed and unsettled. While $L \leq d$ and there remain unsettled vertices, L is incremented and the process is repeated.

Analysis of faster implementation This implementation reduces the costs of Make_certain, Settle, and Make_changed by a factor of $\log d$. If no distances from the source are changed, the cost of Delete is $O(1)$.

If there is at least one vertex whose distance from the source s has changed, then there are two costs to consider: the cost charged to each changed vertex and the cost of incrementing L. L is incremented until all uncertain vertices are settled, up to $d - 1$ times. This introduces an extra additive cost of $O(d)$ for each run of Delete. The total cost of these increments is $O(md)$ since there are no more than m deletions.

Since each changed vertex is examined each time L is incremented until the vertex is settled, the cost charged to each changed vertex w in this implementation is is $O(\Delta_w + degree(w))$ where $\Delta_w =$ change in distance of w from s. Hence the worst case cost per operation becomes $O(nd+m)$. However the total cost over all deletions remains $O(md)$ since the cost per vertex w is maximized when its distance increases by only one each time it is increased, for a maximum cost of $O(d * degree(w))$. Thus the total cost over all deletions is $O(\sum_w (degree(w) * d)) = O(md)$.

In later sections, we will assume that the faster implementation is used. Its only drawbacks are its worst case performance and extra costs incurred for the operation of increasing an edge weight, described below.

2.1.2 Handling edge weight increases

It is not hard to modify the algorithm to handle an increase in edge weight. Suppose $weight(u, v)$ is increased. Then make (u, v) a nontree edge by inserting u into $predlist(v)$ and run $Delete(u, v)$.

If the first implementation method is used, then the cost of Delete is, as before, $O(\log d)$ plus $O(degree(w) \log d)$ for each w such that $l(w)$ is increased. Since each edge can be increased no more than $b \leq d$ times, no more than mb calls to Delete can be made for the purpose of increasing edge weights. Hence adding this operation results in a cost of $O(mb \log d + md)$ which leaves the asymptotic total time unchanged.

If the second implementation method is used and no distances from s are increased then the cost of the edge increase is $O(1)$. If an increase to an edge weight causes some vertex w to increase its distance from the source, then, as described above, there is an extra additive cost of $O(d)$. This can occur no more than min$\{inc, nd\}$ times, where inc is the number of edges whose weights have increased.

Hence the total cost is $O(md + \min\{inc * d, nd^2\})$. So the operations to increase an edge weight could increase the asymtotic value of the total cost if $nd > m$ and $inc > m$.

In the later sections, we do not discuss the special operation of increasing edge weights, but the technique and analysis follow easily from what is described here. We leave this to the reader.

2.1.3 All-pairs shortest paths, deletions only

We maintain all-pairs shortest paths in a graph whose weights are positive integers less than b and where updates are restricted to edge deletions and edge weight increases. It suffices to maintain a single source shortest path tree of depth $d = nb$ for each vertex. Then the shortest path from a vertex u to w is given by $l(w)$ in the tree for u. The total cost of processing the deletions is $O(mn^2b)$. Hence, if there are $\Omega(m)$ update operations, amortized cost is $O(n^2b)$ per update operation.

2.2. A forest of In and Out trees

We show how to maintain all-pairs shortest paths and distances for pairs of vertices no more than distance d apart.

During the algorithm, we maintain single source shortest path trees In_v and Out_v of depth no greater than d for each vertex $v \in V$. The first is for vertices which reach v and the second is for vertices which are reachable from v. Let $In_v(u)$ be the distance from u to v in the tree In_v and $Out_v(u)$ be the distance from v to u in tree Out_v. We define $In_v(v) = Out_v(v) = 0$ for all v.

For each pair of vertices u, w and depth $k = 1, 2, ...d$, we keep $count(u, w, k)$ equal to the number of v such that $In_v(u) + Out_v(w) = k$ and a list $list(u, w, k)$ of these vertices. For each pair of vertices u, w, we keep $D(u, w)$ set to the minimum k such that $count(i, j, k)$ is positive. If there is no such k, then $D(u, w) = \infty$.

To initialize, it suffices to maintain only Out_v trees for each vertex v. Then set $count(v, w, k) = 1$ if $Out_v(w) = k$; else $count(v, w, k) = 0$.

- *To do $insert(E_v)$:* Remove In_v and Out_v if they exist. Build and maintain a new In_v and a new Out_v, using the set of edges in the current graph G. For each u, w, k, adjust $count(u, w, k)$, $list(u, w, k)$ and $D(u, w)$ accordingly. We call vertex v the *center of $insert(E_v)$*.

- *To do $delete(E')$*: Process each deleted edge in every In and Out tree data structure which contains it. Each time a vertex u moves down in a tree, we may need to adjust the $count(u, w, k)$, $count(w, u, k)$, $list(u, w, k)$, and $D(u, w)$.

- *To answer the query:* "What is the distance from u to w?", return $D(u, w)$.

- *To return the shortest path from vertex u to w:* Let v be a vertex in $list(i, j, D(i, j))$. The path from u to v in $In_v(u)$ and from v to w in $Out_v(w)$ is a shortest path.

Analysis and implementation details: We can initialize and maintain each single source shortest path tree in $O(md)$. For the original forest of Out trees, we can initialize $count$, $list$ and D in $O(n)$ time per tree. The total cost to maintain these is $O(nd)$ per tree, since $Out_v(w)$ can increase at most d times, for any v and w. Thus the total cost for initializing and maintaining the initial forest of Out trees is $O(nmd + n^2d) = O(nmd)$.

For each $insert(E_v)$, when In_v and Out_v trees are destroyed and recreated, we can reset $count$, $list$ and D in $O(n^2)$ and maintain each tree in $O(md)$ time. Each time a vertex moves down in a tree, In_v for example, $count$ and $list$ must be revised, for each vertex in Out_v, or up to $n - 1$ vertices. Hence if every vertex moves down d times in each tree, the maximum cost of maintaining the count and the lists over all modifications to the single-souce shortest path tree is $O(n^2d)$. This cost may be charged to the insert operation which constructs the single-source shortest path trees. Thus the total cost per insert is $O(n^2d + md) = O(n^2d)$.

Determining $D(u, w)$ takes no longer than $O(d)$ time for each pair of vertices u, w. We can charge $O(n^2d)$ to each $delete$ operation for the cost of updating the D's. Alternatively, for each u, w, we can maintain the k's such that $count(u, w, k) > 0$ in a heap to extract the minimum in $O(\log d)$ time with each change to the $count$. This gives an additional $\log d$ factor but reduces the cost of a deletion.

The total cost of a sequence with del deletions, ins insertions and q queries is $O(nmd + (del + ins)n^2d + q)$ or $O((nmd \log d + ins * n^2d \log d + q)$, depending on the implementation. The first implementation has amortized time $O(n^2d)$ per update operation in a sequence of length $\Omega(m/n)$. The second has amortized time $O(n^2d \log d)$ per insertion if there are $\Omega(m/n)$ insertions and $O(nd \log d)$ per deletion, if $del + ins * n = \Omega(m)$.

Proof of correctness:

Lemma 2.1 *The algorithm maintains the following invariant:*

Invariant: *Let p be a shortest path from u to w of length $d' \leq d$. Let v be the vertex in p which was most recently a* *center of an insertion. Then there is a path from u to v in In_v and from v to w in Out_v such that $In_v(u) + Out_v(w) = d'$.*

Proof: Since the edges in p from u to v and from v to w were present in the graph at the time In_v and Out_v were most recently created, they are included in these data structures. Also, only edges currently in the graph are in these data structures, since edges which have been deleted from G since In_v and Out_v were built have been deleted from these data structures. Since In_v and Out_v maintain shortest paths from each vertex into v and to each vertex out of v, then in particular, the sum of distance from u to v and from v to u is d'.

It follows from the invariant and the accurate updating of $count$, $list$ and D that:

Theorem 2.2 *For any pair of vertices $u, w \in V$, if the distance from u to w is no greater than d, then the distance from u to w is given by the minimum k such that $count(i, j, k) > 0$, i.e., $D(u, w)$.*

3. Transitive closure

We maintain $k = \lceil \lg n \rceil$ forests $F^1, F^2, ..., F^k$ where each F^i contains a pair of breadthfirst search trees In_v^i and Out_v^i of depth 2 for each vertex $v \in V$.

We define $count^0(u, w) = 1$ if $(u, w) \in E$ and 0 otherwise. For all pairs of vertices u, w, and each forest F^i we maintain

- $count^i(u, w)$: the number of vertices v such that $u \in In_v^i$ and $w \in Out_v^i$;
- $list^i(u, w)$: the set of vertices v such that $u \in In_v^i$ and $w \in Out_v^i$.

F^1 is the forest of In and Out trees of depth 2, constructed as in the previous section for unweighted edges, for the current graph $G = (V, E)$. We maintain $E^i = \{(u, w) \mid count^{i-1}(u, w) > 0\}$. Initially, F^i is the forest of breadthfirst search trees of depth 2 constructed for the graph (V, E^i).

Note also that $count^i(u, w)$ is used, rather than $count^i(u, w, k)$. Different values of k are not distinguished and we keep the count for depths up to 4 which is greater than the depth of the breadthfirst search trees.

In the routines below, $insert(E_v, i)$ and $delete(E', i)$ are defined as in the previous section, with the modified definition of count, for the forest F^i. That is, $insert(E_v, i)$ adds E_v to E_i and uses the set of edges E^i to construct new trees In_v^i and Out_v^i in F^i; $delete(E', i)$ deletes edges in E' from the data structures for F^i.

To insert a set of edges incident to a vertex v:

InsertTC(E_v)
for $i = 1, ..., k$ do
  $insert(E_v, i)$;
  $E_v \leftarrow \{(u, v) \in E^i\} \cup \{(v, u) \in E^i\}$.

Note that when executing $insertTC(E_v)$, we only rebuild In_v^i and Out_v^i even though it is possible that inserting edges incident to v may have made $count^{i-1}(u, w)$ positive for some vertices $u, w \neq v$. Thus, after the initialization, it may no longer be the case that F^i is the same as the forest of breadthfirst search trees of depth 2 that we would have constructed and maintained for the graph (V, E^i).

To delete any set of edges E':

DeleteTC(E')
for $i = 1, ..., k$ do
  $delete(E', i)$;
  $E' \leftarrow \{(x, y) \mid count^i(x, y)$ changed from$positive\ \ to\ \ 0\}$

Proof of correctness: Let $dist(u, w)$ denote the shortest distance from u to w in the current graph G. We first show:

Lemma 3.1 *The following invariant holds for $i = 0, ..., k$:*
Invariant(i): *For all vertices u and w, the $count^i(u, w) > 0$ if $dist(u, w) \leq 2^i$, and only if there is a path from u to w.*

It follows that:

Theorem 3.2 *For all $u, w \in V$, $count^k(u, w) > 0$ iff there is a path from u to w.*

Proof of Lemma: The proof is by induction on i. For $i = 0$, we have $count^0(u, w) > 0$ iff there is an edge $(u, w) \in E$. Let us assume both directions of the invariant are true for $i - 1$.

Suppose there is path p from u to w of length $l \leq 2^i$. Let v be the most recent center of an insertion in p. Let x be the midpoint of p. Assume x lies between u and v. (If x lies between v and w, the argument is similar.) Then, $dist(u, x) \leq 2^{i-1}$, $dist(x, v) \leq 2^{i-1}$ and $dist(v, w) \leq 2^{i-1}$. Since v is the most recent center of an insertion, these inequalities were true when In_v and Out_v were most recently built, and have been true since that time.

Hence, by induction, $count^{i-1}(u, x)$, $count^{i-1}(x, v)$ and $count^{i-1}(v, w)$ were all positive when In_v and Out_v were built, and therefore (u, x), (x, v), and, (v, w) are edges in these data structures. Hence, $In_v(u) \leq 2$ and $Out_v(w) \leq 1$, and $count^i(u, w) > 0$. This concludes one direction of the proof.

Suppose $count^i(u, w) > 0$ but there is no path from u to w. But $count^i(u, w) > 0$ implies that there is some v such that $In_v^i(u) + Out_v^i(w) \leq 4$. But each edge (x, y) in In_v and Out_v is inserted only if $count^{i-1}(x, y) > 0$ and is deleted if $count^{i-1}(x, y)$ becomes 0. Hence $count^{i-1}(x, y) > 0$. By our induction assumption, each edge represents a path, hence there is a path from u to w consisting of their concatenation.

Analysis: From Section 2.2, we see that the cost of maintaining an initial forest is $O(nmd)$ and a total of $O(knmd)$ for maintain all k initial forests. Each update operation requires an update operation to each forest, at a cost of $O(n^2 d)$ per update operation or a total of $O(kn^2 d)$. Hence, the cost of per update is $O(n^2 dk)$ for a sequence of length $\Omega(m/n)$. For $k = \lceil \lg n \rceil$ and $d = 2$, this is $O(n^2 \log n)$.

4. Approximate shortest path

We give an approximate shortest path algorithm in directed graphs, first for graphs with unweighted edges.

Let $d = \lg n$ and $k = \lg n / \lg\lg n$. We initialize and maintain $k = \log n / \log d$ forests $F^1, F^2, ..., F^k$ of breadth-first search trees In_v^i and Out_v^i of depth d for each vertex $v \in V$.

For all pairs of vertices $u, w, j = 1, ..., d$, and each forest F^i we maintain

- $count^i(u, w, j)$: the number of vertices v such that $In_v^i(u) + Out_v^i(w) = j$;
- $E^i = \{(u, w) \mid count^{i-1}(u, w, j) > 0$ for any $j\}$; ;
- $list^i(u, w, j)$: $\{v \mid In_v^i(u) + Out_v^i(w) = j\}$;
- $approxdist(u, w)$: the pair (i', j') such that i' is the minimal i such that $(u, w) \in E^i$ and j' is the minimal j such that $count^{i'}(u, w, j) > 0$.

For each vertex, In_v^1 and Out_v^1 are the breadthfirst search trees constructed in the previous section, for depth d, for the graph $G = (V, E)$.

We perform updates and answer queries as follows.

- To insert a set of edges E_v incident to a vertex v or delete an arbitrary set of edges E', we use the routines $InsertTC(E_v)$ and $DeleteTC(E')$ with the definitions of $count$ and $list$ as defined above, and also maintain $approxdist$.
- To answer a query: What is the approximate shortest path from u to w?, we output jd^{i-1} where $(i, j) = approxdist(u, w)$.
- To generate an approximate shortest path from u to w: Let $(i, j) = approxdist(u, w)$. We choose a vertex v from $list^i(u, w, j)$ and recursively determine approximate shortest paths for each edge on the path from u to v in $In_v^i(u)$ and each edge on the path from v to w in $OUt_v^i(w)$.

Proof of correctness: We first show the following.

Lemma 4.1 *Let $dist(u, w)$ be the shortest distance from u to w in the current graph. The following invariant holds:*
Invariant(i):
1. For each edge (u, w) in E^i, $dist(u, w) \leq d^{i-1}$.
2. If there is a path of length j from u to w in F^i then $dist(u, w) \leq jd^{i-1}$.
3. If $dist(u, w) \leq (d-1)^{i-1}$ then (u, w) is an edge in E^i.
4. For $j = 2, 3, 4, ..., d$, if $dist(u, w) \leq (j-1)(d-1)^{i-1}$ then there is a path from u to w of length no greater than j in F^i.

Proof: We prove Invariant (1) is by induction on i. It is straightforward and is left to the reader.

Invariant (2) follows easily from (1) and is left to the reader.

Invariant (3) is true for $i = 0$. We assume it is true for i and show it is true for $i + 1$. Let v be the most recent center of insertion in the path of length no greater than $(d-1)^i$. We can partition the path from u to v and the path from v to w into segments of length $(d-1)^{i-1}$ with no more than two segments of length less than $(d-1)^{i-1}$, one on either side of v. Either there are $d-1$ segments of length $(d-1)^{i-1}$ or there are $d-2$ segments of length $(d-1)^{i-1}$ and two segments with fewer edges. In either case, there are no more than d total segments. Since each segment has length no greater than $(d-1)^{i-1}$ then by the induction assumption, each segment is represented by an edge in E^i, and was represented by an edge in E^i when In_v^i and Out_v^i were constructed. Then $In_v^i(u) + Out_v^i(w) \leq d$ and hence (u, w) is represented by an edge in E^{i+1}.

The proof of Invariant (4) is similar to the proof of Invariant (3) and is left to the reader.

The invariant implies the following theorem.

Theorem 4.2 *Let $approxdist(u, w) = (i, j)$. If $j = 1$ then $(d-1)^{i-1} \leq dist(u, w) \leq d^{i-1}$; and if $j \geq 2$, then $(j-1)(d-1)^{i-1} \leq dist(u, w) \leq jd^{i-1}$. I.e., jd^{i-1} is within a factor of $2[d/(d-1)]^{k+1}$ of the actual distance, for any $i \leq k$.*

To bound every length, we must have $(d-1)^{k+1} \geq n$, or $k \geq \log n / \log(d-2) + 1$. Choosing $d = \lg n$ and $k = \lg n / \lg\lg n$, we get a error less than $2 + \epsilon$ for any positive constant ϵ.

Analysis: As in the analysis of the transitive closure algorithm we see that the cost of maintaining all initial forests is $O(knmd)$. Each update operation requires $O(kn^2 d)$. Maintaining $approxdist$ for every pair of vertices can be performed $O(n^2 k)$. Hence, the cost of per update is $O(n^2 dk)$ for a sequence of length $\Omega(m/n)$, or $O(n^2 \log^2 n / \log\log n)$ when $d = \lg n$ and $k = \lg n / \log\log n$.

4.1. Reducing the error

We sketch the technique for lowering the error to $1 + \epsilon$ for any $\epsilon < 1$.

The error is worst when j is a small value greater than 1. To reduce this error, we can add a certain redundancy, so that when j is small, the path of length j is also represented at a lower level in the hierarchy of forests.

To do this, we use trees of distance d^2. As before, $E^i = \{(u, w) \mid count^{i-1}(u, w, j) > 0 \text{ for any } j \leq d\}$. However, we define $weight(u, w) = j'$ where j' is the minimal over all $j \leq d$ such that $count^i(u, w, j) > 0$.

The invariants follow as before, where length of a path refers to the sum of the weights of the edges in the path. The only difference is that if $approxdist(u, w) = (i, j)$, it must be the case that either $i = 1$ or $j > d$; otherwise there is a path from u to w in F^{i-1}. In the first case, j is the exact distance. In the second case, the error factor is no greater than $(d/(d-1))^{i+1}$. For $d = 2\lg n/\ln(1+\epsilon)$, since $i \leq \log n/\log d$, the error factor is less than $1+\epsilon$. The update time is increased by a factor of d to $O(n^2 d^2 k)$ since trees of depth d^2 are maintained. Setting $k = \lg n$ and substituting in for k and d, the update time is $O(n^2 \log^3 n/\epsilon^2)$.

4.1.1 Graphs with weighted edges

If $weight(u, w)$ is between $j - 1$ and $j(d^i)$, for $d^2 \geq j > d$, represent (u, w) by an edge weighted jd in E^i. If $weight(u, w) < d$ then add an edge weighted j into E^1.

Furthermore, since the shortest distance may be bn where b is the maximum weight of any edge, then we need to increase the the number k of forests to $\Omega(\log(nb)/\log d)$. Choosing $d = O(\log(bn)/\log(1 + \epsilon))$ gives the desired error. Setting $k = \lg bn$ and substituting into $O(n^2 d^2 k)$ gives an update time of $O(n^2 \log^3(bn)/\log^2(1 + \epsilon))$.

5. Exact shortest paths for arbitrarily long paths

For graphs with unweighted edges, we construct and maintain a forest F of In and Out breadthfirst search trees of depth d. We refer to v as the root of In_v and Out_v.

A subset $S \subset V$ is a *blocker* for F if every directed path starting or ending at the root, i.e., in an Out tree or an In tree respectively, of exactly length d contains a vertex in S which is distinct from the root.

Lemma 5.1 *Let S be a blocker for F. Then for all $u, w \in V$ such that w is reachable from u, there is a shortest path p from u to w which may be partitioned into consecutive subpaths $(u, s_1), (s_1, s_2), ..., (s_r, w)$ (where (x, y) denotes a subpath from vertex x to vertex y) with the following properties: (1) $s_1, ..., s_r \in S$; and (2) For each subpath (x, y), there is some v such that $x \in In_v$, $y \in Out_v$ and (x, y) consists of the contenation of the path from x to v in In_v with the path from u to y in Out_v.*

Proof: We prove this by induction on the length of p. Let p be a shortest path from u to w. If p has length no greater than d then there is some v such that u is in In_v and w is in Out_v.

Now suppose the lemma is true for all pairs u, w and all lengths up to $s \geq d$. Consider a path of length $s + 1$. Let $e = (i, j)$ be the most recently inserted edge in the path. Let the distance of i from u be d'. Then if $d' \leq d$, then there is a shortest path from u to i in In_i. Let i' be a vertex of distance $d' + d$ from u in p. Since all edges from p were present in the graph at the time Out_i was constructed, i' is in Out_i and must be a leaf at distance d from i. Then there is some $s \in S$ on the path from i to i' in Out_u not equal to i, such that a shortest path from i to i' passes through s. Hence there is a shortest path from u to w containing the path from u to i in In_i and from i to s in Out_u. While it is possible that $u = i$, since $s \neq i$, the remainder of the path p, from s to w, is shorter and can be appropriately partitioned, by induction. If $d' > d$ let i' be a vertex of distance d from i, between u and i. Then i' is a leaf in In_i at distance d from i, and there is an element s between i' and i. Hence the path from u to s and the path from s to w are both of size less than p. By induction, the lemma holds for them, and hence for their concatenation. ∎

We observe that given a set of n elements and L subsets of these elements, each of size d, some element is contained in at least Ld/n subsets. We use this to show:

Lemma 5.2 *A blocker S for F of size $O(n/d \log n)$ can be constructed deterministically in time $O(n^2 + nd \ln n)$.*

Proof: Let the *score* of vertex be the number of leaves of depth d in the subtrees rooted at that vertex, summed over all In and Out trees in which that vertex appears as a nonroot. Let L be the total number of leaves at depth d in all In and Out trees.

To construct S, repeatly find the vertex of maximum score, add it to S, then remove it and its subtrees from every tree in which it appears. Hence, the number of leaves remaining in the set of trees has been reduced, from L to $(1 - d/n)L$. After $O(n \ln L/d)$ selections, S is a blocker.

To find the scores, it suffices to traverse each tree in postorder, labeling a parent node with the sum of its children's labels. To revise the score when a vertex v is added to S, let $l_t(v)$ be the number of leaves in v's subtree in tree t. Then for each w in v's subtree, when w is removed from t, subtract $l_t(w)$ from $score(w)$, and subtract $l_t(v)$ from v's ancestors. The running time of this algorithm is $O(1)$ per vertex in a tree, since each is removed once from a tree or $O(n^2)$ over all trees plus for each v inserted into S a cost of $O(d)$ per tree to visit v's ancestors, for a total cost of $O(n^2 + nd \ln n)$.

Recall that $D(u,w) = dist(u,w)$ if the distance is no greater than d and ∞ otherwise.

Stitching Algorithm

1. Construct a blocker S for F.

2. Use any $\tilde{O}(n^3)$ algorithm to compute the static all-pairs shortest paths on the graph $G' = (S, E^s)$ where the weight of edge (s, s') is given by $D(s, s')$. Let $SS(s, s')$ denote the shortest distance from s to s', as computed by this algorithm.

3. For each $u \in V, s \in S$, if $D(u,s) = \infty$, let $D(u,s) = \min_{s' \in S} D(u,s') + SS(s', s)$. A shortest path from u to s is the concatenation of a shortest path from u to s'' with one from s'' to s, where s'' is a vertex which minimizes the expression.

4. For each $u, w \in V$, if $D(u,w) = \infty$, let $D(u,w) = \min_{s \in S} D(u,s) + D(s,w)$. A shortest path from u to w is the concatenation of a shortest path from u to s" with one from s" to w, where s" is a vertex which minimizes the expression.

Analysis: The cost of constructing the blocker is $O(n^2 + nd \log n)$. The cost of performing the stitching algorithm is dominated by the cost of the last step which is $O(n^2|S|) = O(n^2(n \log n/d))$. The amortized cost per update of maintaining the shortest paths up to length d is $O(n^2 d)$. Choosing $d = |S| = (n \log n)^{.5}$ gives an amortized update time of $O(n^{2.5}\sqrt{\log n})$.

5.1. Exact shortest paths for graphs with weighted edges

We maintain all-pairs shortest distances up to d in weighted graphs, using the algorithm described previously, and we stitch together as we did before. The only difference occurs in the definition and size of the blocker S.

Let b be the bound on the maximum weight, $b < \sqrt{n}$. We call a subset $S \subset V$ a *b-blocker* for F if every path starting or ending at the root in an Out tree or an In tree respectively, of length greater than $(d - b)$ contains a vertex in S which is distinct from the root. (Here, length refers to the sum of the weights of edges in the path.) Each path contains $\lceil (d-b)/b \rceil$ vertices excluding the root. Hence we can construct in a similar way a b-blocker of size $O(n \log n/(d/b))$. The cost of stitching together is $O((n^3 b \log n)/d)$.The cost per update of maintaining the all-pairs shortest paths up to distance d is $O(n^2 d)$. Choosing $d = |S| = \sqrt{nb \log n}$ gives a running time of $O(n^{2.5}\sqrt{b \log n})$.

References

[1] G. Ausiello, G. Italiano, A. Spaccamela, and U. Nanni. Incremental algorithm for minimal length paths. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 12–21, 1990.

[2] A. Buchsbaum, P. Kanellakis, and J. Vitter. A data structure for arc insertion and regular path finding,. *Annals of Mathematics and Artificial Intelligence*, 3:187–210, 1991.

[3] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:1–6, 1990.

[4] 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.

[5] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molinas. Proximity search in databases. In *Proceedings of the 24th VLDB Conference*, 1998.

[6] M. Henzinger and M. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22:351–362, 1998.

[7] M. Henzinger and V. King. Fully dynamic biconnectivity and transtive closure. In *36th Symposium on Foundations of Computer Science (FOCS)*, 1995.

[8] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[9] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, pages 5–11, 1988.

[10] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the Thirty-first Annual Symposium on the Theory of Computing (STOC)*, 1999.

[11] H. L. Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. LNCS 314, Springer Verlag, 1988.

[12] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Jounal of Algorithms*, 21(2):267–305, 1996.

[13] T. Reps. `www.cs.wisc/sim/reps`.

[14] T. Reps and G. Ramalingam. On the computational complexity of dynamic graph problems. *Theoretical Computer Science A*, 158:233–277, 1996.

[15] U. Zwick. All pairs shortest paths in weighted directed graphs–exact and almost exact algorithms. In *39th Symposium on Foundations of Computer Science (FOCS)*, pages 310–319, 1998.