# Fully Dynamic Biconnectivity and Transitive Closure

Monika Rauch Henzinger [*]

Department of Computer Science
Cornell University
Ithaca, NY 14850

Valerie King [†]

Department of Computer Science
University of Victoria
Victoria, BC

## Abstract

*This paper presents an algorithm for the fully dynamic biconnectivity problem whose running time is exponentially faster than all previously known solutions. It is the first dynamic algorithm that answers biconnectivity queries in time $O(\log^2 n)$ in a n-node graph and can be updated after an edge insertion or deletion in polylogarithmic time. Our algorithm is a Las-Vegas style randomized algorithm with the update time amortized update time $O(\log^4 n)$. Only recently the best deterministic result for this problem was improved to $O(\sqrt{n}\log^2 n)$.*

*We also give the first fully dynamic and a novel deletions-only transitive closure (i.e. directed connectivity) algorithms. These are randomized Monte Carlo algorithms. Let n be the number of nodes in the graph and let $\hat{m}$ be the average number of edges in the graph during the whole update sequence: The fully dynamic algorithms achieve (1) query time $O(n/\log n)$ and update time $O(\hat{m}\sqrt{n}\log^2 n+n)$; or (2) query time $O(n/\log n)$ and update time $O(n\hat{m}^{\mu-1/\mu})\log^2 n = O(n\hat{m}^{0.58}\log^2 n)$, where $\mu$ is the exponent for boolean matrix multiplication (currently $\mu = 2.38$).*

*The deletions-only algorithm answers queries in time $O(n/\log n)$. Its amortized update time is $O(n\log^2 n)$.*

## 1 Introduction

Given a graph $G = (V, E)$ with $m = |E|$ and $n = |V|$, a *fully dynamic* graph algorithm for a graph property $P$ provides three types of operations. *Insert*$(e)$ : insert the edge $e$, *delete*$(e)$ : delete the edge $e$, and *query*$(u, v)$ : test if the nodes $u$ and $v$ fulfill $P(u, v)$ in the current graph $G$. They are a model of *dynamic/interactive* situations occurring, for example, in data bases, incremental compilers, and interactive verification systems. However, they are also useful to improve the worst-case efficiency of *static* graph algorithms, for example of various matching algorithms [1].

**Biconnectivity.** Two nodes of an undirected graph are *biconnected* iff they are connected by two vertex-disjoint paths. A *biconnected component* or *block* is a maximal set of nodes that are biconnected. A node that belongs to more than one block is called an *articulation point.* A node is an articulation point iff its removal disconnects the graph [10].

*Our Result.* This paper presents the first fully dynamic biconnectivity algorithm with polylogarithmic time per operation. The algorithm is a Las-Vegas style randomized algorithm whose amortized update time is $O(\log^4 n)$ and whose worst case query time is $O(\log^2 n)$. The algorithms also can output all the nodes that belong to a block in time linear in their number and all $p$ articulation points that belong to a block in time $O(p\log n)$.

The biconnectivity properties of a network can be used to determine if the network can tolerate the failure of one of its node without becoming disconnected. If the edges of the network change over time, a fully dynamic algorithm for biconnectivity is needed. Another application area are approximation algorithms for network design problems: The 3-approximation algorithm by Ravi and Williamson [18] for the $\{0, 1, 2\}$-survivable network design problem repeatedly adds and deletes edges from the graph and tests its biconnectivity properties in between. Our new fully dynamic biconnectivity algorithm improves the time of the 3-approximation algorithm from $O(n^3)$ to $\tilde{O}(n^2)$ [20].

*Previous Work.* Two nodes are *2-edge connected* iff they are connected by two edge-disjoint paths. In [11] we presented the first fully dynamic connectivity and

*Email: mhr@cs.cornell.edu. Author's Maiden Name: Monika H. Rauch. This research was supported by an NSF CAREER Award, Grant No. CCR-9501712.

†Email: val@csr.uvic.ca. This research was supported by an NSERC Grant.

2-edge connectivity algorithms with polylogarithmic time per operation, $O(\log^3 n)$ for connectivity and $O(\log^4 n)$ for 2-edge connectivity. The algorithms are Las-Vegas style randomized algorithm. Our algorithm shows that 2-vertex connectivity can be maintained fully dynamically as efficiently as 2-edge connectivity and almost as efficiently as connectivity. Note that fully dynamic biconnectivity is at least as hard as fully dynamic connectivity or 2-edge connectivity: Fully dynamic connectivity or 2-edge connectivity can be reduced to fully dynamic biconnectivity [9], but no reduction in the other direction is known. This is also reflected in the design of efficient fully dynamic algorithms: In 1992, Eppstein, Galil, Italiano, and Nissenzweig gave the best known deterministic connectivity and 2-edge connecitivity algorithms with update time $O(\sqrt{n})$ [3, 2], but only recently Henzinger and La Poutré improved the best biconnectivity algorithm to $O(\sqrt{n}\log^2 n)$ per update [12]. There exists a lower bound on the amortized time per operation of $\Omega(\log n/\log\log n)$ for all three problems that also applies to randomized algorithms [8].

*New Ideas.* Our algorithm reduces the fully dynamic biconnectivity problem in a graph to a fully dynamic biconnectivity problem on a *chain* and $n$ fully dynamic *connectivity* problems. For this reduction we use the leveled graph decomposition of [11] and add three new concepts: (1) *Local graphs:* We store at each node a partition of its neighbors into biconnected components and use these local graphs to answer queries. Since the edges of the graph are distributed over various levels, each level "inherits" the edges of the local graph from all previous levels. To efficiently maintain the local graphs, we reduce the maintenance of the local graph to a connectivity (instead of biconnectivity) problem and we build a simple dynamic connectivity data structure to efficiently maintain the local graphs under the inheritance law. (2) *Cover data structure:* After an edge deletion we quickly need to find all newly created articulation points in a suitably chosen subgraph of $G$. Since all new articulation points lie on a path of $G$, we use a new fully dynamic data structure for testing biconnectivity along an (arbitrary) path. (3) *Active nodes:* The above ideas give a polylogarithmic deletions-only algorithm. To get an efficient fully dynamic algorithm, we cannot afford to maintain local graph at each node at each level. Instead we label suitable nodes on each level as active and update only the local graphs of active nodes at each level.

**Transitive Closure.** Given a directed graph $G$ the fully dynamic *transitive closure* problem is to maintain the property $P(u,v)$ "can the node $u$ reach the node $v$?". The fully dynamic transitive closure problem has applications to industrial robotics [21] and databases [23], but no fully dynamic algorithm better than recomputation from scratch was previously known. As pointed out by Khanna, Motwani, and Wilson [14], the sparsification technique [3], used in the best deterministic algorithm for the dynamic connectivity problem in undirected graphs cannot be used to design an efficient fully dynamic transitive closure algorithm.

*Our Results.* We give the first fully dynamic algorithms that are better then recomputation from scratch, and also a novel deletions-only transitive closure algorithm. Let $n$ be the number of nodes in the graph and let $\hat{m}$ be the average number of edges in the graph during the whole update sequence: The fully dynamic algorithms achieve (1) query time $O(n/\log n)$ and update time $O(\hat{m}\sqrt{n}\log^2 n+n)$; or (2) query time $O(n/\log n)$ and update time $O(n\hat{m}^{\mu-1/\mu}\log^2 n) = O(n\hat{m}^{0.58}\log^2 n)$, where $\mu$ is the exponent for boolean matrix multiplication (currently $\mu = 2.38$).

The deletions-only algorithm answers queries in time $O(n/\log n)$. Its amortized update time is $O(n\log^2 n)$.

*Previous Work.* The best insertions-only algorithm takes amortized time $O(n)$ per insert and $O(1)$ per query [13, 15], or $O(m^*\Delta)$ for $m$ insertions [24], where $m^*$ is the number of edges in the transitive closure and $\Delta$ is the out-degree in the final graph. The previously best deletions-only algorithm takes amortized time $O(m)$ for $m$ deletions and $O(1)$ per query, where $m$ is the number of edges in the initial graph [15], or $O(m^*\Delta)$ for $m$ deletions [24], where $m^*$ is the number of edges in the transitive closure and $\Delta$ is the out-degree in the final graph.

We first present the biconnectivity and then the transitive closure algorithms.

## 2 Biconnectivity

### 2.1 A simple algorithm

Given a graph $G = (V, E)$, we maintain (A) $G$ in a dynamic connectivity data structure which provides a spanning forest $F$, (B) $G \setminus x$ in a dynamic connectivity data structure for each node $x$, and (C) $F$ augmented with costs at nodes in a dynamic tree data structure [19]. In time $O(\log n)$ per operation, the dynamic tree data structure supports insertions and deletions of edges and a mincost query, that returns the minimum cost on a path.

The *dynamic tree data structure* decomposes $F$ into *heavy* and *light* edges such that (1) the heavy edges form *heavy* paths that are connected by *light* edges, (2) each node belongs to at most one heavy path, and (3) the path between any two nodes $u$ and $v$ is decomposed into at most $O(\log n)$ heavy paths and $O(\log n)$ light edges.

If $x$ is incident to at most one heavy edge, we store cost 0 at $x$. Otherwise, if $a$ and $b$ are the two nodes connected to $x$ by a heavy edge, we store a cost $>$ 0 at $x$ iff $a$ and $b$ are connected in $G \setminus x$. We call the resulting dynamic tree a *block-labeled dynamic tree data structure (BDT)* because of the lemma below.

We denote the tree path between two nodes $u$ and $v$ by $\pi(u, v)$.

**Lemma 2.1** **[16]** *Two nodes $u$ and $v$ are biconnected iff no node on $\pi(u,v)$, where $\pi(u,v)$ consists only of heavy edges, has cost 0.*

This lemma implies that to determine if two nodes $u$ and $v$ are biconnected, it suffices to transform the path from $u$ to $v$ to a path consisting of heavy edges only. The dynamic tree data structure implicitly allows for such a transformation, by rooting the tree first at $u$ and then at $v$. This takes time $O(\log n)$ if no costs are updated. However, our data structure has to update the cost of a node if an incident light edge becomes heavy.

When converting a light edge incident to $x$ into a heavy edge, we test in $G \setminus x$ if the two (new) heavy neighbors of $x$ are connected and set the cost of $x$ appropriately at a cost of $O(\log n)$.

Since there are only $O(\log n)$ light edges on any simple tree path, only $O(\log n)$ costs of nodes have to be updated when the tree is rooted at $u$ and then at $v$. Thus, determining if two nodes are biconnected takes time $O(\log^2 n)$.

Each insertion or deletion of an edge in $G$ requires an update in possibly all dynamic connectivity data structures. Thus, an update takes time $O(n \log^3 n)$.

The idea of the polylogarithmic algorithm is to (1) store information about each $G \setminus x$ more compactly; and (2) use sampling and the BDT to determine quickly which of the $G \setminus x$'s have become disconnected or have small cuts; (3) move edges across small cuts to a sparser "higher level" graph and update the $G \setminus x$'s. This leads to a polylogarithmic expected time fully dynamic algorithm. We describe first a deletions-only algorithm and extend it then to a fully dynamic algorithm.

## 2.2 A polylogarithmic deletions-only algorithm

We present an algorithm whose amortized time per deletion is $O(\log^5 n)$. The deletion time can be improved to $O(\log^4 n)$, but we omit these details in this abstract.

**Data structure.** The edges of $G$ are partitioned into $l = \log m$ levels $E_1, \ldots, E_l$ such that $\cup_i E_i = E$, following ideas of [11]. Initially $E_1 = E$ and $E_i = \emptyset$ for $i > 1$. We define $H_i = (V, \cup_{j \leq i} E_j)$.

For each $i$, we keep a forest $F_i$ of *tree edges* such that $F_i$ is a spanning forest of $H_i$, $F_{i-1} \subseteq F_i$ and $F_i \setminus F_{i-1} \subset E_i$. We set $F$, a spanning forest of $G$ to $F_l$ and maintain $F$ in a dynamic tree data structure.

For nodes $x$ and $u$, $n_u(x)$ denotes the neighbor of $x$ on $\pi(u, x)$. The weight $w(T)$ of a tree $T$ of $F$ is the number of nontree edges incident to the nodes of $T$, where edges whose both endpoints lie in $T$ are counted twice. The size $size(T)$ is the number of nodes in $T$.

A *block* is a maximal set of nodes that are biconnected. A node that belongs to more than one block is called an *articulation point*. A node is an articulation point iff its removal disconnects the graph [10]. An edge is said to be in a given block if both its endpoints are in the block. Every nontree edge is in exactly one block.

For every *level* $i$ with $1 \leq i \leq l$ we keep the following data structures: (1) a dynamic connectivity data structure of $(V, E_i \cup F_{i-1})$, (2) an ET-tree storing the (nontree) edges of $E_i \setminus F_i$, (described below), (3) a BDT of $F_i$, (described above), (4) a partition of the set of neighbors, ("neighbor partition") of $x$ in $H_i$, denoted $H_i(x)$, such that $y$ and $z$ are in the same subset of $H_i(x)$ iff they are connected in $H_i \setminus x$. This partition is stored in a disjoint-set data structure, which allows find-set, join, backtrack from a sequence of joins, split, and list operations. (The list operation lists all elements of a subset.) (5) for each nontree edge of $E_i$, its block in $H_i$.

**Queries.** Since $H_l = G$, we use $H_l(x)$ together with the BDT of $F$ to answer biconnectivity queries in time $O(\log^2 n)$ as described in the simple algorithm.

**Main Idea.** The edges of $G$ are partitioned into $O(\log n)$ levels such that edges in highly-connected parts are on lower levels than those in loosely-connected parts. When an edge $\{u, v\}$ of $E_i$ is deleted, $H_j(x)$ with $j \geq i$ may require updating. The nodes $x$ whose neighbor partitions have to be updated are exactly the new articulation points of $H_i$, i.e. the nodes $z$ on $\pi(u, v)$ such that $u$ and $v$ are disconnected in $H_i \setminus z$. We randomly select a set *Sampled* of $O(\log^2 n)$ edges of a subset of $E_i$ and determine the articulation points in $H_{i-1} \cup F_i \cup Sampled$. (1) If there exists no such articulation point, then no articulation point has been created in $H_i$ and we are done. (2) If no suitable articulation point is found in $H_{i-1} \cup F_i \cup Sampled$, we repeat the sampling with a new subset of $E_i$. (3) If a suitable articulation point $z$ is found in $H_{i-1} \cup F_i \cup Sampled$, with high probability there is a cut in $H_i \setminus z$ separating $u$ and $v$ that is too sparse for level $i$. First we move $\{u,v\}$ to $E_{i+1}$. Then we move all edges on the cut to $E_{i+1}$ by recursively deleting these edges from $E_i$. Finally we recursively delete $\{u,v\}$ from $E_{i+1}$.

We initialize all neighbor partitions $H_i(x)$ and maintain them as follows: Whenever a set $S$ of edges is moved from $E_i$ to $E_{i+1}$, we test for each edge $\{u,v\} \in S$ if $n_u(x)$ and $n_v(x)$ are connected in (the new) $H_i \setminus x$. (I) If yes, the connected components of $H_i \setminus x$ are unchanged and, we do not modify $H_i(x)$. (II) If $n_u(x)$ and $n_v(x)$ are not connected in $H_i \setminus x$, let $B$ be the old block containing $\{u,v\}$, and let $B_1$ and $B_2$ be the new blocks incident to $x$. We split the subset of $H_i(x)$ contained in $B$ into two subsets, those nodes contained in $B_1$ and those contained in $B_2$.

## 2.3 The deletions-only algorithm

When an edge $\{u,v\}$ in some $E_i$ is deleted or removed from $E_i$, the nodes on the path $\pi(u,v)$ may become articulation points in $H_i$. The algorithm *Test_Path* below determines those points which are articulation points in certain randomly chosen subgraphs of $H_i$, of the form $H'_i = (V, E_{i-1} \cup F_i \cup Sampled)$, where *Sampled* is a randomly chosen set of edges described below. For a given set *Sampled*, we may accordingly define for any node $x$, the neighbor partition $H'_i(x)$.

In addition, *Test_Path* decides that a set $S'$ of edges has to be moved up from $E_i$ to $E_{i+1}$ which in turn may create new articulation points on $H_i$. Hence a single edge deletion may result in numerous calls to *Test_Path*. The call *Test_Path(u,v,i)* removes $\{u,v\}$ from $H_i$ and the data structure of level $i$. At termination of the call *Test_Path(u,v,i)* the neighbor partition of $x$ is

- $H''_i(x)$, where $H''_i = H_i \setminus S'$, for every node $x$ on $\pi(u,v)$, and
- $H_i(x)$ for every node $x$ not on $\pi(u,v)$.

We first present *Test_Path* and then the *delete* algorithm which uses *Test_Path* as a subroutine.

**Definitions.** An edge $e$ in a graph $H$ *covers a node $z$ on a path $\pi(u,v)$* if it is connected to $n_u(z)$ and $n_v(z)$ in $H \setminus z$. A node is *uv-covered* iff there is an edge $e$ which covers it on $\pi(u,v)$. A node is *covered* in $H$ if it is not an articulation point of $H$.

If $T$ contains the vertices $u$ and $x$, let $T_u \setminus x$ denote the subtree of $T \setminus x$ which contains $u$.

A vertex $x$ on a path $\pi(u,v)$ of tree $T$ is a *weighted midpoint* of the path if $w(T_u \setminus x) \leq (1/2)w(T)$ and $w(T_u \setminus n_v(x)) \geq (1/2)w(T)$.

In the algorithm *Test_Path* below, $\pi(u,v)$ is a path in a tree $T \in F_i$. Initially, $Sampled = S' = \emptyset$.

**Test_Path(u,v,i):**

- **Split path:** Determine the mid-edge $(y,z)$ of $\pi(u,v)$. Wlog $y \in \pi(u,z)$.
  **If** $size(T_u \setminus y) \leq size(T_v \setminus z)$
  **then** $y' \leftarrow y$, $z' \leftarrow z$, $u' \leftarrow u$, $v' \leftarrow v$
  **else** $y' \leftarrow z$, $z' \leftarrow y$, $u' \leftarrow v$, $v' \leftarrow u$.
- **Sample:** If $u' = y'$, stop. Let $T_1 \leftarrow T_{u'} \setminus z'$. Let $x$ be the weighted midpoint of $\pi(u', y')$. Sample $2c\log^2 m$ edges from $E_i$ which are incident to $T_1$ for some appropriate constant $c$. Call this set *Sampled.*
- Determine and mark the nodes in $\pi(u',y')$ which are covered in $H'_i \leftarrow (V, \cup E_{j<i} \cup F_i \cup Sampled)$;
- **If** all the nodes in $\pi(x,y')$, inclusive $y'$ and exclusive $x$, are covered, **then** set $y' \leftarrow n_{u'}(x)$, $z' \leftarrow x$ and **goto** Sample step.
- **Else** let $w \in \pi(x,y')$ be a node which is not covered. Let $\{a_1, \ldots, a_d\}$ be the set of all neighbors of $w$ that are connected to $n_{u'}(w)$ in $H'_i(w)$. Let $T_2 \leftarrow \cup_{1 \leq j \leq d} T_{a_j}(w)$. Search all nontree edges incident to $T_2$ to determine the $S \leftarrow$\{edges with only one endpoint in $T_2$\}.
  - If $0 \leq |S| \leq w(T_2)/2c' \log m$ then label each edge in $S$ by $\{u', w\}$ and set $S' \leftarrow S \cup S'$.
  - Determine the nodes in $\pi(u',w)$ which are covered in $H_i \setminus S'$.
  - *Update data structures:* Each node on $\pi(u',w)$ that is not covered is a new articulation point in $H_i \setminus S'$. For each new articulation point $s$, update $H_i(s)$ such that two neighbors of $s$ lie in the same subset of $H_i(s)$ iff they are connected in $(H_i \setminus S) \setminus s$. Update the ET-tree of the block of $H_i \setminus S$ at the new articulation points.
- *Repeat for second half of path:* Let $u \leftarrow y'$ and let $v \leftarrow v'$; **goto** Split path step.

We can now describe the deletion algorithm.

**The Deletion Algorithm.** If $e = (u, v) \in G_i$ is deleted, we denote for all $j$ by $H_j^b$ the graph $H_j$ before and by $H_j^a$ the graph $H_j$ after the deletion of $e$.

**Delete**$(e, i)$

*Case A:* $e \in F$ : If the removal of $e$ disconnects $G$, then remove $e$ from $G_i$ and $F$ and stop. Otherwise, find the replacement edge $e'$ from the lowest-numbered possible level. Remove $e$ from $F$, add $e'$ to $F$, and continue as in Case B.

*Case B:* $e \in G \setminus F$ : We denote by $S$ the edges that still have to be moved from $E_i$ to $E_{i+1}$.

```
1. $S$ is a set of edges returned by $Test\_Path(u, v, i)$,
   i.e., $S \leftarrow Test\_Path(u, v, i)$.
2. while $S \neq \emptyset$ do
   (a) Remove an edge $\{a, b\}$ from $S$. Remove
       $\{a, b\}$ from $E_i$ and add it to $E_{i+1}$. (This
       modifies $H_i$.)
   (b) Let $\{u, w\}$ be the label of $\{a, b\}$. Find the
       node $a'$ ($b'$) closest to $a$ ($b$) on $\pi(u, w)$.
   (c) $S =$
       $S \cup Test\_Path(a, a', i) \cup Test\_Path(b, b', i)$.
3. Call Delete($e$,$i + 1$).
```

The algorithm maintains the following invariant after every call to *Test_Path*.

**Lemma 2.2** *Let $x$ be a vertex. Let $S_x$ contain all edges of $S$ that are labeled with an edge $\{u, w\}$ and $x \in \pi(u, w)$. Then two neighbors of $x$ are in the same subset of $H_i(x)$ iff they are connected in $(H_i \setminus S_x) \setminus x$.*

When the while-loop of *Delete*$(e, i)$ terminates, $S = \emptyset$, and, thus, two neighbors of $x$ are in the same subset of $H_i(x)$ iff they are connected in $H_i \setminus x$. After each *delete*$(e)$ operation the following invariant is maintained.

**Theorem 2.3** *For every level $i$ and every node $x$ two neighbors of $x$ are in the same subset of $H_i(x)$ iff they are connected in $H_i \setminus x$.*

This shows the correctness of the algorithm.

## 2.4 The Cover Data Structure

We are given a graph $H = (V, E)$ with neighbor partitions $H(x)$ for each $x \in V$, a set of edges $S$, a spanning tree $F$ of the graph $H' = (V, E \cup S)$, and a path $\pi(u, v)$ in $F$. Edge $\{u, v\}$ is not an element of $E \cup S$. xxxxx??Is this needed? Initially, a node in $V$ is marked iff it is $uv$-covered in $H$.

The deletions algorithm uses the cover data structure with $H = H_{i-1}$.

The following algorithm marks the nodes in $\pi(u, v)$ which are covered in $H' = (V, E \cup S)$.

*Cover*$(H, S, u, v)$

```
While $S \neq \emptyset$ move an edge $\{s, t\}$ from $S$ to $E$
1. Find the node $a$ closest to $s$ on $\pi(u, v)$ and the
   nodes $n_s(a)$ and $n_t(a)$. Find also the correspond-
   ing node $b$ closest to $t$, and $n_s(b)$ and $n_t(b)$.
2. Mark all nodes on $\pi(a, b)$ (excluding $a$ and $b$).
3. {Adjust the neighbor partitions of $a$ and $b$ in $H$.}
   If $a \neq n_s(a)$ then
      (a) In $H(a)$, do join of findset($n_s(a)$) and
          findset($n_t(a)$).
      (b) If
          in $H(a)$, findset($n_u(a)$) = findset($n_v(a)$)
          then mark $a$. (The vertices $n_u(a)$ and $n_v(a)$
          are connected in $H$.)
   If $b \neq n_s(b)$ then
      (a) In $H(b)$, do join of findset($n_s(b)$) and
          findset($n_t(b)$).
      (b) If in $H(b)$, findset($n_u(b)$) = findset($n_v(b)$)
          then mark $b$. (The vertices $n_u(b)$ and $n_v(b)$
          are connected in $H$.)
```

If $F$ is stored as a BDT then the nodes in a path may be marked by adding 1 to each node in the path, in a single operation. To find an uncovered node in $\pi(u, y)$: Execute a mincost query on the BDT of $F$ which returns a node with cost 0, if there is such a node.

**Proof of Correctness**

## 2.5 Other Implementation Details

**The ET-trees.** We present a modified version of the data structure of [11], called *ET-trees.* We encode an arbitrary tree $T$ with $n$ vertices using a sequence of $2n-1$ symbols, which is generated as follows: Root the tree at an arbitrary vertex. Then traverse $T$ in depth-first search order traversing each edge twice (once in each direction) and visiting every degree-$d$ vertex $d$ times, except for the root which is visited $d+1$ times. Each time any vertex $u$ is encountered, we call this an *occurrence* of the vertex. Let $ET(T)$ be the sequence of node occurrences representing an arbitrary tree $T$.

For each spanning tree $T(B)$ of a block $B$ of $H_i$ each occurrence of $ET(T(B))$ is stored in a node of a balanced binary search tree, called the *ET(T(B))-tree.* For each vertex $u \in T(B)$, we arbitrarily choose one occurrence to be the *active* occurrence of $u$.

With the active occurrence of each vertex $v$, we keep the (unordered) list of nontree edges in $B$ which are incident to $v$, stored as a balanced binary tree. Each node in the ET-tree contains the number of nontree edges stored in its subtree.

Using this data structure for each level we can sample an edge of $T_1$ in time $O(\log n)$. The data structure can be maintained during the insertion or deletion of an edge in $H_i$ or $F_i$ in time $O(\log n)$. The total time spent in updating the ET-trees on all levels during $m$ deletions is $O(m \log^2 n)$. The details are omitted in this abstract.

**The neighbor partitions** $H_i(x)$. Store each disjoint set of nodes in a balanced tree. Then a join, delete, and insert can be executed in time $O(\log n)$. A split of a set $S$ into $S_1$ and $S_2$ can be executed in $O(\log n) \min\{|S_1|, |S_2|\})$ and a list of elements in a set can be generated in constant time per element.

**finding weighted midpoints, node on** $\pi(u, v)$ **closest to** $s$

**Running Time Analysis.** We show that the amortized cost for a deletion is $O(\log^5 n)$ if there are $m$ deletions. We test in time $O(\log^2 n)$ for a replacement edge in the dynamic connectivity data structure of $G_j \cup F_j$ for $j \geq i$, starting with $j = i$. Each call of *Test_Path* takes time $O(\log n)$ to determine $a'$, $n_v(a')$, $b'$, $n_v(b)$, $x$, and $y$, and to build the ET-tree of $T_u(x)$ and of $T_v(y)$ and time $O(\log^3 n)$ to sample and cover. If no articulation point $x'$ with $w(T_u(x')) > w(T_1)/2$ is found, we repeat the sampling on a subtree of half the weight, when (after at most $\log n$ repetitions) $w$ is reached, we recurse on the other half of the path. Thus, the total sampling cost per removed edge of $G_i$ is $O(\log^5 n)$ and the total cost of this case over the course of the algorithm for level $i$ $O(m_i \log^5 n)$. We show that $\sum_i m_i = O(m)$, giving a total sampling cost $O(m \log^5 n)$.

**Lemma 2.4** *The total number $m_i$ of edges ever in $G_i$ is $m/c'^{i-1}$.*

If the block containing $(u, v)$ is split at $x'$, let $E'$ be the set of tree and non-tree edges of $G_i \cup F_{i-1}$ that are incident to the nodes of $F_w(x') \cup x'$. We show below that the time spent for splitting $B$ except for sampling is $O(|E'| \log^2 n)$. If $|S_w| \leq w(F_w(x'))/(c' \log m)$ we charge $O(\log^2 n)$ to each edge in $E'$ to amortize these costs. Since the size of the block to which each edge of $E'$ belongs is halved, each edge is charged at most $\log n$ times per level, for a total of $O(\log^3 n)$ per level. If $|S_w| > w(F_w(x'))/(c' \log m)$, the probability of this subcase occurring is $(1 - 1/(c' \log m))^{c \log^2 n} = O(1/n^2)$ for $c = 4c'$ and the total cost of this case is $O(|E'| \log^2 n)$. Thus this contributes an expected cost of $O(\log^2 n)$ per operation.

We still have to show that the cost for all other work is $O(|E'| \log^2 n)$. Computing $S_w$, adding all edge incident to $F_w(x') \setminus S$, and splitting the ET-tree at the new articulation points takes $O(|E'| \log n)$. We receive the $O(|E'| \log^2 n)$ bound by showing that updating the graphs $H_j(s)$ for all new articulation points $s$ in $H_i$ takes time $O(|E'| \log n)$ for each level $j$. Showing the bound for level $i$ suffices, since we showed above that the work for $j > i$ is dominated by the work on level $i$. We also showed above that the work on level $i$ is $O(d_i(s) \log n)$, where $d_i(s)$ is the number of edges incident to $s$ in $E'$. Summed over all $s$, this gives $O(|E'| \log n)$.

This gives an $O(\log^5 n)$ amortized expected time per deletion, which can be improved to $O(\log^4 n)$ by avoiding the double-recursion during sampling.

## 2.6 A polylogarithmic fully dynamic algorithm

We increase the number of levels to $l = 2 \log n$ and insert every edge on level $l$. If the number of non-tree edges on a level $i$ becomes $b_i = n^2/2^{i-2}$, we move all edges from level $i$ to $2 \log n$ to level $i - 1$. This is called a *rebuild at level* $i$, its cost is $O(b_i \log^2 n)$. In [11] we showed the following lemma.

**Lemma 2.5** *The number of insertions between two rebuilds on level i is at least $b_i/2$.*

Instead of charging $O(\log^3 n)$ for splits to the non-tree and tree edges on level $i$, we charge them to the non-tree edges and the nodes on level $i$. To recharge the non-tree edges and nodes after a rebuild on level $i$ and to pay for the rebuild, we charge each insertion $O(\log^3 n)$ per level, $O(\log^4 n)$ altogether. The total recharging costs are $O((b_i + n) \log^3 n)$ for the nodes and the non-tree edges on level $i - 1$ and $O(n)$ for the nodes on each level $\geq i$ (there are no non-tree edge after the rebuild) for a total of $O(b_i \log^3 n + n \log^4 n)$. Thus, if $b_i = \Omega(n \log n)$, i.e. $i \leq \log n - \log \log n$ the insertions can pay for recharging all nodes. For levels $i > \log n - \log \log n$, we modify the deletions algorithm as follows: We define $O(b_i)$ active nodes and only update the graphs $H_i$ of active nodes. For each inactive node $x$ we keep the graph $H_i(x)$ identical to the graph $H_j(x)$, where $j$ is the largest level smaller than $i$ on which $x$ is active. (The cost of updating $H_i(x)$ is charged to the work on level $j$, not $i$.) Additionally, the total number of active nodes on all levels $\geq i$ will be $O(b_i)$. Thus, the total recharging cost after a rebuild on level $i$ is $O(b_i \log^3 n)$ for the non-tree edges and active nodes on level $i-1$ and $O(b_i \log^3 n)$ for the active nodes on all levels $\geq i$.

For every non-tree edge of $G_i$ assume the path between its two endpoints is colored red. A node on level $i > \log n$ is *active* if it is (1) the endpoint of a non-tree edge of $G_i$, (2) incident to at least 3 red edges, or (3) connected by a red edge to an active node of type (1) or (2).

This guarantees that there are $O(b_i)$ active nodes. Each inactive node lies either on a tree path between 2 (unique) active nodes of type (3) or not between any 2 active nodes.

We modify the coverage data structure as follows: All pairs of edges incident to a non-active nodes are considered to be $uv$-covered (no matter whether they were $uv$-covered by a *Cover* operation or not). Thus, a *FirstUncovered* query returns only active nodes which guarantees that the time spent at level $i$ between 2 rebuilds at level $i$ is at most $O(b_i \log^3 n)$. We omit the details.

**Theorem 2.6** *Given a graph with $m_0$ initial edges, the presented data structure answer biconnectivity queries in time $O(\log^2 n)$, executes insertions in amortized time $O(\log^4 n)$, and executes deletions in amortized expected time $O(\log^4 n)$.*

## 3 Reachability in directed graphs

We present one algorithm to answer reachability queries in dynamic (deletions-only) directed graphs and two for fully dynamic directed graphs. All three algorithms are Monte Carlo; that is, they always answer queries correctly when the answer is "yes, node $i$ is reachable from node $j$, but, with probability $O(1/n^c)$ (where $c$ depends on the constants chosen by the algorithm), they may err when answering "no".

Two techniques are combined in a novel fashion. The first technique is suggested by the following theorem. A similar theorem is used in [22] in the problem of computing transitive closure in parallel in a static digraph.

**Theorem 3.1** *Let $c$ be a constant. If $cs \ln n$ nodes are chosen at random, then for all times during a sequence of $n^3$ updates, for all pairs of nodes $x$ and $y$, if there is a path from $x$ to $y$, then with probability $1 - O(1/n^{c-6})$ there is a shortest acyclic path from $x$ to $y$ such that every gap longer than $n/s$ has a distinguished node.*

The second technique is the simple procedure described in [5] which is used for updating a breadth-first search tree in an undirected deletions-only dynamic graph. An easy modification of their analysis gives the following:

**Theorem 3.2** *The set of all nodes reachable from (or which reach) a specified node by a path of distance no greater than $k$ in a dynamic (deletions only) directed graph can be maintained in time $O(mk)$.*

Denote the set of nodes reachable from (or which reach) a node $x$ by a path of distance no greater than $k$, by $out(x, k)$, $(in(x, k))$, respectively.

The proofs of correctness of the algorithms described here are omitted in this extended abstract. They involve a straightforward application of the first theorem above.

### 3.1 Deletions-only

Let $2 \leq r \leq n$ be a parameter that can be chosen by the user. For $i = 1, \ldots, \ln r$, randomly select a set of $\min\{O(2^i \log n), n\}$ distinguished nodes $S_i$. For each distinguished node $x$, maintain $out(x, n/2^i)$ and $in(x, n/2^i)$ for each $i$ such that that $x \in S_i$; and $Out(x) = \cup_{\{i|x \in S_i\}} out(x, n/2^i)$ and $In(x) = \cup_{\{i|x \in S_i\}} in(x, n/2^i)$. For each node $u \in V$ maintain the sets $out(u, n/r)$ and $in(u, n/r)$.

To answer a query$(u, v)$, test first if $v$ is in $out(u, n/r)$. If not, then test to see if for any distinguished node $x$, $u \in In(x)$ and $v \in Out(x)$. If for some $x$, both test results are positive, output "yes."

If the shortest path from $u$ to $v$ has length no greater than $n/r$ then the query is answered correctly. If the shortest path between $u$ and $v$ has length greater than $n/r$, then the query is answered correctly with high probability.

The total update time is $O(\sum_i (2^i \log n)(n/2^i)m + n(n/r)m) = O(mn \log n \log r + n^2 m/r)$ . The query time is proportional to the number of distinguished nodes which is $O(\min\{r \log n, n\})$. The amortized update time is $O(n \log^2 n + n^2/r)$. If $r = n/\log^2 n$ the amortized update time is $O(n \log^2 n)$, the amortized query time is $O(n/\log n)$.

### 3.2 A fully-dynamic transitive closure algorithm

**Approach 1:** Keep the deletions-only data structure with $r = n/\log^2 n$ to give the correct answer if there is an "old" path between two nodes.

Additionally after each insertion of an edge $(x, y)$, compute $in(x, n)$ and $out(x, n)$. After each deletion, recompute $in(x, n)$ and $out(x, n)$, for all inserted edges $(x, y)$, and adjust the deletions-only data structure for old paths. Rebuild the deletions-only data structure after $\sqrt{n}$ updates. To answer a query $(u, v)$, test if there's an old path between $u$ and $v$ and if not, test if $u \in in(x, n)$ and $v \in out(x, n)$ for all $x$ which are tails of newly inserted edges.

Let $m_0$ be the number of edges in the graph at the time of the last rebuild. The total time for no more than $\sqrt{n}$ deletions and no more than $\sqrt{n}$ insertions since the last rebuild is $O(m_0 n \log^2 n + n(m_0 + \sqrt{n}))$, which is $O(m_0 \sqrt{n} \log^2 n + n)$ per update. Let $\hat{m}$ be the average number of edges in $G$ during the sequence of updates. Since $m_0 \leq \hat{m} + \sqrt{n}$, this is $O(\hat{m}\sqrt{n}\log^2 n + n)$ amortized time per update, with $O(n/\log n)$ query time.

**Approach 2:** As before, keep the Approach 1 deletions-only data structure with $r = n$ to give the correct answer if there is an "old" path between two nodes.

Let $t$ be a parameter selected by the user, $2 \leq t \leq n$. Randomly select a set of $t' = ct \log n$ *special nodes* $S = \{s_1, s_2, ..., s_{t'}\}$. Maintain the following data structures: 1. For each special node $s$, $out(s, n/t)$ and $in(s, n/t)$.
2. An $t' \times t'$ matrix $M$, where $M_{i,j} = 1$ iff $s_j$ lies in $out(s_i)$ and the matrix $M^*$ that contains the transitive closure of $M$.
3. An $n \times t'$ matrix $NS$ whose $i, j$ entry is 1 iff node $i$ is in $in(s_j, n/t)$ and matrix $SN$ whose $i, j$ entry is 1 iff node $j$ is in $out(s_i, n/t)$.

When an edge is inserted, make its tail $u$, a new special node and: (1) determine $in(u, n/t)$ and $out(u, n/t)$; (2) add a new row and a new column to $M$ for $u$ and recompute $M^*$ by adding all values that involve the new row and the new column; and (3) add a new column to $NS$ and a new row to $SN$. Compute $NSM^*$ and $M^*SN$ to determine all special nodes that are reachable from and can reach any given node, respectively.

When an edge is deleted, update: the old deletions-only data structures; $in(s, n/t)$ and $out(s, n/t)$, for each special node $s$; $M$ and $M^*$, recomputing the latter from scratch; $NS$ and $SN$, and recomputing $NSM^*$ and $M^*SN$.

To answer a query $(x, y)$, test (1) if there exists a path from $x$ to $y$ without a new edge using the old deletions-only data structure and (2) if there exists a special node $s$ such that $x$ can reach $s$ and $s$ can reach $y$ using the above data structure.

We analyze the running time of approach 2. Let $m_0$ be the number of edges at the time of the last rebuild, let $u_d$ be the number of deletions since the last rebuild, and let $u_i$ be the number of insertions since the last rebuild. The total time for the old deletions-only data structure is $O(nm_0 \log^2 n)$.

In addition, for each deletion, $O((t' + u_i)(n/r)m)$ for the $in(s, n/r)$'s and $out(s, n/t)$'s; $O(u_d(t' + u_i)^2 + u_d M(t' + u_i))$ to update $M$ and $M^*$ and $O(u_d(n/t')M(t' + u_i))$ to update and then multiply $NS$ and $SN$ with $M^*$. Thus, the total time for $u_d$ deletions on the new data structures is $O(u_d(t'+u_i)^2 + u_d M(t + u_i) + u_d(n/t)M(t + u_i))$.

The total time for $u_i$ insertions is $O(u_i(m_0 + u_i))$ for computing $in(u, n/t)$ and $out(u, n/t)$ for each new special node $u$, $O(u_i(t' + u_i)^2)$ for updating $M$, $O(u_i(n/(t' + u_i))M(t' + u_i))$ for the resulting matrix multiplications. Thus, the total time is $O(u_i(m_0 + u_i) + u_i(t + u_i)^2 \log^2 n + u_i(n/(t + u_i)M(t + u_i))$. If we rebuild when $u_i = t$, the total time for $t$ insertions is $O(tm_0 + t^3 \log^2 n + nM(t))$.

In a sequence of $t$ insertions, if the sequence contains at most $t$ deletions, charge the cost for the deletions to the insertions. Otherwise, amortize the cost of the deletions over the deletions. The amortized cost of each update operation is $O(nm_0 \log^2 n/t + m_0 + t^2 \log^2 n + nM(t)/t) = O(n\hat{m} \log^2 n/t + \hat{m} + t^2 \log^2 n + nM(t)/t)$ where $m_0 \leq \hat{m} + t$ is the average number of edges in the graph during the sequence of updates. Query time is $O(n/\log n + t \log n)$.

Various trade-offs can be shown with approach 2. If we assume $M(t) = O(t^3)$ and want to minimize the total time per update operaton, we choose $t = \hat{m}^{1/3}$. This gives an amortized update time of $O(n\hat{m}^{2/3} \log^2 n)$ and a query time of $O(n/\log n + \hat{m}^{1/3} \log n) = O(n/\log n)$. If we assume $M(t) = O(t^{2.38})$ and want to minimize the total time per update, we choose $t = \hat{m}^{1/2.38}$. This gives an amortized update time of $O(n\hat{m}^{1.38/2.38} \log^2 n) = O(n\hat{m}^{0.58} \log^2 n)$ and a query time of $O(n/\log n + \hat{m}^{0.4202 \log n}) = O(n/\log n)$.

## References

[1] J. Cheriyan, "Randomized $\tilde{O}(M(|V|))$ Algorithms for Problems in Matching Theory", Research report (revised) CORR 93-25, Department of Combinatorics & Optimization, Univ. Waterloo, January 1995.

[2] D. Eppstein, Z. Galil, G. F. Italiano, "Improved Sparsification", Tech. Report 93-20, Department of Information and Computer Science, University of California, Irvine, CA 92717.

[3] D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig, "Sparsification - A Technique for Speeding up Dynamic Graph Algorithms" *Proc. 33rd Symp. on Foundations of Computer Science*, 1992, 60–69.

[4] D. Eppstein, Z. Galil, G. F. Italiano, and T. Spencer. "Separator Based Sparsification for Dynamic Planar Graph Algorithms". *Proc. 25th Symp. on Theory of Computing*, 1993, 208–217.

[5] S. Even and Y. Shiloach, "An On-Line Edge-Deletion Problem", *J. ACM* 28 (1981), 1–4.

[6] G. N. Frederickson, "Data Structures for On-line Updating of Minimum Spanning Trees", *SIAM J. Comput.*, 14 (1985), 781–798.

[7] G. N. Frederickson, "Ambivalent Data Structures for Dynamic 2-edge-connectivity and $k$ smallest spanning trees", *Proc. 32nd Symp. on Foundations of Computer Science*, 1991, 632–641.

[8] M. L. Fredman and M. Rauch Henzinger, "Lower Bounds for Fully Dynamic Connectivity Problems in Graphs", to appear in *Algorithmica.*

[9] Z. Galil and G. F. Italiano, "Reducing Edge Connectivity to Vertex Connectivity" *SIGACT News* 22 (1991), 57–60.

[10] F. Harary, "Graph Theory", *Addison-Wesley, Reading, MA*, 1969.

[11] M. Rauch Henzinger and V. King, "Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation", to appear in *Proc. 27th Symp. on Theory of Computing*, 1995.

[12] M. Rauch Henzinger and H. La Poutré, "Sparse Certificates for Dynamic Biconnectivity in Graphs", submitted.

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

[14] S. Khanna, R. Motwani, R. Wilson, "Graph Certificates and Lookahead in Dynamic Directed Graph Problems, with Applications", submitted.

[15] 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, 1988, 106–120. submitted.

[16] M. H. Rauch, "Fully Dynamic Biconnectivity in Graphs". *Proc. 33rd Symp. on Foundations of Computer Science*, 1992, 50–59.

[17] M. H. Rauch, "Improved Data Structures for Fully Dynamic Biconnectivity in Graphs". *Proc. 26th Symp. on Theory of Computing*, 1994, 686–695.

[18] R. Ravi and D. Williamson, "An Approximation Algorithm for Minimum-Cost Vertex-Connectivity Problems", *Proc. 6th Symp. on Discrete Algorithms*, 1995, 332–341.

[19] D. D. Sleator, R. E. Tarjan, "A data structure for dynamic trees" *J. Comput. System Sci.* 24 (1983), 362–381.

[20] D. Williamson, personal communication.

[21] R. H. Wilson, "On Geometric Assembly Planning", PhD thesis, Stanford University, 1992. Stanford Technical Report STAN-CS-92-1416.

[22] J. D. Ullman and M. Yannakakis, "High-Probability Parallel Transitive Closure Algorithms", *SIAM J. Comput.*, 20 (1991), 100–125.

[23] M. Yannakakis, "Graph-theoretic Methods in Database Theory", *Proc. 22nd Symp. on Theory of Computing*, 1990, 230–242.

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