An Extended Abstract will appear in Proc. European Symp. on Algorithms (ESA'97)

# Quasi-Fully Dynamic Algorithms for Two-Connectivity, Cycle Equivalence and Related Problems

*Madhukar Reddy Korupolu** *Vijaya Ramachandran**

June 2, 1997

**Abstract**

In this paper we introduce a new class of dynamic graph algorithms called *quasi-fully dynamic algorithms*, which are much more general than the backtracking algorithms and are much simpler than the fully dynamic algorithms. These algorithms are especially suitable for applications in which a certain core connected portion of the graph remains fixed, and fully dynamic updates occur on the remaining edges in the graph.

We present very simple quasi-fully dynamic algorithms with $O(\log n)$ worst case time, per operation, for 2-edge connectivity and cycle equivalence. The former is deterministic while the latter is Monte-Carlo type randomized. For 2-vertex connectivity, we give a randomized Las Vegas algorithm with $O(\log^4 n)$ expected amortized time per operation. We introduce the concept of quasi-$k$-edge-connectivity, which is a slightly relaxed version of $k$-edge connectivity, and show that it can be maintained in $O(\log n)$ worst case time per operation. We also analyze the performance of a natural extension of our quasi-fully dynamic algorithms to fully dynamic algorithms.

The quasi-fully dynamic algorithm we present for cycle equivalence (which has several applications in optimizing compilers) is of special interest since the algorithm is quite simple, and no special-purpose incremental or backtracking algorithm is known for this problem.

## 1 Introduction

Dynamic graph algorithms have received a great deal of attention in the last few years (see e.g., [4]). These algorithms maintain a property of a given graph under a sequence of suitably restricted updates and queries. Throughout this paper we will be concerned with edge updates (insertions/deletions) only: insertion/deletion of isolated vertices can be implemented trivially in all the known dynamic graph algorithms. The existing dynamic algorithms can be classified into three types depending on the nature of (edge) updates allowed:

- **Partially Dynamic:** Only insertions are allowed (**Incremental**) or only deletions are allowed (**Decremental**).
- **Backtracking:** Arbitrary insertions are allowed. But only backtracking deletions (Undo operation) are allowed [17].
- **Fully Dynamic:** Arbitrary insertions and arbitrary deletions are allowed.

*Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712. This research was supported in part by the NSF grant CCR/GER-90-23059. E-mail: {madhukar,vlr}@cs.utexas.edu. An extended abstract of this work will appear in [14].

Fully dynamic algorithms tend to involve complicated data structures and are quite difficult to implement. The deterministic fully dynamic algorithms for 2-edge connectivity (given in [3]), 2-vertex connectivity (given in [10]) and cycle equivalence (given in [8]) are good examples of this. The randomized fully dynamic algorithms for 2-edge connectivity (given in [9, 15]) and 2-vertex connectivity (given in [11]) are pretty involved too. In fact, the 2-vertex connectivity algorithm of [11] does not work for some graphs in which the maximum degree is $\omega(polylog(n))$ ([13]). In view of this, simpler algorithms will be more useful for applications which do not require the generality of the fully dynamic algorithms.

With this motivation, we consider another class of dynamic algorithms, where both insertions and deletions are allowed but the deletions are slightly restricted. The restriction is as follows: The algorithm maintains a spanning forest $F$ of the current graph $G$. Arbitrary edge insertions are allowed; Arbitrary nonforest edge deletions are allowed; But deletion of a forest edge is allowed only if it is a cut edge. An operation which attempts to delete an edge of $F$ which is not a cut edge, will be considered *invalid*. Such a valid sequence of operations, w.r.t. $F$, will be referred to as a *quasi-fully dynamic sequence* of operations (w.r.t. $F$) and the algorithms that support such a sequence of operations will be called *quasi-fully dynamic algorithms*. The algorithm will detect if a given operation is valid or not, and if not, it would flag an error.

Why quasi-fully dynamic algorithms?

**Firstly**, these algorithms are more general than backtracking algorithms; i.e., a backtracking sequence of operations is a quasi-fully dynamic sequence (w.r.t. a suitable spanning forest $F$). By maintaining $F$ in a natural way, we can show that during a backtracking sequence of updates, a forest edge is deleted only if it is a cut edge. This natural way of maintaining $F$ is the following: If $u$ and $v$ are in different connected components when the edge $(u, v)$ is added, then this edge is added to $F$. Otherwise $(u, v)$ never enters $F$. In this scenario when $(u, v) \in F$ is about to be deleted, all the edges that were inserted after $(u, v)$ would have been removed and hence $(u, v)$ would be a cut edge.

Another useful feature of the quasi-fully dynamic algorithms is the following: these algorithms can be extended to handle the invalid deletions in a way which is more efficient than rebuilding the entire data structure. On the other hand, in the backtracking algorithm of [17], performing an invalid operation requires the rebuild of the entire data structure. As expected, these invalid deletions can be very expensive in the worst case. Section 5 discusses this feature.

**Secondly**, these algorithms are much simpler than the fully dynamic algorithms. For instance, the quasi-fully dynamic algorithms for 2-edge connectivity and cycle equivalence are as simple as maintaining a dynamic tree data structure. For the sake of completeness, a brief review of the essential features of the dynamic tree data structure is given in the appendix (section 6).

**Thirdly**, these algorithms would be ideal for situations where some core structure (of the input graph) remains fixed and the updates occur only on the remaining part. The only requirement is that we should be able to extract a spanning tree from the core structure.

Throughout the paper, $n$ denotes the number of vertices in the graph $G$. Unless otherwise mentioned, deletions considered in quasi-fully dynamic algorithms will be valid ones only. The current status of the dynamic algorithms for connectivity, 2-connectivity and cycle equivalence is summarized below.

**Connectivity:** Backtracking connectivity can be solved in $\Theta(\log n/\log\log n)$ time per operation by a straightforward application of the backtracking algorithm for the union-find problem (see [22, 21]). Currently the best deterministic fully dynamic connectivity algorithm takes $O(\sqrt{n})$ time per update and $O(1)$ per query [3]. In [9] a randomized fully dynamic algorithm, taking $O(\log^3 n)$ amortized expected time per update and $O(\log n/\log\log n)$ worst case time per query, is presented.

This paper also gives a simpler deterministic fully dynamic connectivity algorithm with $O(\sqrt{n}\log n)$ time per update. An empirical study of the dynamic connectivity algorithms is presented in [1].

**2-Edge Connectivity and Quasi-$k$-Edge Connectivity:** An incremental algorithm with $O(\alpha(m,n))$ amortized time per operation was given in [23, 16]. A backtracking algorithm with $O(\log n)$ worst case time per operation is presented in [17]. The best known deterministic fully dynamic algorithm takes $O(\sqrt{n})$ time per update and $O(\log n)$ time per query [3]. A randomized fully dynamic algorithm with an $O(\log^4 n)$ expected amortized time per update and $O(\log n/\log\log n)$ worst case time per query is claimed in [9]: the details of the algorithm presented there are rather sketchy. A somewhat different randomized fully dynamic algorithm with polylog time per operation is given in [15].

In this paper, we present a simple quasi-fully dynamic algorithm with the same time bounds as the backtracking case: $O(\log n)$ worst case time per operation. We then introduce the concept of quasi-$k$-edge connectivity and show that the above algorithm can be extended to answer the quasi-$k$-edge connectivity queries within the same time bounds.

**Cycle Equivalence:** Two edges $e_1$ and $e_2$ of an undirected graph are *cycle equivalent* iff the set of cycles that contain $e_1$ is exactly the same as the set of cycles that contain $e_2$. Finding the cycle equivalence classes is central to several compilation problems. (See [12, 20, 6] for some applications of cycle equivalence.) As mentioned in [8], dynamic algorithms for this problem can speed up incremental compilers.

No special-purpose incremental or backtracking algorithms are known for this problem. The only dynamic algorithms known for handling an incremental or a backtracking sequence of updates are the fully dynamic algorithms. A deterministic fully dynamic algorithm with $O(\sqrt{n}\log n)$ time per update and $O(\log^2 n)$ time per query is presented in [8]. A randomized fully dynamic algorithm with $O(\log^3 n)$ amortized expected time for updates and queries is given in [9].

In this paper, we present a very simple randomized quasi-fully dynamic algorithm which takes $O(\log n)$ worst case time per operation. This algorithm is Monte-Carlo type. We also show some connection of cycle equivalence to 3-edge connectivity.

Our quasi-fully dynamic algorithm for cycle-equivalence is of special interest because of the absence of special-purpose incremental/backtracking algorithms for this problem.

**2-Vertex Connectivity:** Incremental algorithms with $O(\alpha(m,n))$ amortized time per operation are given in [23, 16]. A backtracking algorithm with $O(\log n)$ worst case time per operation is presented in [17]. The best known deterministic fully dynamic algorithm takes $O(\sqrt{n}\log^2 n)$ amortized time per update and $O(1)$ worst case time per query [10]. A randomized fully dynamic algorithm with an $O(\log^4 n)$ expected amortized time per update and $O(\log^2 n)$ worst case time per query is stated in [11]: however, this algorithm does not work for some graphs in which the maximum degree is $\omega(polylog(n))$ ([13]).

In this paper, we present a randomized quasi-fully dynamic algorithm that takes $O(\log^4 n)$ amortized expected time per operation. This is the largest class of dynamic operations for which a polylog time bound per operation is currently known for biconnectivity on general graphs.

**Towards Fully Dynamic Algorithms**: In section 5, we analyze the complexity of fully dynamic algorithms obtained from our quasi-fully dynamic algorithms by implementing the invalid deletions in a natural way. We show that these algorithms can take $\Omega(n)$ time for certain operations. We also show that if we use a uniform random spanning tree, the worst case complexity is $\Omega(\sqrt{n})$ per operation. We leave open the possibility that some other natural extension of our quasi-dynamic algorithms could give fully dynamic algorithms that run in poylog time.

## 2 2-Edge Connectivity

In this section, we present a straightforward quasi-fully dynamic algorithm for 2-edge connectivity. Queries ask whether a given pair of vertices, $u$ and $v$ are 2-edge connected (or equivalently, whether there are at least two edge-disjoint paths between $u$ and $v$). The algorithm takes $O(\log n)$ worst-case time for insertions, (valid) deletions and queries.

Let $G = (V, E)$ be a graph and $F \subseteq E$ be a spanning forest of $G$. We use $F_{uv}$ to denote the tree path between vertices $u$ and $v$. Edges in $F$ will be called *tree edges* and edges in $E - F$ will be called *nontree edges*. A tree edge $e$ is said to be *covered* (with respect to $F$) by a nontree edge $(u, v)$ iff $e$ lies on $F_{uv}$. Equivalently, $e$ lies on the fundamental cycle of nontree edge $(u, v)$. For an edge $e \in E$, we define $CoverSet_F(e) = \{e' \in E - F : e$ lies on the fundamental cycle of $e'$ w.r.t. $F\}$. Observe that for a nontree edge $e \in E - F$, $CoverSet_F(e) = \{e\}$, and for a cutedge $e \in F$, $CoverSet_F(e) = \emptyset$. We sometimes use $cover_F(e)$ to denote $|CoverSet_F(e)|$. Throughout this paper, unless otherwise mentioned, covering will be with respect to $F$ only and the subscript $F$ will be dropped when there is no ambiguity.

**Fact 1** *[5] Two vertices $u$ and $v$ are 2-edge connected iff $cover_F(e) \geq 1$ for every edge $e \in F_{uv}$.*

We store $F$ in a dynamic tree data structure ([19]) with edge costs representing the cover values. The basic idea behind our algorithm is that the cover values of the tree edges (which are sufficient to answer the queries) can be maintained easily under insertions and valid deletions. During insertion (or deletion) of nontree edges we add $+1$ (or $-1$) to the cover values on the corresponding tree path. Deletion of a tree edge is allowed only if it is a cut edge. Such a deletion will not affect the cover values of other tree edges. This is because such a tree edge does not lie on the fundamental cycle of any nontree edge. Hence its removal does not change any fundamental cycles and therefore does not change the cover values. To answer a query we just need to check the minimum cover value on the corresponding tree path. The implementations of the operations are briefly described below.

**Query**$(u, v)$**:** If $u$ and $v$ are in different trees, then return *no*. If they are in the same tree, then perform $evert(v)$ followed by $min\_cost(u)$. If the minimum value is zero, then return *no*. Otherwise return *yes*.

**Insert**$(u, v)$**:** If $u$ and $v$ are in the same tree, then mark $(u, v)$ as a nontree edge and perform $evert(v)$ and $add\_cost\_path(u, 1)$. If $u$ and $v$ are in different trees, then mark $(u, v)$ as a tree edge and modify $F$ as follows : perform $evert(u)$, followed by $link(u, v)$; set $cover(u, v) \leftarrow 0$.

**Delete**$(u, v)$**:** If $(u, v)$ is a nontree edge, then perform $evert(v)$ followed by $add\_cost\_path(u, -1)$. If $(u, v)$ is a tree edge and if it is not a cut edge, then report *invalid deletion*. If it is a cut edge then modify $F$ as follows: perform $evert(v)$ followed by $cut(u)$.

**Complexity:** To check whether a tree edge is a cut edge, we can just check whether its cover value is zero. Testing whether $u$ and $v$ are in the same tree can be done by performing $evert(v)$ and checking whether $root(u) = v$. Thus each of the insert, delete and query operations uses only a constant number of dynamic tree operations, and hence takes $O(\log n)$ worst case time. If amortized $O(\log n)$ time is sufficient, then a simpler implementation of dynamic trees can be used (see section 6).

**Other Types of 2-Edge Connectivity Queries:** We note that our algorithm can be extended to answer two other types of 2-edge connectivity queries as well.

Firstly, we can test whether a given edge $e$ is a cut edge. For this, we just need to check whether $e \in F$ and $cover(e) = 0$. This takes $O(\log n)$ time per operation in the worst-case.

Secondly, we can also test whether the entire graph is 2-edge connected. This query is equivalent to asking whether there exists a cut edge in the entire graph. To answer this query, it suffices to find the global minimum cover value (i.e., $min_{e \in F} cover_F(e)$) and check whether it is zero. The original dynamic tree data structure (of [19]) supports the path minimum operation but does not support the global minimum operation. An extension of the dynamic tree data structure which also supports the global minimum operation, in $O(\log^2 n)$ worst-case time per operation, is outlined in the appendix (section 6).

### 2.1 Quasi-k-Edge Connectivity

In this subsection, we introduce the concept of quasi-$k$-edge connectivity which is a restriction of the concept of $k$-edge connectivity to the case where only the valid edge deletions are allowed. We show that quasi-$k$-edge connectivity information can be maintained easily with $O(\log n)$ worst case time per operation.

**Definition 1** *(Quasi-k-Edge Connectivity) Given a graph $G = (V, E)$ with a spanning forest $F$, two vertices $u, v \in V$ are said to be quasi-$k$-edge connected iff with respect to $F$, no quasi-fully dynamic sequence of $k-1$ edge deletions disconnects $u$ and $v$.*

Observe that $u$ and $v$ are quasi-2-edge connected (w.r.t. any spanning forest) iff they are 2-edge connected. For $k > 2$, it is possible that a pair of vertices $u$ and $v$ that are quasi-$k$-edge connected are not $k$-edge connected, since there could exist a set of $k-1$ edges (of which at least two are tree edges), whose deletion disconnects $u$ and $v$.

The relevance of this concept can be understood by considering graphs for which a core part remains fixed while updates occur on the remaining part. Assume that we can extract a (fixed) spanning tree $T$ from the core part of the graph. Then, all valid sequences of edge deletions are quasi-fully dynamic sequences. Further, if vertices $u$ and $v$ are quasi-$k$-edge connected, then at least $k-1$ valid deletions are needed on the current graph in order to reduce the number of paths between $u$ and $v$ to one, which is the minimum possible (since $T$ forms a core part of the graph).

**Lemma 1** *Two vertices $u, v$ are quasi-$k$-edge connected, w.r.t. a spanning forest $F$, iff $cover_F(e) \geq k-1$ for every edge $e \in F_{uv}$.*

**Proof:** ($\rightarrow$) Suppose there exists $e \in F_{uv}$ such that $cover_F(e) < k-1$. Then by deleting the edges in $CoverSet_F(e)$ followed by $e$, would disconnect $u$ and $v$. Clearly this is a sequence of at most $k-1$ valid edge deletions. Hence $u$ and $v$ cannot be quasi-$k$-edge connected w.r.t. $F$.

($\leftarrow$) Suppose $u$ and $v$ are not quasi-$k$-edge connected. Let $s = \langle e_1, e_2, \ldots e_{|s|} \rangle$ be a sequence of at most $k-1$ valid edge deletions that disconnects $u$ and $v$. Without loss of generality, assume that $e_{|s|} \in F_{uv}$. As the deletion of $e_{|s|}$ is valid at that point in the sequence, it follows that $CoverSet_F(e_{|s|}) \subseteq \{e_1, e_2, \ldots e_{|s|-1}\}$ and hence $cover_F(e_{|s|}) < k-1$, which is a contradiction.
□

The above lemma immediately implies a quasi-fully dynamic algorithm with $O(\log n)$ worst case time per operation. The update operations are implemented exactly as in the 2-edge connectivity case. A query asks whether $u$ and $v$ are quasi-$k$-edge connected (w.r.t. $F$). It can be answered as follows:

**Query**$(u, v)$**:** If $u$ and $v$ are in different trees, then return *no*. If they are in the same tree, then perform $evert(v)$ followed by $min\_cost(u)$. If the minimum value is less than $k-1$, then return *no*. Otherwise return *yes*.

Thus each of the insert, (valid) delete, quasi-$k$-edge connectivity queries can be performed in $O(\log n)$ worst-case time.

## 3 Cycle Equivalence

We present a simple quasi-fully dynamic randomized algorithm for the cycle equivalence problem (defined below) that takes $O(\log n)$ worst-case time for updates (insertions and valid deletions) and queries.

**Definition 2** *(Cycle Equivalence) Edges $e_1, e_2 \in E$ are cycle equivalent iff the set of cycles that contain $e_1$ is exactly the same as the set of cycles that contain $e_2$.*

Note that edges $e_1$ and $e_2$ are cycle equivalent iff $CoverSet(e_1) = CoverSet(e_2)$. A pair of edges $(e_1, e_2)$ will be called a *cut-edge pair* iff the removal of $e_1$ and $e_2$ increases the number of connected components in the graph. As observed in [8] two edges are cycle equivalent iff they are a cut-edge pair in the graph.

### 3.1 Isolating Lemma

In this subsection, we describe a simple probabilistic lemma which is the basis for our cycle equivalence algorithm. A *set system* $(S, Q)$ consists of a finite set $S$ of elements, i.e., $S = \{x_1, x_2, \ldots x_{|S|}\}$, and a family $Q$ of subsets of $S$, i.e., $Q = \{S_1, S_2, \ldots S_{|Q|}\}$ with $S_j \subseteq S$, for $1 \leq j \leq |Q|$. We assign a weight $w(x_i)$ to each element $x_i \in S$, and define the weight of a subset $S_j$ to be $\sum_{x_i \in S_j} w(x_i)$.

**Lemma 2** *[18]* **(Unique Set Isolating Lemma)** *Let $(S, Q)$ be a set system. If the elements in $S$ are assigned integer weights chosen randomly and uniformly from $[1 \ldots 2|S|]$, then $Pr$[Minimum weight subset in $Q$ is unique] $\geq 1/2$.*

The above lemma enables us to assign random polynomial weights such that a single subset in $Q$ is isolated from the other subsets in $Q$. We strengthen the above lemma so that we can isolate all subsets in $Q$. i.e., we want each subset in $Q$ to have a distinct weight. If exponential weights are allowed, this task becomes easy (even deterministically): just assign $w(x_i) = 2^i$, and represent each subset as a bit vector of length $|S|$. However as we restrict ourselves to polynomial sized weights only, the task is no longer trivial. The following lemma gives a randomized method for this task, using only polynomial sized weights.

**Lemma 3 (All Sets Isolating Lemma)** *Let $(S, Q)$ be a set system. Let $Z = |Q|^2|S|$. If the elements in $S$ are assigned integer weights chosen randomly and uniformly from $[1 \ldots Z]$, then $Pr$[every subset in $Q$ gets a distinct weight] $\geq 1/2$.*

**Proof:** Let $W : S \to [1 \ldots Z]$ be a uniformly random weight assignment. An element $y \in S$ is said to be $W$-**bad** if there exist distinct subsets $S_i, S_j$ in $Q$ such that $y \in S_i - S_j$ and $W(S_i) = W(S_j)$.

Consider a fixed element $y \in S$. Suppose we fix $W(x)$, for all the elements, $x$ in $S$, except $y$. $W(y)$ is still to be chosen. We are interested in knowing the number of values of $W(y)$ that would make it $W$-bad. For each pair $(S_i, S_j)$ such that $y \in S_i - S_j$, there is at most one value of $W(y)$ that would make $W(S_i) = W(S_j)$. Hence there are at most $|Q|^2/2$ values of $W(y)$ that could cause $y$ to become $W$-bad. Hence $Pr_W[y \text{ is } W\text{-bad}] \leq |Q|^2/2Z = 1/(2|S|)$. Here, the subscript $W$ indicates that the probability is over all choices of $W$. We now have,

$$
\begin{aligned}
&Pr_W[\text{Some two subsets in } Q \text{ get the same weight with assignment } W] \\
&\quad = Pr_W[\exists S_i, S_j \in Q \text{ such that } S_i \neq S_j \text{ and } W(S_i) = W(S_j)] \\
&\quad = Pr_W[\exists x \in S \text{ such that } x \text{ is } W\text{-bad}] \\
&\quad \leq \textstyle\sum_{x \in S} Pr_W[x \text{ is } W\text{-bad}] \\
&\quad \leq 1/2.
\end{aligned}
$$

In the second step above, we used the fact that any element $x \in (S_i - S_j) \cup (S_j - S_i)$ will be $W$-bad. In the third step, we used the Boole's inequality.

□

**Corollary:** Let $Z = |Q|^2|S|$. If the elements in $S$ are assigned integer weights uniformly at random from $[1 \ldots NZ]$, then $Pr$[some two distinct subsets in $Q$ get the same weight] $\leq 1/(2N)$.

### 3.2 Algorithm for Cycle Equivalence

Let $S = E - F$ be the set of nontree edges in $G$ and let $Q = \{CoverSet(e) : e \in E\}$. We note that $(S, Q)$ is a set system, with $|S| \leq |E| < n^2$ and $|Q| \leq |E| < n^2$. Hence, we can choose $Z = n^6$.

For each nontree edge of $G$, we assign a random weight chosen uniformly from $[1 \ldots NZ]$. Let $W$ be this random weight assignment. For an edge $e \in E$, define $cost(e) = W(CoverSet(e))$. Then it follows from the isolating lemma that with high probability, $e_1$ and $e_2$ are cycle equivalent iff $cost(e_1) = cost(e_2)$. Our quasi-fully dynamic algorithm is based on this idea, and is outlined below.

For a nontree edge $e \in E - F$, as $CoverSet(e) = \{e\}$, we have $cost(e) = w(e)$. Note that the cost of a nontree edge does not change as long as that edge is in the graph. For a tree edge $e \in F$, we maintain $cost(e)$ using a dynamic tree data structure. We store $F$ in a dynamic tree data structure with costs on (tree) edges. The update operations (insertions and valid deletions) are very similar to those in 2-edge connectivity. The only difference is the following: when an edge $e = (u, v)$ is inserted as a nontree edge, then we pick $w(e)$ uniformly at random from $[1..NZ]$ and perform $add\_cost\_path(F_{uv}, w(e))$ (instead of $add\_cost\_path(F_{uv}, 1)$ in the 2-edge connectivity case). Correspondingly, when a nontree edge $e = (u, v)$ is deleted, we perform $add\_cost\_path(F_{uv}, -w(e))$. For a $query(e_1, e_2)$, it suffices to check whether $cost(e_1) = cost(e_2)$. For a tree edge $e$, the $cost(e)$ can be obtained from the dynamic tree in $O(\log n)$ time.

**Complexity and Error Probability:** If we choose $N$ to be a polynomial in $n$, the costs will still be polynomial in $n$ (i.e., they are only $O(\log n)$ bits long). Hence addition operations on these can still be done in $O(1)$ time, assuming that each word in memory is of size $\Theta(\log n)$. Thus updates and queries can be performed in $O(\log n)$ worst-case time.

The above algorithm is a Monte-Carlo randomized algorithm. For each query, there is a nonzero probability that it may return *yes* even when $e_1$ and $e_2$ are not cycle equivalent. The probability of error can be made as small as desired. To be precise, it can be made $O(n^{-c})$ for any constant $c$, by choosing $N = n^c$. When a sequence of operations is performed on the graph, the error probabilities will add up. To keep the overall error probability small, it suffices to modify the range of the weights suitably. For instance, if we know an upper bound $M$ on the number of operations (insertions, valid deletions, queries) to be performed, then by choosing the weights from $[1..(MNZ)]$, we can show that $Pr$[Algorithm gives a wrong answer some time during the sequence] $\leq 1/(2N)$. In this case, the weights will be $O(\log M + \log n)$ bits long and each addition operation on these weights would take $O(max\{1, \log M / \log n\})$ time. Hence each operation would cost $O(\log n + \log M)$ time in the worst-case.

### 3.3 Relation to 3-Edge Connectivity

A 3-edge connectivity query asks whether there are 3 edge-disjoint paths between two given vertices $u$ and $v$. The following lemma gives a characterization of the 3-edge connectivity property and reveals its close connection to the cycle equivalence problem.

**Lemma 4** *Two vertices $u$ and $v$ are not 3-edge connected iff one of the following holds:*

1. *There exists a tree edge $e$ on $F_{uv}$ such that $cover(e) \leq 1$; OR*

2. *There exists a tree edge $e$ on $F_{uv}$ and another tree edge $y$ on $F - F_{uv}$ such that $e$ and $f$ are cycle equivalent.*

**Proof:** (Sketch) Condition (1) of the requirement accounts for cut edges $e$ (i.e., $cover(e) = 0$) and cut edge pairs $(e, f)$ where $e$ is a tree edge and $f$ is a nontree edge (i.e., $cover(e) = 1$).

Condition (2) accounts for cut edge pairs $(e, f)$ where both $e$ and $f$ are tree edges. By the result in [8], two edges $e$ and $f$ are cycle equivalent iff they are cut edge pair. A cut edge pair $(e, f)$, of tree edges, separates vertices $u$ and $v$ iff one of them lies on $F_{uv}$ and the other lies on $F - F_{uv}$.
□

Condition (1) can be checked easily by storing $F$ as a dynamic tree data structure augmented with edge costs as the cover value (as in the 2-edge connectivity algorithm, section 2). The only remaining task for obtaining a quasi-fully dynamic algorithm for 3-edge connectivity is to be able to check condition (2). In view of its close relation to the cycle equivalence problem, it would be interesting to see if this can be tested efficiently.

## 4 2-Vertex Connectivity

In the 2-vertex connectivity problem, a query asks whether two given vertices $u$ and $v$ are in the same biconnected component of $G$ (or equivalently, whether there exist two vertex-disjoint paths between $u$ and $v$). In this section, we present a quasi-fully dynamic algorithm for 2-vertex connectivity.

Let $F$ be a spanning forest of $G$. For a vertex $v$, we use $N_F(v)$ to denote the set of neighbors of $v$ in $F$. Let $x, y \in N_F(v)$. We use $(F - v)_x$ to denote the subtree of $F - v$ that contains $x$. We define the *neighborhood graph* of $v$, denoted $NG_F(v)$, as follows: the vertex set is $N_F(v)$; an edge $(x, y)$ is present in $NG_F(v)$ iff there exists a nontree edge in $G$ between $(F - v)_x$ and $(F - v)_y$. The following lemma is a direct consequence of the above definition.

**Lemma 5** *Suppose $x, y \in N_F(v)$. Then $x$ and $y$ belong to the same biconnected component of $G$ iff $x$ and $y$ belong to the same connected component of $NG_F(v)$.*

Our dynamic algorithm uses some additional parameters which are defined below. For $x, y \in N_F(v)$, we use $D_v(x, y)$ to denote the number of nontree edges in $G$ from $(F - v)_x$ to $(F - v)_y$. We also define $C_v(x, y)$ to be 1 if $x, y$ belong to the same connected component of $NG_F(v) - (x, y)$ and 0 otherwise. Here, $NG_F(v) - (x, y)$ denotes the graph obtained by deleting the edge $(x, y)$, if it exists, from $NG_F(v)$. Observe that $C_v(x, y)$ captures the existence of an indirect path (i.e., using more than one edge) from $x$ to $y$ in $NG_F(v)$ while $D_v(x, y)$ captures the existence of a direct path (i.e., using exactly one edge) from $x$ to $y$ in $NG_F(v)$. Finally, we use $L_v(x, y)$ to denote $C_v(x, y) + D_v(x, y)$. By the above lemma, $x$ and $y$ belong to the same biconnected component of $G$ iff $L_v(x, y) > 0$. Strictly speaking, the above three definitions must specify the underlying spanning forest $F$; but for ease of notation we drop $F$ when there is no ambiguity.

We store $F$, augmented with costs at vertices, in a dynamic tree data structure (see section 6, theorem 2). The costs on the vertices have the following interpretation: If vertex $v$ lies on a solid path, with $(v, x)$ and $(v, y)$ as its solid edges, then $cost(v) = L_v(x, y)$. Otherwise (i.e., if $v$ has atmost one solid edge incident on it) $cost(v)$ is arbitrary.

**Fact 2** *[7] Two vertices $u$ and $v$ are biconnected iff after making $F_{uv}$ a solid path, no internal vertex on $F_{uv}$ has cost 0.*

### 4.1 Overview of the Algorithm

To determine whether $u$ and $v$ are biconnected, we will transform the tree path $F_{uv}$ into a solid path, and check whether the minimum cost on this solid path is zero. We will see later that even for inserting (or deleting) a nontree edge $(u, v)$ it suffices to transform the tree path $F_{uv}$ into a solid path and increment (or decrement) the cost of each vertex on this solid path by one.

Hence the basic requirement is to convert the tree path $F_{uv}$ into a solid path. Using the *expose* and *evert* operations, this takes $O(\log n)$ amortized time, provided no costs are updated. However, our data structure has to update the cost of a vertex $v$ whenever the edges incident on $v$ change from solid to dashed or vice-versa. The basic dynamic tree operation that converts dashed edges into solid edges (and vice-versa) is the *splice* operation (see section 6). We will show that the splice operation can be extended so that the costs at the vertices involved in the splice operation can be correctly updated in roughly $O(\log^3 n)$ amortized expected time. This would imply that each of the update/query operations takes roughly $O(\log^4 n)$ amortized expected time.

**Additional Data Structures:** The *modified neighborhood graph* of a vertex $v$, denoted by $NG'_F(v)$, is defined as follows: if vertex $v$ has two solid edges, say $(v, x)$ and $(v, y)$, then $NG'_F(v) = NG_F(v) - (x, y)$; otherwise $NG'_F(v) = NG_F(v)$. Note that $NG'_F(v)$ changes if the solid edges incident on $v$ change. For each vertex $v$, we store the modified neighborhood graph, $NG'_F(v)$, in a randomized fully dynamic connectivity data structure [9].

For each vertex $v$, we also use a *dictionary* $DDS_v$ to store the nonzero $D_v(x, y)$ values. To keep the space requirements small, the zero $D_v(x, y)$ values are not stored. A dictionary is a data structure that supports insert, delete and search operations on a set of items. Dictionary implementations which take $O(\log r)$ worst case time for each operation are well known ($r$ is the number of items in the dictionary). More specifically, the $DDS_v$ stores the set $\{D_v(x, y) : x, y \in N_T(v) \text{ and } D_v(x, y) > 0\}$. The items in $DDS_v$ are indexed by the $(x, y)$ tuple.

### 4.2 The Update and Query Operations:

In the next subsection we present an augmented splice operation that correctly updates the costs on the vertices when their incident edges change from dashed to solid or vice-versa. In this subsection, we present simple implementations of the update and query operations assuming that the splice operation (and hence the *expose* and *evert* operations) correctly updates the costs.

**Query**$(u, v)$**:** If $u$ and $v$ are in different trees, then return *no*. If $u$ and $v$ are in the same tree, then perform $evert(v)$ followed by $min\_cost(u)$. This gives the minimum cost on the solid path $F_{uv}$. If the minimum value is zero, then return *no*. Otherwise return *yes*.

**Insert**$(u, v)$**:** If $u$ and $v$ are in the same tree, then mark $(u, v)$ as a nontree edge. We convert the tree path $F_{uv}$ into a solid path and add 1 to all the costs on this path as follows: perform $evert(v)$ followed by $add\_cost\_path(u, 1)$. If $u$ and $v$ are in different trees, then mark $(u, v)$ as a tree edge and modify $F$ (and the neighborhood graphs) as follows: perform $evert(u)$, followed by $link(u, v)$; set $cover(u, v) \leftarrow 0$; Add isolated vertices $u$ and $v$ in $NG'_F(v)$ and $NG'_F(u)$ respectively.

**Delete**$(u, v)$**:** If $(u, v)$ is a nontree edge, then we make the path $F_{uv}$ solid and decrement the costs along the path as follows: perform $evert(v)$ followed by $add\_cost\_path(u, -1)$. If $(u, v)$ is a tree edge and if it is not a cut edge then report *invalid deletion*. If it is a cut edge, we modify $F$ (and the neighborhood graphs) as follows: perform $evert(v)$, followed by $cut(u)$; Remove the isolated vertices $u$ and $v$ from $NG'_T(v)$ and $NG'_T(u)$ respectively.

Note that each of the above operations invokes a constant number of dynamic tree operations.

**Proof of Correctness:** To prove that the queries give the correct answer, it suffices to argue that for every vertex $v$, as long as edges $(v, x)$ and $(v, y)$ remain solid, $cost(v)$ is always equal to $L_v(x, y)$.

**Lemma 6** *Suppose that at a vertex $v$, edges $(v, x)$ and $(v, y)$ become solid at some time and that the $cost(v)$ is correctly set to $L_v(x, y)$ at that time. Then as long as both $(v, x)$ and $(v, y)$ remain solid, after any sequence of updates and queries, $cost(v)$ will be equal to $L_v(x, y)$.*

**Proof:** As $L_v(x, y) = C_v(x, y) + D_v(x, y)$, we will show that $C_v(x, y)$ will not change at all and that changes in $D_v(x, y)$ are correctly reflected in $cost(v)$.

Firstly, note that $NG'_F(v)$ does not change as long as both $(v, x)$ and $(v, y)$ remain solid. This is because, the only way $NG'_F(v)$ can change is if a nontree edge between $(F - v)_w$ and $(F - v)_z$ is inserted or deleted for some $w, z \in N_F(v)$ $(w, z \neq x, y)$. But when this happens edges $(v, w)$ and $(v, z)$ become solid and atleast one of $(v, x)$ and $(v, y)$ becomes dashed. This implies that $C_v(x, y)$ does not change.

Secondly, insertion (deletion) of a nontree edge from $(F-v)_x$ to $(F-v)_y$ increments (decrements) both $cost(v)$ and $D_v(x, y)$.

□

To complete the proof of correctness, it remains to ensure that $cost(v)$ is correctly initialized each time the solid edges at $v$ change. We do this in the next subsection.

### 4.3 Modifying the Splice Operation

The original splice operation is briefly outlined in the appendix (section 6). In this subsection, we look at the modifications to the splice operation which are needed to ensure that $cost(v)$ is correctly initialized each time the solid edges ar $v$ change.

Let $v$ be a parent of $w$ and also let $(v, y)$ and $(v, x)$ be the solid edges incident on $v$. We only consider the case where both $x$ and $y$ exist. The other cases can be handled in a similar fashion. The $splice(w)$ operation converts the dashed edge $(w, v)$ to solid and the solid edge $(v, y)$ to dashed. During this conversion, we need to store and update the parameters corresponding to the outgoing solid edge $(v, y)$. More specifically, we update the modified neighborhood graph of $v$ (i.e., $NG'_F(v)$) and we also store the parameter $D_v(x, y)$ in $DDS_v$. Similarly for the incoming solid edge $(w, v)$, we update $NG'_F(v)$ and compute the new $cost(v)$ using $DDS_v(w, x)$ and $NG'_F(v)$. The extended *splice* operation, which performs all these updates, is outlined below in pseudocode.

**Splice**($w$):
  { $parent(w) = v$; Edge $(w, v)$ is solid }
  { Current solid edges at $v$ are $(v, x)$ and $(v, y)$ }
**begin**
  Perform the steps of the original splice operation (see section 6);
  {For outgoing solid edge $(v, y)$ }
    Compute $C_v(x, y)$ by a query $Connected(x, y)$? in $NG'_F(v)$;
    $D_v(x, y) \leftarrow cost(v) - C_v(x, y)$;
    **if** $D_v(x, y) > 0$ **then**
      $insert\_edge(x, y)$ into $NG'_F(v)$;
      $insert\_item(x, y, D_v(x, y))$ into $DDS_v$;
    **endif**
  { For incoming solid edge $(w, v)$ }
    $delete\_edge(w, x)$ from $NG'_F(v)$ if this edge exists;
    Compute $C_v(x, w)$ by a query $Connected(x, w)$? in $NG'_F(v)$;
    Search for $D_v(x, w)$ in $DDS_v$;
    **if** found **then** $delete\_item(x, w, D_v(x, w))$ from $DDS_v$;

**else** $D_v(x, w) \leftarrow 0$
**endif**
$cost(v) \leftarrow C_v(x, w) + D_v(x, w)$
**end**

Let $G = (V, E)$ be a graph with $n$ vertices and $m_0$ initial edges. For the sake of simplicity, we keep all $n$ vertices in $NG'_F(v)$, for each $v \in V$, instead of just $|N_F(v)|$. Moreover, for the sake of analysis, we build the $NG'_F(v)$ structures as follows: We first put $n$ isolated vertices in each of them. We then insert the $m_0$ edges of $G$ one at a time into our quasi-fully dynamic 2-vertex connectivity structure, which causes the appropriate modifications of $NG'_F(v)$. This ensures that each of the $NG'_F(v)$ starts with no initial edges.

**Theorem 1** *Let $G = (V, E)$ be a graph with $n$ vertices and $m_0$ initial edges.*

1. **Space Bound:** *Our quasi-fully dynamic algorithm for 2-vertex connectivity uses $O(n^2 \log n)$ space.*
2. **Time Bound:** *The expected running time for a sequence of $k$ insert, (valid) delete and biconnectivity query operations is $O(n^2 \log n + (k + m_0) \log^4 n)$.*

**Proof:**
In our proof we will make use of the following result:

**Fact 3** *(Fully Dynamic Connectivity Data Structure) [9] Let $G = (V, E)$ be a graph with $n$ vertices and $m_0$ initial edges. Then the expected time for a sequence of $k$ insert, delete and connectivity queries on $G$ is $O(k \log^3 n + m_0 \log n)$. At an instant when $G$ has $m$ edges, this data structure uses $O((m + n) \log n)$ space.*

We now prove Theorem 1:
**Space Bound:** The dynamic tree data structure uses $O(n)$ space. We will show that the space requirement of all the fully dynamic connectivity data structures and all the dictionaries put together is $O(n^2)$.

Let $d_v$ denote $|N_F(v)|$, the degree of $v$ in $F$. The number of edges in $NG'_F(v)$ is atmost $O(d_v^2)$. Hence, by fact 3, the fully dynamic connectivity data structure for $NG'_F(v)$ needs $O((n + d_v^2) \log n)$ space. On the other hand, the dictionary $DDS_v$ stores atmost $d_v^2$ items and hence uses $O(d_v^2)$ space. Hence the space taken by all these put together is $O(\sum_{v \in V} d_v^2 \log n)$ which is $O(n^2 \log n)$, as $\sum_{v \in V} d_v \leq 2(n - 1)$.
**Time Bound:** Initializing each of the $NG'_F(v)$ with $n$ isolated vertices takes $O(n^2 \log n)$ time. As the initial $m_0$ edges are also inserted using the *insert* operation, we can analyze the entire algorithm as a sequence of $k + m_0$ operations.

Firstly, as each insert/delete/query causes $O(\log n)$ splices (see subsection 6), the total number of splice operations is $O((k + m_0) \log n)$. This implies that the total number of updates/queries to all the dictionary and connectivity data structures (i.e, $NG'_F(v)$) is atmost $O((k + m_0) \log n)$. For the dictionary, each update/query takes $O(\log n)$ time.

It remains to analyze the time spent over all the connectivity structures. Let $k'_v$ be the number of update/query operations on $NG'_F(v)$. ¿From the above arguments, it follows that $\sum_{v \in V} k'_v$ is $O((k + m_0) \log n)$. By fact 3, a sequence of $k'_v$ update/query operations on $NG'_F(v)$ takes an expected time of $O(k'_v \log^3 n)$ as the initial $NG'_F(v)$ has zero edges. Summing it up, the expected time spent over all connectivity structures is $O((k + m_0)\log^4 n)$.

□

Thus if $k = \Omega(m_0 + n^2)$, the above theorem implies an amortized expected cost of $O(\log^4 n)$ per operation.

## 5 Towards Fully Dynamic Algorithms

In this section, we analyze the performance of fully dynamic algorithms obtained by a natural extension of our quasi-fully dynamic algorithms. To extend the quasi-fully dynamic algorithms to fully dynamic algorithms we need to show how the invalid tree edge deletions are to be handled. Recall that, for a tree edge $e \in F$, $CoverSet_F(e) = \{e' \in E - F : e'$ covers $e$ with respect to $F\}$. Deleting a tree edge $e$ with $CoverSet(e) = \emptyset$ is a valid operation and it is handled by the quasi-fully dynamic algorithm. On the other hand, deletion of a tree edge $e$ with $CoverSet(e) \neq \emptyset$ is an invalid operation in the quasi-fully dynamic case.

One natural way of implementing an invalid deletion of a tree edge $e$ is the following: first find $CoverSet(e)$, delete the nontree edges in $CoverSet(e)$ one at a time, then delete the tree edge $e$ (which is now a cut edge), and then reinsert the edges of $CoverSet(e)$ one at a time. Pick a random edge of $CoverSet(e)$ to replace $e$ in $F$. The complexity of this implementation is analyzed below.

**Finding CoverSet$(e)$:** We store $G$ in a randomized fully dynamic connectivity data structure ([9]) with $F$ as the spanning forest. For a tree edge $e$, the $CoverSet(e)$ can be found as follows: let $e'$ be the random edge found by the algorithm in [9] to replace $e$ in $F$. Remove $e'$ and find its next replacement, remove that and so on until no more replacements exist. All these removed replacement edges form the $CoverSet(e)$.

**Overall Complexity:** Using the above method for finding the $CoverSet(e)$, the time taken to delete $e$ is $O(|CoverSet(e)| \cdot polylog(n))$. However, in the worst case, $|CoverSet(e)|$ can be $\Omega(m)$ as shown by the example below. This implies that for the above natural implementation of invalid deletions, there exist sequences of operations in which the expected cost of deleting an edge is $\Omega(m)$.

**Worst Case Example for this Algorithm:** Consider the following sequence of insertions: we first create a path $P$ of length $n-1$, with vertices $1, 2 \ldots n$ in that order. Let $e$ be the middle edge, i.e., $(n/2, n/2+1)$, of this path (assuming $n$ is even). Now we add edges from every vertex in $\{1, 2 \ldots n/2\}$ to every vertex in $\{n/2+1, \ldots n\}$. All these edges will be nontree edges and form the $CoverSet(e)$. Hence $|CoverSet(e)|$ is $\Omega(m)$.

□

Interestingly, a similar result holds even if $F$ were chosen uniformly at random from the set of all labeled spanning trees of $G$ (see, e.g., [2] for properties and construction of random spanning trees).

**Lemma 7** *Let $U$ be the uniform distribution on the set $\mathcal{F}$ of all labeled spanning forests of $G = (V, E)$ (i.e., $U[F_i] = 1/|\mathcal{F}|$). Then, $\exists G$ and $\exists e \in E$, such that the expected cost of deleting $e$ using the fully dynamic algorithm described above is $\Omega(\sqrt{n})$.*

**Proof:** We first introduce some notations and definitions. Let $\mathcal{F} = \{F_1, F_2, \ldots F_N\}$ be the set of all labelled spanning forests of $G = (V, E)$. Consider an edge $e \in E$. Let $\mathcal{F}_e = \{F \in \mathcal{F} : e \in F\}$, the set of forests containing $e$. Recall that $CoverSet_F(e) = \{e\}$ for $F \notin \mathcal{F}_e$ (because $e \notin F$). For an edge $e$, let $h(U, e)$ denote the expected cost of deleting $e$, when the spanning forest $F$ is chosen according to $U$. Then we have, $h(U, e) = E_U[\text{cost of deleting edge } e] = \sum_{F_i \in F} Pr_U[F = F_i] \cdot |CoverSet_{F_i}(e)| \cdot polylog(n)$.

Consider the graph $G = (V, E)$ consisting of $t$ cycles $C_1, C_2, \ldots C_t$ (each consisting of $\frac{n}{t} + 2$ edges) such that each of them contains a fixed edge $e = (u, v)$ and $C_i - \{u, v\}$, $C_j - \{u, v\}$ are vertex disjoint for all $i \neq j$ $(1 \leq i, j \leq t)$. The parameter $t$ will be chosen later. We have $|V| = n+2$ and $|E| = n + t + 1$. As $G$ is connected, the number of nontree edges will be $t$.

If $e$ is chosen to be a tree edge, then $E - T$ must contain exactly one edge from each $C_i - \{e\}$ for $1 \leq i \leq t$. Hence the number of spanning trees that contain $e$, $|\mathcal{F}_e| = (\frac{n}{t} + 1)^t$. For each tree $T \in \mathcal{F}_e$, $|CoverSet_T(e)| = t$.

If $e$ is not a tree edge, then $E - T$ must contain no edges from $C_k - \{e\}$ for exactly one $k$ and exactly one edge from $C_i - \{e\}$ for all $i \neq k$ (here, $1 \leq i, k \leq t$). Hence $|\mathcal{F} - \mathcal{F}_e| = t(\frac{n}{t} + 1)^{t-1}$. For each tree $T \in \mathcal{F} - \mathcal{F}_e$, $|CoverSet_T(e)| = 1$. Using the fact that $Pr_U[F = F_i] = 1/|\mathcal{F}|$, we now have,

$$h(U, e) = \frac{t(\frac{n}{t} + 1)^t + t(\frac{n}{t} + 1)^{t-1}}{(\frac{n}{t} + 1)^t + t(\frac{n}{t} + 1)^{t-1}} \cdot polylog(n) = \frac{n + 2t}{\frac{n}{t} + t + 1} \cdot polylog(n).$$

Choosing $t = \sqrt{n}$, we get $h(U, e) = \Omega(\sqrt{n})$.

□

We leave it as an open question to determine whether there exists some other reasonable extension of our quasi-fully dynamic algorithms that leads to fully dynamic algorithms that run in polylog time per operation.

## References


[1] David Alberts, Giuseppe Cattaneo, and Giuseppe F. Italiano. An empirical study of dynamic graph algorithms. In *Proceedings of the Seventh Annual ACM SIAM Symp. on Discrete Algorithms*, pages 192–201, 1996.

[2] David A. Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *Siam J. Disc. Math.*, 3(4):450–465, November 1990.

[3] D. Eppstein, Z. Galil, and G. Italiano. Improved sparsification. Technical Report 93-20, University of California at Irvine, Dept of Information and Computer Science, 1993.

[4] J. Feigenbaum and Sampath Kannan. *Handbook of Discrete and Combinatorial Mathematics*, chapter Dynamic Graph Algorithms, pages 583–591. 1995.

[5] G.N. Frederickson. Ambivalent data structures for dynamic 2-edge connectivity and $k$ smallest spanning trees. In *Proceedings of 32nd Symp. on Foundations of Computer Science*, pages 632–641, 1991.

[6] R. Gupta and M.L. Soffa. Region scheduling. In *Proc. 2nd International Conference on Supercomputing*, pages 141–148, 1987.

[7] M. Rauch Henzinger. Fully dynamic biconnectivity in graphs. In *Proceedings of 33rd Symp. on Foundations of Computer Science*, pages 50–59, 1992.

[8] M. Rauch Henzinger. Fully dynamic cycle equivalence in graphs. In *Proceedings of 35th Symposium on Foundations of Computer Science*, pages 744–755, 1994.

[9] M. Rauch Henzinger and V. King. Randomized dynamic algorithms with polylogarithmic time per operation. In *Proceedings of 27th Annual Symp. on Theory of Computing*, pages 519–527, 1995.

[10] M. Rauch Henzinger and J. A. La Poutre. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. In *Proceedings of Third Annual European Symposium on Algorithms (ESA)*, pages 171–184, 1995.

[11] M.R Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proceedings 36th Symp. on Foundations of Computer Science*, pages 664–672, 1995.

[12] Richard Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. In *Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[13] Valerie King. Personal communication, July 1996.

[14] Madhukar R. Korupolu and Vijaya Ramachandran. Quasi-fully dynamic algorithms for two-connectivity, cycle equivalence and related problems. In *Proceedings of the Fifth Annual European Symposium on Algorithms (ESA'97.* Springer-Verlag LNCS. September 1997. To appear.

[15] Madhukar R. Korupolu. Randomized fully dynamic two edge connectivity: A variant of the Henzinger-King sketch. Manuscript, Univ of Texas at Austin, May 1997.

[16] J.A. La Poutre. Maintenance of 2- and 3- connected components of graphs, part ii: 2- and 3-edge connected components and 2-vertex connected components. Technical Report RUU-CS-90-27, Utrecht University, 1990.

[17] J.A. La Poutre and J. Westbrook. Dynamic two-connectivity with backtracking. In *Proceedings of 4th Symp. on Discrete Algorithms*, pages 204–212, 1994.

[18] Ketan Mulmuley, U.V. Vazirani, and V.V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[19] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[20] R.E Tarjan and Valdes Jacobo. Prime subprogram parsing of a program. In *Conference record of the Seventh Annual ACM Symp. on Principles of Programming Languages*, pages 28–30, 1980.

[21] J. Westbrook. *Algorithms and Data Structures for Dynamic Graph Problems.* PhD thesis, Dept of Computer Science, Princeton University, Princeton, NJ, 1989.

[22] J. Westbrook and R.E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Jl. Computing*, 18:1–11, 1989.

[23] J. Westbrook and R.E. Tarjan. Maintaining bridge connected and biconnected components online. *Algorithmica*, pages 433–464, 1992.

## 6 APPENDIX

### Data Structure for Dynamic Trees - Review and Extension

For the sake of completeness, we give a brief review of the data structure for dynamic trees given by [19]. For further details, see [19]. We also give an extension of this data structure that also supports the global minimum operation.

The data structure for dynamic trees given in [19] maintains a forest of vertex-disjoint rooted trees, with real valued costs on the edges, under the eight operations described below. The data structure can be easily modified to handle costs on vertices, instead of on the edges.

- **parent**(*vertex* $v$) : Return the parent of $v$. If $v$ is a tree root, return *Null*.
- **root**(*vertex* $v$) : Return the root of the tree containing $v$.
- **cost**(*edge* $e$) : Return the cost of the edge $e$ (if the edge $e$ exists).
- **mincost**(vertex $v$) : Return the vertex $w$ closest to $root(v)$ such that the edge $(w, parent(w))$ has the minimum cost among edges on the tree path from $v$ to $root(v)$. Assumes that $v$ is not a tree root.
- **add_cost_path**(*vertex* $v$, *real* $k$) : Modify the costs of all edges on the tree path from $v$ to $root(v)$ by adding $k$ to the cost of each edge.
- **link**(*vertex* $v, w$, *real* $k$) : Combine the trees containing $v$ and $w$ by adding the edge $(v, w)$ of cost $k$, making $w$ the parent of $v$. This operation assumes that $v$ and $w$ are in different trees and $v$ is a tree root.
- **cut**(*vertex* $v$) : Divide the tree containing $v$ into two trees by deleting the edge $(v, parent(v))$. This operation assumes that $v$ is not a tree root. Assumes that $v$ is a not a tree root.
- **evert**(*vertex* $v$): Modify the tree containing $v$ by rerooting it at $v$; i.e., reverse the direction of every edge on the path from $v$ to the original root.

The dynamic tree data structure of [19] partitions the edges into two kinds: *solid* and *dashed* such that atmost one solid edge enters any vertex. Thus, every vertex is incident to atmost two solid edges. The solid edges define a collection of solid paths which partition the vertices. These solid paths are connected by dashed edges. The solid paths are stored in *balanced* binary trees which makes it possible to have a fast implementation of path operations (like, $root(v)$, $add\_cost\_path(v)$, $mincost(v)$ and $evert(v)$).

To implement any of the above operations, the basic requirement is to convert a given tree path (usually the tree path from $v$ to $root(v)$) into a solid path. The dynamic tree data structure implicitly allows for such a transformation, using the *expose* operation. The *expose* operation is implemented using a basic operation called *splice* which essentially converts a dashed edge into a solid edge. The semantics of **splice** are given below. In section 4, we augment the splice operation to handle the requirements of our quasi-fully dynamic algorithm for 2-vertex connectivity.

**Splice**(*vertex* $w$) : Let $parent(w) = v$. This operation assumes that the edge $(w, v)$ is dashed. It converts the edge $(w, v)$ to solid. If there was a solid edge $(y, v)$ coming into $v$, then it will be converted to dashed. This operation also extends the solid path $p$ coming in to $w$, by appending to $p$ the edge $(w, v)$ and the solid path leaving $v$.

The following theorem implies that the dynamic tree operations can be performed in $O(\log n)$ amortized time per operation. We use this implementation of dynamic trees for our 2-vertex connectivity algorithm.

**Theorem 2** *([19], Theorem 5) (With naive partitioning and representation of solid paths as locally biased binary trees), Any sequence of $m$ dynamic tree operations can be performed in $O(m \log n)$ operations.*

The following theorem implies an $O(\log n)$ worst case time per operation. Using this implementation, we achieve the $O(\log n)$ worst case time bounds for our 2-edge connectivity (section 2 and cycle equivalence (section 3) algorithms. However, as stated in [19], this implementation of dynamic trees is more complicated than the previous one and the former one may perform better in practice (in the amortized sense). So if amortized time bounds are sufficient, we can use the previous implementation.

**Theorem 3** *([19], Theorem 8) (With partitioning by size and a representation of solid paths as globally biased binary trees), Any dynamic tree operation takes $O(\log n)$ time in the worst case.*

### Extension to Support the Global Minimum Operation

Recall that the original dynamic tree data structure allows the path minimum operation only. The global minimum operation asks for an edge $e$ such that $cost(e) = min_{e' \in F} cost(e')$. To support this query we use some additional data structures. Specifically, we use a heap that allows us to insert an item, delete an item and find the minimum valued item among all the items in the heap. Implementations of the heap that take $O(\log r)$ worst-case time per operation, are well known. Here, $r$ is the number of items currently in the heap.

The items in the heap will be the solid paths and the light edges of the dynamic tree data structure. The value of a light edge will be its cost while the value of a solid path will be the minimum cost on that path. Whenever the solid paths or light edges or their costs change, we modify the heap. As only $O(\log n)$ paths change during a single dynamic tree operation, the time taken for maintaining this augmented dynamic tree structure is $O(\log^2 n)$ per operation. The queries can be answered in $O(\log n)$ time. Note that we cannot afford to store all the edges in the heap because a single *add_cost_path* operation on the dynamic tree could change $\Omega(n)$ edge costs simultaneously, and this would require $\Omega(n)$ heap operations to update all the edge costs.

Thus, by using this heap along with the dynamic tree data structure, we can also support the global minimum operation in $O(\log^2 n)$ worst-case time per operation.