Deterministic Worst Case Dynamic Connectivity: Simpler and Faster

Casper Kejlberg-Rasmussen† Tsvi Kopelowitz‡ Seth Pettie§ Mikkel Thorup¶

Abstract

We present a deterministic dynamic connectivity data structure for undirected graphs with worst-case update time \(O\left(\min\left\{\frac{\sqrt{n}}{w^{1/4}}, \sqrt{\frac{n \log n \log(\frac{n}{w})}{w}}\right\}\right)\) and constant query time, where \(w = \Omega(\log n)\) is the word size. This bound is always at most \(O(\sqrt{n} / \log^{1/4} n)\). It improves on the previous best deterministic worst-case algorithm of Frederickson (STOC, 1983) and Eppstein Galil, Italiano, and Nissenzweig (J. ACM, 1997), which had update time \(O(\sqrt{n})\). All known faster dynamic connectivity algorithms are either randomized, or have amortized updates, or both. The simplest of our algorithms uses only commodity data structures (binary search trees, bit vectors) and may be practically competitive.

1 Introduction

The dynamic connectivity problem is to represent an undirected graph \(G = (V, E)\) that is subject to three operations.

\textbf{INSERT}(u, v) : Set \(E \leftarrow E \cup \{(u, v)\}\).

\textbf{DELETE}(u, v) : Set \(E \leftarrow E \setminus \{(u, v)\}\).

\textbf{CONN?}(u, v) : Determine whether \(u\) and \(v\) are in the same connected component in \(G\).

Dynamic connectivity data structures typically maintain a spanning forest of the current graph, which lets them answer additional queries such as counting the number of connected components. In 1983 Frederickson [8] introduced topology trees and 2-dimensional topology trees to solve the dynamic connectivity problem. The update time of Frederickson’s algorithm is \(O(\sqrt{m})\), which was subsequently improved [4] to \(O(\sqrt{m} \log(m/n))\) using a simple, generic sparsification technique and later [3] to \(O(\sqrt{n})\) using more sophisticated sparsification. Except for the improvements due to

---

*This work is supported in part by the Danish National Research Foundation grant DNRF84 through the Center for Massive Data Algorithmics (MADALGO). S. Pettie is supported by NSF grants CCF-1217338 and CNS-1318294 and a grant from the US-Israel Bina
tional Science Foundation. M. Thorup’s research is partly supported by Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

†MADALGO, Aarhus University
‡University of Michigan. Email: kopolot@gmail.com
§University of Michigan. Email: pettie@umich.edu
¶University of Copenhagen. Email: mthorup@di.ku.dk
sparsification [4, 3]. Frederickson’s worst-case data structure has not been improved in over thirty years. However, there have been dramatic improvements in dynamic connectivity data structures that have amortized bounds, or use randomization, or both. See Table 1 for an at-a-glance history of dynamic connectivity.

| Worst Case Data Structures |
|----------------------------|
| Ref. | Update Time | Query Time | Notes |
| [8] | $O(\sqrt{m})$ | $O(1)$ | |
| [3, 8] | $O(\sqrt{n})$ | $O(1)$ | $[8]$ + sparsification [4]. |
| [14] | $O(c \log^5 n)$ | $O(\log n / \log \log n)$ | Randomized Monte Carlo; no connectivity witness; $n^c$ ops. err with prob. $n^{-c}$. |
| new | $O(\sqrt{n/w^{1/4}})$ | $O(1)$ | $w = \Omega(\log n)$ |
| | $O\left(\sqrt{n \log n \log\left(\frac{w}{\log n}\right)/w}\right)$ | | |

| Amortized Data Structures |
|----------------------------|
| Ref. | Amort. Update | W.C. Query | Notes |
| [10] | $O(\log^3 n)$ | $O(\log n / \log \log n)$ | Randomized Las Vegas. |
| [12] | $O(\log^2 n)$ | $O(\log n / \log \log n)$ | Randomized Las Vegas. |
| [13] | $O(\log^2 n)$ | $O(\log n / \log \log n)$ | |
| [19] | $O(\log n(\log \log n)^3)$ | $O(\log n / \log \log n)$ | Randomized Las Vegas. |
| [20] | $O(\log^2 n / \log \log n)$ | $O(\log n / \log \log n)$ | |

| Amort./Worst Case Lower Bounds |
|-------------------------------|
| Ref. | Update Time $t_u$ | Query Time $t_q$ | Notes |
| [9, 11, 15] | | $t_q = \Omega(\log n / \log(t_u \log n))$ | |
| [16] | $t_u = \Omega(\log n / \log(t_q / t_u))$ | $t_q = \Omega(\log n / \log(t_u / t_q))$ | Implies $\max\{t_u, t_q\} = \Omega(\log n)$. |
| [17] | $o(\log n)$ | implies $\Omega(n^{1-o(1)})$ | |

Table 1: A survey of dynamic connectivity results. The lower bounds hold in the cell probe model with word size $w = \Theta(\log n)$.

1.1 New Results

We give a new deterministic dynamic connectivity data structures with worst-case update time on the order of

$$\min\left\{ \sqrt{m / w^{1/4}}, \sqrt{m \log n \log\left(\frac{w}{\log n}\right)} \right\},$$
where $w = \Omega(\log n)$ is the word size. These are the first improvements to Frederickson’s 2D-topology trees \cite{Frederickson82} in over 30 years. Using the sparsification reduction of Eppstein, Galil, Italiano and Nissenzweig \cite{EppsteinGalilItalianoNissenzweig}, the running time expressions can be made to depend on ‘$n$’ rather than ‘$m$’, so we obtain $O(\sqrt{n}/\log^{1/4} n)$ bounds (or faster) for all $w$ and all graph densities.

1.2 Amortized and/or Randomized Dynamic Connectivity Structures

Holm, de Lichtenberg, and Thorup \cite{HolmLichtenbergThorup} presented a relatively simple deterministic structure with amortized update time $O(\log^2 n)$ and query time $O(\log n / \log \log n)$\footnote{In general, any data structure that maintains a spanning forest can have query time $t_q = O(\log_{\log w} n)$ if the update time is $t_u = \Omega(\log n)$. See \cite{SleatorTarjan}.}. By introducing (Las Vegas) randomization, Thorup \cite{Thorup04} reduced the amortized update time to $O(\log n (\log \log n)^3)$. Wulff-Nilsen \cite{Wulff-Nilsen10} recently improved the deterministic amortized update time of \cite{HolmLichtenbergThorup} to $O(\log^2 n / \log \log n)$.

In a major breakthrough, Kapron, King, and Mountjoy \cite{KapronKingMountjoy} discovered a randomized algorithm for dynamic connectivity with worst case polylogarithmic time per operation. Unlike prior Las Vegas randomized algorithms \cite{HenzingerKing02,SleatorTarjan85,SleatorTarjan90}, the algorithm of \cite{KapronKingMountjoy} can have false negatives (reporting that two vertices are disconnected when they are, in fact, connected). Moreover, the Kapron et al. algorithm does not maintain a public certificate of connectivity, i.e., a spanning forest.\footnote{We use the standard repertoire of $AC^0$ operations: left and right shifts, bitwise operations on words, additions and comparisons. They do not assume unit-time multiplication.}

With update time $O(c \log^5 n)$, a sequence of $n^c$ operations is handled correctly with probability $1 - n^{-c}$. Some special graph classes can be handled more efficiently. For example, Sleator and Tarjan \cite{SleatorTarjan83} maintain a dynamic set of trees in $O(\log n)$ worst-case update time subject to $O(\log n)$ time connectivity queries. (See also \cite{HenzingerKing02,SleatorTarjan85,SleatorTarjan90}.) Connectivity in dynamic planar graphs can be reduced to the dynamic tree problem \cite{Frederickson83,Frederickson86}, and therefore solved in $O(\log n)$ time per operation. The cell probe lower bounds of Pătraşcu and Demaine \cite{PtrascuDemaine} show that Sleator and Tarjan’s bounds are optimal in the sense that some operation must take $\Omega(\log n)$ time. Sublogarithmic queries can be obtained by modestly increasing the update time, but Pătraşcu and Thorup \cite{PtraScuThorup10} prove the reverse is not possible. In particular, any dynamic connectivity algorithm with $o(\log n)$ update time has $n^{1−o(1)}$ query time. Refer to Table 1 for a summary of cell probe lower bounds.

Organization. In Sections \ref{section2}–\ref{section5} we describe the $O(\sqrt{n}/w^{1/4})$-time dynamic connectivity structure. It includes data structures that, if used by themselves, give an $O(\sqrt{n \log n \log(w/\log n)/w})$-time dynamic connectivity structure. The modifications needed to obtain this second algorithm are described in Section \ref{section6}.

2 A New Dynamic Connectivity Structure

2.1 Preliminaries and Overview

The algorithm maintains a spanning tree of each component of the graph as a witness of connectivity. Each such witness tree $T$ is represented as an Euler tour $\text{Euler}(T)$\footnote{Henzinger and King \cite{HenzingerKing02} were the first to use Euler tours to represent dynamic trees. G. Italiano (personal communication) observed that Euler tours could be used in lieu of Frederickson’s topology trees to obtain an $O(\sqrt{m})$-time dynamic connectivity structure.}. $\text{Euler}(T)$ is the sequence of
vertices encountered in some Euler tour around \( T \), as if each undirected edge were replaced by two oriented edges. It has length precisely \( 2(|V(T)| - 1) \) (the number of oriented edges) if \( |V(T)| \geq 2 \) or length 1 if \( |V(T)| = 1 \). Vertices may appear in Euler\((T)\) several times. We designate one copy of each vertex the principle copy, which is responsible for all edges incident to the vertex. Each vertex in the graph maintains a pointer to its principle copy. Each \( T \)-edge \((u, v)\) maintains two pointers to the (possibly non-principle) copies of \( u \) and \( v \) that precede the oriented occurrences of \((u, v)\) and \((v, u)\) in Euler\((T)\), respectively. Note that cyclic rotations of Euler\((T)\) are also valid Euler tours; if Euler\((T) = (u, \ldots, v)\) the last element of the list is associated with the tree edge \((v, u)\).

When an edge \((u, v)\) that connects distinct witness trees \( T_0 \) and \( T_1 \) is inserted, \((u, v)\) becomes a tree edge and we need to construct Euler\((T_0 \cup \{(u, v)\} \cup T_1)\) from Euler\((T_0)\) and Euler\((T_1)\). In the reverse situation, if a tree edge \((u, v)\) is deleted from \( T = T_0 \cup \{(u, v)\} \cup T_1 \) we first construct Euler\((T_0)\), Euler\((T_1)\) from Euler\((T)\) then find a replacement edge, \((\hat{u}, \hat{v})\) with \( \hat{u} \in V(T_0) \) and \( \hat{v} \in V(T_1) \), and construct Euler\((T_0 \cup \{\hat{u}, \hat{v}\} \cup T_1)\) from Euler\((T_0)\), Euler\((T_1)\). Lemma 2.1 establishes the nearly obvious fact that the new Euler tours can be obtained from the old Euler tours using \( O(1) \) of the following surgical operations: splitting and concatenating lists of vertices, and creating and destroying singleton lists containing non-principle copies of vertices.

**Lemma 2.1.** If \( T = T_0 \cup \{(u, v)\} \cup T_1 \) and \((u, v)\) is deleted, Euler\((T_0)\) and Euler\((T_1)\) can be constructed from Euler\((T)\) with \( O(1) \) surgical operations. In the opposite direction, from Euler\((T_0)\) and Euler\((T_1)\) we can construct Euler\((T_0 \cup \{(u, v)\} \cup T_1)\) with \( O(1) \) surgical operations. It takes \( O(1) \) time to determine which surgical operations to perform.

**Proof.** Suppose without loss of generality that Euler\((T) = (P_0, u, v, P_1, v, u, P_2)\) where \( P_0, P_1, \) and \( P_2 \) are sequences of vertices. (Note that Euler tours never contain immediate repetitions. If \( P_1 \) is empty then Euler\((T)\) would be just \( (P_0, u, v, u, P_2) \); if both \( P_0 \) and \( P_2 \) are empty then Euler\((T) = (u, v, P_1, v)\).) Then we obtain Euler\((T_0) = (P_0, u, P_2)\) and Euler\((T_1) = (v, P_1)\) with \( O(1) \) surgical operations, which includes the destruction of non-principle copies of \( u \) and \( v \); at least one of the two copies must be non-principle. Since cyclic rotations of Euler tours are valid Euler tours, we could also set Euler\((T_1) = (P_1, v)\), which would be more economical if the \( v \) following \( P_1 \) in Euler\((T)\) were the principle copy.

In the reverse direction, write Euler\((T_0) = (P_0, u, P_1)\) and Euler\((T_1) = (P_2, v, P_3)\), where the labeled occurrences are the principle copies of \( u \) and \( v \). Then Euler\((T_0 \cup \{(u, v)\} \cup T_1) = (P_0, u, v, P_3, P_2, v, u, P_1)\), where the new copies of \( u \) and \( v \) are clearly non-principle copies. If \( P_2 \) and \( P_3 \) were empty (or \( P_0 \) and \( P_1 \) were empty) then we would not need to add a non-principle copy of \( v \) (or a non-principle copy of \( u \)).

Define \( \hat{m} \) to be an upper bound on \( m \), the number of edges. The update time of our data structure will be a function of \( \hat{m} \). The sparsification method of [3] creates instances in which \( \hat{m} \) is known to be linear in the number of vertices. Nonetheless, using standard rebuilding techniques it is possible to obtain a direct algorithm (without using sparsification) whose worst case running time depends on the current \( m \). The idea is to progressively rebuild the data structure whenever \( m \) gets within \( \sqrt{\hat{m}} \) of \( \hat{m} \) or \( \hat{m}/4 \), doubling or halving \( \hat{m} \) as appropriate.

### 2.2 A Dynamic List Data Structure

We have reduced dynamic connectivity in graphs to implementing several simple operations on dynamic lists. We will maintain a pair \((\mathcal{L}, E)\), where \( \mathcal{L} \) is a set of lists (containing principle and
non-principle copies of vertices) and $E$ is the dynamic set of edges joining principle copies of vertices. In addition to the creation and destruction of single element lists we must support the following primitive operations.

**LIST**(x) : Return the list in $\mathcal{L}$ containing element $x$.

**JOIN**(L₀, L₁) : Set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L₀, L₁\} \cup \{L₀L₁\}$, that is, replace $L₀$ and $L₁$ with their concatenation $L₀L₁$.

**SPLIT**(x) : Let $L = L₀L₁ \in \mathcal{L}$, where $x$ is the last element of $L₀$. Set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L\} \cup \{L₀, L₁\}$.

**REPLACEMENTEDGE**(L₀, L₁) : Return any edge joining elements in $L₀$ and $L₁$.

Our implementations of these operations will only be efficient if, after each INSERT or DELETE operation, there are no edges connecting distinct lists. That is, the REPLACEMENTEDGE operation is only employed by DELETE when deleting a tree edge in order to restore Invariant 2.2.

**Invariant 2.2.** Each list $\mathcal{L}$ corresponds to the Euler tour of a spanning tree of some connected component.

The dynamic connectivity operations are implemented as follows. To answer a **CONN**(u, v) query we simply check whether **LIST**(u) = **LIST**(v). To insert an edge (u, v) we do **INSERT**(u, v), and if **LIST**(u) ≠ **LIST**(v) then make (u, v) a tree edge and perform suitable **SPLIT**s and **JOIN**s to merge the Euler tours **LIST**(u) and **LIST**(v). To delete an edge (u, v) we do **DELETE**(u, v), and if (u, v) is a tree edge in $T = T₀ \cup \{(u, v)\} \cup T₁$, perform suitable **SPLIT**s and **JOIN**s to create Euler($T₀$) and Euler($T₁$) from Euler($T$). At this point Invariant 2.2 may be violated as there could be an edge joining $T₀$ and $T₁$. We call **REPLACEMENTEDGE**(Euler($T₀$), Euler($T₁$)) and if it finds an edge, say (ǔ, ń), we perform more **SPLIT**s and **JOIN**s to form Euler($T₀ \cup \{ǔ, ń\} \cup T₁$).

Henzinger and King [10] observed that most off-the-shelf balanced binary search trees can support **SPLIT**, **JOIN**, and other operations in logarithmic time. However, they provide no direct support for the **REPLACEMENTEDGE** operation, which is critical for the dynamic connectivity application.

### 2.3 Chunks and Superchunks

Recall that edges are only incident to principle copies of vertices. If $L'$ is a sublist of a list $L \in \mathcal{L}$, define mass($L'$) to be the number of edges incident to elements of $L'$, counting an edge twice if both endpoints are in $L'$. The sum of list masses, $\sum_{L \in \mathcal{L}}$ mass($L$), is clearly at most $2\hat{m}$, where $\hat{m}$ is a fixed upper bound on the number of edges. We maintain a partition of each list $L \in \mathcal{L}$ into chunks, each of whose mass is ideally close to a parameter $K$ that depends on $\hat{m}$, but may deviate from this ideal (below or above) out of necessity. (In this algorithm we set $K = \sqrt{m}/w^{1/4}$. It is set differently in the algorithm presented in Section 6.)

**Invariant 2.3.** Each $L \in \mathcal{L}$ is partitioned into chunks $L = C₀C₁ \cdots C_{p-1}$ such that for all $l \in [p]$, exactly one of the following mass criteria is satisfied.

**Criterion 1.** $K \leq$ mass($C_l$) ≤ 4$K$ and all elements of $C_l$ have mass at most 2$K$.

**Criterion 2.** mass($C_l$) > 2$K$ and $C_l$ consists of a single element, or
Lemma 2.4. Let \( \mathcal{L} \) be a set of lists whose total mass is at most \( 2m \). Any partition of its lists into chunks that conforms to Invariant 2.3 uses at most \( 2m/K + |\mathcal{L}| \) chunks. Excluding lists with mass less than \( K \) (each consisting of a single Criterion 3 chunk), there are \( O(m/K) \) chunks.

Proof. Pair each Criterion 3 chunk with the Criterion 2 chunk to its right, if any. Each chunk-pair has mass at least \( 2K \) and each unpaired chunk has mass at least \( K \), except for possibly the last chunk of each list.

The chunks are partitioned into contiguous sequences of superchunks according to Invariant 2.5. Without loss of generality suppose \( \sqrt{w} \) is an integer, where \( w \) is the word size.

Invariant 2.5. A list in \( \mathcal{L} \) having fewer than \( \sqrt{w}/2 \) chunks forms a single superchunk with ID \( \bot \). A list in \( \mathcal{L} \) with at least \( \sqrt{w}/2 \) chunks is partitioned into superchunks, each consisting of between \( \sqrt{w}/2 \) and \( \sqrt{w} - 1 \) consecutive chunks. Each such superchunk has a unique ID in \( [J] \), where \( J = O(m/(K\sqrt{w})) \). (IDs are completely arbitrary. They do not encode any information about the order of superchunks within a list.)

Call an Euler tour list short if it consists of fewer than \( \sqrt{w}/2 \) chunks. We shall assume that no lists are ever short, as this simplifies the description of the data structure and its analysis. In particular, all superchunks have proper IDs in \( [J] \). In Section 5.1 we sketch the uninteresting complications introduced by \( \bot \) IDs and short lists.

2.4. Word Operations

Invariant 2.5 implies that we can store a matrix \( A \in \{0,1\}^{\sqrt{w} \times \sqrt{w}} \) in one word that represents the adjacency between the chunks in two superchunks \( i \) and \( j \), where the rows and columns are identified with chunks in \( i \) and \( j \), respectively. This matrix will always be represented in row-major order; rows and columns are indexed by \( \sqrt{w} = \{0, \ldots, \sqrt{w} - 1\} \). In this format it is straightforward to insert a new all-zero row above a specified row \( k \) (and destroy row \( \sqrt{w} - 1 \) by shifting the old rows \( k, \ldots, \sqrt{w} - 2 \) down by one). It is also easy to copy an interval of rows from one matrix to another. Lemma 2.6 shows that the corresponding operations on columns can also be effected in \( O(1) \) time with a mask \( \mu \) precomputable in \( O(\log w) \) time.

Lemma 2.6. Assume w.l.o.g. that \( \sqrt{w} \) is an even integer and let \( \mu \) be the word \( \{1\sqrt{w}0\sqrt{w}\}^{\sqrt{w}/2} \). Given \( \mu \) we can in \( O(1) \) time copy/paste any interval of columns from/to a matrix \( A \in \{0,1\}^{\sqrt{w} \times \sqrt{w}} \), represented in row-major order.

Proof. Recall that the rows and columns are indexed by integers in \( \sqrt{w} = \{0, \ldots, \sqrt{w} - 1\} \). We first describe how to build a mask \( \nu_k \) for columns \( k, \ldots, \sqrt{w} - 1 \) then illustrate how it is used to copy/paste intervals of columns. In C notation\(^5\) the word \( \nu_k' = (\mu \gg k) \& \mu \) is a mask for the intersection of the even rows and columns \( k, \ldots, \sqrt{w} - 1 \), so \( \nu_k = \nu_k' \land (\nu_k' \gg \sqrt{w}) \) is a mask for columns \( k \) through \( \sqrt{w} - 1 \).

To insert an all-zero column before column \( k \) of \( A \) (and delete column \( \sqrt{w} - 1 \) we first copy columns \( k, \ldots, \sqrt{w} - 2 \) to \( A' = A \& (\nu_{k+1} \ll 1) \) then set \( A = (A \& (\neg \nu_k)) \lor (A' \gg 1) \). Other operations can be effected in \( O(1) \) time with copying/pasting intervals of columns, e.g., splitting an array into two about a designated column, or merging two arrays having at most \( \sqrt{w} \) columns together.

\(^5\) The operations \&, \|, and \~\ are bit-wise AND, OR, and NOT; \ll and \gg are left and right shift.


3 Adjacency Data Structures

In order to facilitate the efficient implementation of REPLACEMENT EDGE we maintain an $O(m/K) \times O(m/K)$ adjacency matrix between chunks, and a $J \times J$ adjacency matrix between superchunks. However, in order to allow for efficient dynamic updates it is important that these matrices be represented in a non-standard format described below. The data structure maintains the following information.

- Each list element maintains a pointer to the chunk containing it. Each chunk maintains a pointer to the superchunk containing it, as well as an index in $[\sqrt{w}]$ indicating its position within the superchunk. Each superchunk maintains its ID in $[J] \cup \{\perp\}$ and a pointer to the list containing it.

- ChAdj is a $J \times J$ array of $w$-bit words, indexed by superchunk IDs. The entry ChAdj($i, j$) is interpreted as a $\sqrt{w} \times \sqrt{w}$ 0-1 matrix that keeps the adjacency information between all pairs of chunks in superchunk $i$ and superchunk $j$. (It may be that $i = j$.) In particular, ChAdj($i, j$)($k, l$) = 1 iff there is an edge with endpoints in the $k$th chunk of superchunk $i$ and the $l$th chunk of superchunk $j$, so ChAdj($i, j$) = 0 (i.e., the all-zero matrix) if no edge joins superchunks $i$ and $j$. The matrix ChAdj($i, j$) is stored in row-major order.

- Let $S$ be a superchunk with $ID(S) = \perp$. By Invariants 2.2 and 2.3 $S$ is not incident to any other superchunks and has fewer than $\sqrt{w}/2$ chunks. We maintain a single word ChAdj$_S$ which stores the adjacency matrix of the chunks within $S$.

- For each superchunk with ID $i \in [J]$ we keep length-$J$ bit-vectors SupAdj$_i$ and Memb$_i$, where
  
  \begin{align*}
  \text{SupAdj}_i(j) &= 1 \text{ if ChAdj}(i, j) \neq 0 \text{ and 0 otherwise, whereas} \\
  \text{Memb}_i(j) &= 1 \text{ if } j = i \text{ and 0 otherwise.}
  \end{align*}

  These vectors are packed into $\lceil J/w \rceil$ machine words, so scanning one takes $O(\lceil J/w \rceil)$ time.

- We maintain an list-sum data structure that allows us to take the bit-wise OR of the SupAdj$_i$ vectors or Memb$_i$ vectors, over all superchunks in an Euler tour. It is responsible for maintaining the \{SupAdj$_i$, Memb$_i$\} vectors described above and supports the following operations. At all times the superchunks are partitioned into a set $S$ of disjoint lists of superchunks. Each $S \in S$ (a list of superchunks) is associated with an $L \in \mathcal{L}$ (an Euler tour), though short lists in $\mathcal{L}$ have no need for a corresponding list in $S$.

  SCInsert($i$) : Retrieve an unused ID, say $i'$, and allocate a new superchunk with ID $i'$ and all-zero vector SupAdj$_{i'}$. Insert superchunk $i'$ immediately after superchunk $i$ in $i$'s list in $S$. If no $i$ is given, create a new list in $S$ consisting of superchunk $i'$.

  SCDelete($i$) : Delete superchunk $i$ from its list and make ID $i$ unused.

  SCJoin($S_0, S_1$) : Replace superchunk lists $S_0, S_1 \in S$ with their concatenation $S_0S_1$.

  SCSplit($i$) : Let $S = S_0S_1 \in S$ and $i$ be the last superchunk in $S_0$. Replace $S_0S_1$ with two lists $S_0, S_1$.

  UPDATEAdj($i, x \in \{0, 1\}^J$) : Set SupAdj$_i \leftarrow x$ and update SupAdj$_j(i) \leftarrow x(j)$ for all $j \neq i$. 


ADJQUERY(S) : Return the vector $\alpha \in \{0, 1\}^J$ where

$$\alpha(j) = \bigvee_{i \in S} \text{SupAdj}_i(j)$$

The index $i$ ranges over the IDs of all superchunks in $S$.

MEMBQUERY(S) : Return the vector $\beta \in \{0, 1\}^J$, where

$$\beta(j) = \bigvee_{i \in S} \text{Memb}_i(j)$$

We use the following implementation of the list-sum data structure. Each list of superchunks is maintained as any $O(1)$-degree search tree that supports logarithmic time inserts, deletes, splits, and joins. Each leaf is a superchunk that stores its two bit-vectors. Each internal node $z$ keeps two bit-vectors, SupAdj$^z$ and Memb$^z$, which are the bit-wise OR of their leaf descendants’ respective bit-vectors. Because length-$J$ bit-vectors can be updated in $O([J/w])$ time, all “logarithmic time” operations on the tree actually take $O(\log J \cdot J/w)$ time. The UPDATEADJ($i, x$) operation takes $O(\log J \cdot J/w)$ time to update superchunk $i$ and its $O(\log J)$ ancestors. We then need to update the $i$th bit of potentially every other node in the tree, in $O(J)$ time. Since $w = \Omega(\log n) = \Omega(\log J)$ the cost per UPDATEADJ is $O(J)$. The answer to an ADJQUERY(S) or MEMBQUERY(S) is stored at the root of the tree on $S$.

4 Creating and Destroying (Super)Chunks

There are essentially two causes for the creation and destruction of (super)chunks. The first is in response to a SPLIT operation that forces a (super)chunk to be broken up. (The SPLIT may itself be instigated by the insertion or deletion of an edge.) The second is to restore Invariants 2.3 and 2.5 after a JOIN or INSERT or DELETE operation. In this section we consider the problem of updating the adjacency data structures after four types of operations: (i) splitting a chunk in two, keeping both chunks in the same superchunk, (ii) merging two adjacent chunks in the same superchunk, (iii) splitting a superchunk along a chunk boundary, and (iv) merging adjacent superchunks. Once we have bounds on (i)–(iv), implementing the higher-level operations in the stated bounds is relatively straightforward. Note that (i)–(iv) may temporarily violate Invariants 2.3 and 2.5.

Splitting Chunks Suppose we want to split the $k$th chunk of superchunk $i$ into two pieces, both of which will (at least temporarily) stay within superchunk $i$. We first zero-out all bits of ChAdj($i, \star$)(*$k$, $\star$) and ChAdj($\star$, $i$)(*$k$, $\star$) in $O(J)$ time. For each $j$ we need to insert an all-zero row below row $k$ in ChAdj($i, j$) and an all-zero column after column $k$ of ChAdj($j, i$). This can be done in $O(1)$ time for each $j$, or $O(J)$ in total; see Lemma 2.6.

Since the $k$th chunk was originally not a Criterion 2 chunk, in $O(K)$ time we scan the edges incident to the new chunks $k$ and $k + 1$ and update the corresponding bits in ChAdj($i, \star$)(*$k'$, $\star$) and ChAdj($\star$, $i$)(*$k'$, $\star$), for $k' \in \{k, k + 1\}$.

---

6Remember that ‘$k$’ refers to the actual position of the chunk within its superchunk whereas ‘$i$’ is an arbitrary ID that does not relate to its position within the list.
Merging Adjacent Chunks  In order to merge chunks $k$ and $k + 1$ of superchunk $i$ we need to replace row $k$ of ChAdj$(i, j)$, for all $j$, with the bit-wise OR of rows $k$ and $k + 1$ of ChAdj$(i, j)$, zero out row $k + 1$, then scoot rows $k + 2, \cdots$ back one row. A similar transformation is performed on columns $k$ and $k + 1$ of ChAdj$(j, i)$, which takes $O(1)$ time per $j$, by Lemma 2.6. In total the time is $O(J)$, independent of $K$.

Splitting Superchunks  Suppose we want to split superchunk $i$ after its $k$th chunk. We first call SCINSERT$(i)$, which allocates an empty superchunk with ID $i'$ and inserts $i'$ after $i$ in its superchunk list in $S$. In $O(J)$ time we transfer rows $k + 1, \ldots, \sqrt{w} - 1$ from ChAdj$(i, j)$ to ChAdj$(i', j)$ and transfer columns $k + 1, \ldots, \sqrt{w} - 1$ from ChAdj$(j, i)$ to ChAdj$(j, i')$. By Lemma 2.6 this takes $O(1)$ time per $j$.

At this point ChAdj is up-to-date but the list-sum data structure and {SupAdj$_j$} bit-vectors are not. We update SupAdj$_j$, SupAdj$_{ij}$ with calls to UPDATEADJ$(i, x)$ and UPDATEADJ$(i', x')$. Using ChAdj, each bit of $x$ and $x'$ can be generated in constant time. This takes $O(J)$ time.

Merging Superchunks  Let the two adjacent superchunks have IDs $i$ and $i'$. It is guaranteed that they will be merged only if they contain at most $\sqrt{w}$ chunks together. In $O(J)$ time we transfer the non-zero rows of ChAdj$(i', j)$ to ChAdj$(i, j)$ and transfer the non-zero columns of ChAdj$(j, i')$ to ChAdj$(j, i)$. A call to SCDELETE$(i')$ deletes superchunk $i'$ from its list in $S$ and retires ID $i'$. We then call UPDATEADJ$(i, x)$ with the new incidence vector $x$. In this case we can generate $x$ in $O(J/w)$ time since it is merely the bit-wise OR of the old vectors SupAdj$_j$ and SupAdj$_{ij}$, with bit $i'$ set to zero. Updating the list-sum data structure takes $O(J)$ time.

5 Joining and Splitting Lists

Once we have routines for splitting and merging adjacent (super)chunks, implementing JOIN and SPLIT on lists in $L$ is much easier. The goal is to restore Invariant 2.3 governing chunk masses and Invariant 2.5 on the number of chunks per superchunk.

Performing JOIN$(L_0, L_1)$. Write $L_0 = C_0, \ldots, C_{p-1}$ and $L_1 = D_0, \ldots, D_{q-1}$ as a list of chunks and let $S_0, S_1 \in S$ be the superchunk lists corresponding to $L_0, L_1 \in L$. We call SCJOIN$(S_0, S_1)$ to join $S_0, S_1$ in $S$, in $O(J)$ time. If both $C_{p-1}$ and $D_0$ are Criterion 1 or 2 chunks, or if one satisfies Criterion 2 while the other satisfies Criterion 3, then $L = L_0 L_1$ satisfies Invariant 2.3. If $C_{p-1}$ satisfies Criterion 1 while $D_0$ satisfies Criterion 3 then we will ultimately need to merge them and, if mass$(C_{p-1} D_0) > 4K$, respilt $C_{p-1} D_0$ along a different boundary to form two Criterion 1 chunks. Suppose w.l.o.g. that this boundary is in $D_0$, we first split $D_0$ into $D'_0 D''_0$ and split the superchunk containing $D_0$ along this same boundary, then merge the superchunk consisting of $D'_0$ with the superchunk containing $C_{p-1}$, then merge chunks $C_{p-1}, D'_0$ into a single chunk. The number of chunks in any superchunk has not increased, so Invariant 2.5 remains satisfied.

Performing SPLIT$(x)$. Suppose $x$ is contained in chunk $C_i$ of $L = C_0 \cdots C_{i-1} C_i C_{i+1} \cdots C_{p-1}$. We split $C_i$ into two chunks $C'_i C''_i$, and split the superchunk containing $C_i$ along this line. Let $S$ be the superchunk list corresponding to $L$ and $i$ be the ID of the superchunk ending at $C'_i$. We split $S$ using a call to SCSPLIT$(i)$, which corresponds to splitting $L$ into $L_0 = C_0 \cdots C_{i-1} C'_i$ and $L_1 = C'_i C_{i+1} \cdots C_{p-1}$. At this point $C'_i$ or $C''_i$ may violate Invariant 2.3 if mass$(C'_i) < K$ or
mass($C''_i) < K$ and $C_{l-1}$ or $C_{l+1}$ is not a Criterion 2 chunk. Furthermore, Invariant 2.5 may be violated if the number of chunks in the superchunks containing $C'_i$ and $C''_i$ is too small. We first correct Invariant 2.5 by possibly merging and resplitting $C_{l-1}C'_i$ and $C''_iC_{l+1}$ along new boundaries. If the superchunk containing $C'_i$ has fewer than $\sqrt{w}/2$ chunks, it and the superchunk to its left have strictly between $\sqrt{w}/2$ and $3\sqrt{w}/2$ chunks together, and so can be merged (and possibly resplit) into one or two superchunks satisfying Invariant 2.5. The same method can correct a violation of $C''_i$'s superchunk. This takes $O(K + J)$ time.

Performing ReplacementEdge($L_0, L_1$) The list-sum data structure makes implementing the ReplacementEdge($L_0, L_1$) operation easy. Let $S_0$ and $S_1$ be the superchunk lists corresponding to Euler tours $L_0$ and $L_1$. We compute the vectors $\alpha \leftarrow \text{ADQUERY}(S_0)$ and $\beta \leftarrow \text{MEMBQUERY}(S_1)$ and their bit-wise AND $\alpha \land \beta$ with a linear scan of both vectors. If $\alpha \land \beta$ is the all-zero vector then there is no edge between $L_0$ and $L_1$. On the other hand, if $(\alpha \land \beta)(j) = 1$, then $j$ must be the ID of a superchunk in $S_1$ that is incident to some superchunk in $S_0$. To determine which superchunk in $S_0$ we walk down from the root of $S_0$'s list-sum tree to a leaf, say with ID $i$, in each step moving to a child $z$ of the current node for which SupAdj$^z(i, j) = 1$. Once $i$ and $j$ are known we retrieve any 1-bit in the matrix ChAdj$^z(i, j)$, say at position $(k, l)$, indicating that the $k$th chunk of superchunk $i$ and the $l$th chunk of superchunk $j$ are adjacent. If these are Criterion 2 chunks then they consist of single vertices connected by an edge. If either is a Criterion 1 or 3 chunk then we can scan all its adjacent edges in $O(K)$ time and retrieve an edge joining $L_0$ and $L_1$. The total time is $O(J/w + \log J + K) = O(J/w + K)$.

Performing Insert($u, v$) If List($u$) $\neq$ List($v$), first perform $O(1)$ Splits and Joins to restore the Euler tour Invariant 2.2. Now $u$ and $v$ are in the same list in $L$. Let $i, j$ be the IDs of the superchunks containing the principle copies of $u$ and $v$ and let $k, l$ be the positions of $u$ and $v$'s chunks within their respective superchunks. We set ChAdj$(i, j)(k, l) \leftarrow 1$. If ChAdj$(i, j)$ was formerly the all-zero matrix, we call UpdateAdj$(i, x)$ to update superchunk $i$'s adjacency information with the correct vector $x$. Inserting one edge changes the mass of the chunks containing $u$ and $v$, which could violate Invariant 2.3. See the paragraph below on restoring Invariants 2.3 and 2.5.

Performing Delete($u, v$) Compute $i, j, k, l$ as defined above, in $O(1)$ time. After we delete $(u, v)$ the correct value of the bit ChAdj$(i, j)(k, l)$ is uncertain. If the chunks of $u$ and $v$ satisfy Criterion 2 then they each consist of a single element (namely $u$ and $v$) so we can immediately set ChAdj$(i, j)(k, l) \leftarrow 0$. If either is a Criterion 1 or Criterion 3 chunk, say $u$'s chunk, then it has mass $O(K)$. We scan all its incident edges in $O(K)$ time looking for one connected to chunk $l$ of superchunk $j$. If we do not find such an edge we set ChAdj$(i, j)(k, l) \leftarrow 0$, and if that makes ChAdj$(i, j) = 0$ (the all-zero matrix), we call UpdateAdj$(i, x)$, where $x$ is the new adjacency vector of superchunk $i$; it only differs from the former SupAdj$^z$ at position $j$.

If $(u, v)$ is a tree edge in $T = T_0 \cup \{(u, v)\} \cup T_1$ we perform Splits and Joins to replace Euler($T$) with Euler($T_0$), Euler($T_1$), which may violate Invariant 2.2 if there is a replacement edge between $T_0$ and $T_1$. We call ReplacementEdge(Euler($T_0$), Euler($T_1$)) to find a replacement edge. If one is found, say $(\hat{u}, \hat{v})$, we form Euler($T_0 \cup \{(\hat{u}, \hat{v})\} \cup T_1$) with a constant number of Splits and Joins.\footnote{Since $x$ only differs from the former SupAdj$^z$ at position SupAdj$^z(j)$, this update to the list-sum tree can implemented in just $O(\log J)$ time since it only affects ancestors of leaves $i$ and $j.$}
Restoring Invariants 2.3 and 2.5. Even when an insertion/deletion requires no splits or joins, a single operation can violate the mass criteria of Invariant 2.3. After changing the mass of one list element, the mass invariants can be restored by splitting/merging $O(1)$ (super)chunks.

Suppose we increment the mass of an element $u$ in chunk $C_t$, which formerly satisfied Criterion 1 but now does not. There are two cases to consider depending on how Criterion 1 is violated: either $\text{mass}(C_t) > 4K$ or $\text{mass}(u) > 2K$. In the first case, since all elements of $C_t$ have mass at most $2K$, we can replace $C_t$ with two Criterion 1 chunks $C'_t C''_t$ such that $K + 1 \leq \text{mass}(C'_t), \text{mass}(C''_t) \leq 3K$. In the second case ($\text{mass}(u) > 2K$), we temporarily split $C_t$ into three chunks $C'_t C''_t C'''_t$, where $C'''_t = (u)$ now satisfies Criterion 2. If $\text{mass}(C'_t) < K$ and $C_{t-1}$ satisfies Criterion 2 (or if $\text{mass}(C''_t) < K$ and $C_{t+1}$ satisfies Criterion 2) then $C'_t$ (and $C'''_t$) is a valid Criterion 3 chunk. If $C'_t$ or $C''_t$ do not satisfy Criteria 1 or 3 we need to correct this violation. If $\text{mass}(C_{t-1}) + \text{mass}(C'_t) \leq 4K$ then we can join $C_{t-1}, C'_t$ into a single Criterion 1 chunk. Otherwise we split $C_{t-1} C'_t$ as evenly as possible into chunks $C'''_{t-1} C'''_t C'''_t$. Since $\text{mass}(C_{t-1}) + \text{mass}(C'_t) \leq 6K$ and none of their elements has mass more than $2K$, we can ensure that $K \leq \text{mass}(C'_{t-1}), \text{mass}(C'''_t) \leq 4K$, that is, both will satisfy Criterion 1. A violation of Criterion 1 in which the chunk’s mass drops below $K$ is handled similarly.

Violations of Criteria 2 and 3 are simpler to correct. Incrementing or decrementing the mass of a Criterion 3 chunk or incrementing the mass of a Criterion 2 chunk cannot violate the mass invariants. (However, it may be that a Criterion 3 chunk gets promoted to Criterion 1.) Decrementing the mass of a Criterion 2 chunk $C_t$ to $2K$ violates Criterion 2 locally, and may violate $C_{t-1}$ and/or $C_{t+1}$ if they are Criterion 3 chunks. However, the total mass of these violated chunks is between $2K$ and $4K$, so they can be merged into a single Criterion 1 chunk.

Splitting and merging superchunks may violate Invariant 2.5 if the number of chunks per superchunk is less than $\sqrt{w}/2$ or at least $\sqrt{w}$. These can clearly be corrected by splitting/merging $O(1)$ superchunks in the vicinity of the modified chunks.

5.1 Dealing with Short Lists

Until now we have assumed for simplicity that all superchunks have proper IDs in $[J]$. It is important that we not give out IDs to short lists (consisting of less than $\sqrt{w}/2$ chunks) because the running time of the algorithm is linear in the maximum ID $J$. The modifications needed to deal with short lists are tedious but minor.

Consider an insert $(u, v)$ operation where $u$ and $v$ are in lists $L_0, L_1$ and $L_1$ is a short list consisting of one superchunk $S$ with $\text{ID}(S) = 1$. If $L_0$ is not short (or if it is short but the combined list $L_0 L_1$ will not be short) then we retrieve an unused ID, say $i$, set $\text{ID}(S) \leftarrow i$, set $\text{ChAdj}(i, i) \leftarrow \text{ChAdj}_S$, and destroy $\text{ChAdj}_S$. By Invariant 2.2, $S$ was not incident to any other superchunk, so $\text{ChAdj}(i, j) = 0$ (the all-zero matrix) for all $j \neq i$. At this point $S$ violates Invariant 2.3 (it is too small), so we need to merge it with the last superchunk in $L_0$ and resplit it along a different chunk boundary, in $O(J)$ time.

The modifications to delete $(u, v)$ are analogous. If we delete a tree edge $(u, v)$, splitting its component into $T_0$ and $T_1$ with associated Euler tours $L_0$ and $L_1$, and replacement edge $(u, v)$ fails to find an edge joining $L_0$ and $L_1$, we need to check whether $L_0$ (and $L_1$) are short. If so let $S$ be the superchunk in $L_0$. We allocate and set $\text{ChAdj}_S \leftarrow \text{ChAdj}(\text{ID}(S), \text{ID}(S))$, then set $\text{ChAdj}(\text{ID}(S), \text{ID}(S)) \leftarrow 0$ and finally retire $\text{ID}(S)$.

The implementation of replacement edge $(L_0, L_1)$ is different if $L_0$ and $L_1$ were originally in a short list $L = \text{Euler}(T)$ before a tree edge in $T$ was deleted. Suppose $L$ originally had one superchunk $S$, whose chunk adjacency was stored in $\text{ChAdj}_S$. After $O(1)$ splits and joins, both $L_0$’s
chunks and \( L_1 \)'s chunks occupy \( O(1) \) intervals of the rows and columns of \( \text{ChAdj}_S \). Thus, in \( O(1) \) time we can mask out the intersection of \( L_0 \)'s rows and \( L_1 \)'s columns; see Lemma 2.6. If there is any 1 bit here, say at location \( \text{ChAdj}_S(k,l) \), then we know that there is an edge between \( L_0 \) and \( L_1 \), and can find it in \( O(K) \) time by examining chunks \( k \) and \( l \). The permutation of rows/columns in \( \text{ChAdj}_S \) must be updated to reflect any splits and joins that take place, and if no replacement edge is discovered, \( \text{ChAdj}_S \) must be split into two matrices \( \text{ChAdj}_{S_0} \) and \( \text{ChAdj}_{S_1} \), to be identified with the single superchunks \( S_0 \) and \( S_1 \) in \( L_0 \) and \( L_1 \), respectively.

5.2 Running Time Analysis

Each operation ultimately involves splitting/merging \( O(1) \) chunks, superchunks, and lists, which takes time \( O(K + J + \log n \cdot J/w) = O(K + J) = O(K + \hat{m}/(K\sqrt{w})) \). We balance the terms by setting \( K = \sqrt{\frac{\hat{m}}{w}} \) so the running time is \( O(K) \).

By the sparsification transformation of Eppstein, Galil, Italiano, and Nissenzweig [3] this implies an update time of \( O\left(\frac{\sqrt{n}}{w^{3/4}}\right) \). Each instance of dynamic connectivity created by [3] has a fixed set of vertices, say of size \( \hat{n} \), and a fixed upper bound \( \hat{m} = O(\hat{n}) \) on the number of edges.

6 A Faster Algorithm for Large Words

The algorithm we designed achieves a \( w^{1/4} \)-factor speedup over Frederickson’s 2D-topology trees. When the word size is large enough, roughly \( \log^2 n \), we can get an even better speedup of \( \sqrt{w}/\log n \), and with a less-complicated structure to boot. The idea is to unify the roles played by chunks and superchunks: call them superchunks in this structure. Each list is partitioned into superchunks whose masses satisfy Invariant [23] but with a new \( K \) parameter to be determined shortly, where \( K \) is always at most \( O(\sqrt{m}) \). Each superchunk has an ID in the range \( [O(\hat{m}/K)] \), excluding single-superchunk lists with mass less than \( K \), whose IDs are \( \bot \). We explicitly maintain the adjacency between superchunks using the length-\( O(\hat{m}/K) \) bit-vectors \{\( \text{SupAdj}_i \}\}. In addition, the same list-sum data structure is maintained that allows us to compute the bit-wise OR of lists of \{\( \text{SupAdj}_j \)\} and \{\( \text{Memb}_i \)\} vectors. Every operation can be implemented by splitting/merging \( O(1) \) superchunks, so we need to investigate the costs of these operations.

Splitting and Merging Superchunks Suppose superchunk \( i \) has mass \( O(K) \) and we want to split it into two pieces. We first call \( \text{SCINSERT}(i) \), which allocates a new ID, say \( i' \), with vectors \( \text{SupAdj}_{i'} \) and \( \text{Memb}_{i'} \) and inserts superchunk \( i' \) after \( i \) in its list in \( S \). At this point the vectors \( \text{SupAdj}_{i'} \) and \( \text{SupAdj}_i \) are set incorrectly. We compute the correct incidence vectors \( x \) and \( x' \) for superchunks \( i \) and \( i' \) by scanning their \( O(K) \) adjacent edges, then call \( \text{UPDATEADJ}(i,x) \) and \( \text{UPDATEADJ}(i',x') \). There are \( O(K) \) bit positions in \( \text{SupAdj}_i \) and \( \text{SupAdj}_{i'} \) that change value, and therefore \( O(K) \) superchunks \( j \) for which \( \text{SupAdj}_j(i) \) and \( \text{SupAdj}_j(i') \) change value. In the list-sum data structure the changes to \{\( \text{SupAdj}_j \)\} at the leaves are propagated to all ancestors. The cost of updating the vectors at the ancestors of leaves \( i \) and \( i' \) is \( O(\log n \cdot \hat{m}/(Kw)) \). There are \( O(\hat{m}/K) \) leaves and \( O(K) \) of their vectors are modified at bit-positions \( i \) and \( i' \). The number of distinct ancestors of these \( O(K) \) leaves is \( O(K \log(\hat{m}/K^2)) \), so the total cost of splitting a superchunk is \( O(\log n \cdot \hat{m}/(Kw) + K \log(\hat{m}/K^2)) \).
Merging consecutive superchunks $i$ and $i'$ is done in a similar fashion. We need to replace SupAdj$_i$ with the bit-wise OR of SupAdj$_i$ and SupAdj$_{i'}$, so there is no need to actually scan the edges incident to these superchunks. Nonetheless, the cost is still $O(\log n \cdot \hat{m} / (Kw) + K \log(\hat{m} / K^2))$ to merge superchunks.

**Time Analysis** Each **Split** and **Join** operation can be implemented with $O(1)$ superchunk split/merges and a call to **SCSplit** or **SCJoin** to update the list-sum data structure. Each of these operations takes $O(\log n \cdot \hat{m} / (Kw))$ time to update the bit-vectors of the $O(\log n)$ affected nodes.

Each edge insert/delete is implemented with $O(1)$ **Splits** and **Joins** and if a tree-edge is being deleted, a call to **REPLACEMENT**. The **REPLACEMENT**($L_0, L_1$) operation can be performed in $O(\hat{m} / (Kw))$ time by scanning the bit-vectors at the roots of the list-sum trees for $L_0$ and $L_1$, provided that the superchunks of $L_0$ and $L_1$ have proper IDs. If $L_0$ and $L_1$ both consist of a single superchunk with a mass of $O(K)$ and ID equal to $\perp$, a replacement edge can be found in $O(K)$ time by scanning all incident edges.

The time per edge insertion/deletion is therefore $O(\log n \cdot \hat{m} / (Kw) + K \log(\hat{m} / K^2))$. This is balanced when $K = \sqrt{\frac{\hat{m} \log n}{w \log(w / \log n)}}$ making the update time $O(\sqrt{\frac{\hat{m} \log n \log(w / \log n)}{w}})$. The sparsification transformation [3] can then be used to replace ‘$m$’ with ‘$n$’ in these bounds.

One could also reinterpret this algorithm as being in the external memory model rather than the word RAM model. Note that scanning long bit-vectors is I/O-efficient whereas all other operations use random access. Suppose information is transferred between internal and external memory in blocks of $B$ words, each word consisting of $w$ bits, which is essentially the same as a $w' = Bw$-bit word RAM model. Setting $K = \sqrt{\frac{\hat{m} \log n}{w' \log(w' / \log n)}}$, the I/O complexity of our algorithm would then be $O\left(\sqrt{\frac{\hat{m} \log n \log(Bw / \log n)}{Bw}}\right)$, which can also be sparsified [3]. This bound compares favorably with Frederickson and Eppstein et al. [8, 3], which do not benefit from a large block size $B$.

**References**

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 531–540, 2004.

[2] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. on Algorithms*, 1(2):243–264, 2005.

[3] D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[4] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. In *Proceedings 33rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 60–69, 1992.

---

*Even though scanning bit vectors can be done optimally in the cache-oblivious model, this algorithm is not cache oblivious since we need to know $B$ to choose $K$. 

13
[5] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algor.*, 13(1):33–54, 1992.

[6] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Corrigendum: Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algor.*, 15(1):173, 1993.

[7] R. E. Tarjan and R. F. Werneck. Self-adjusting top trees. In *Proceedings 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 813–822, 2005.

[8] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[9] M. L. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[10] M. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[11] M. R. Henzinger and M. L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[12] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *J. Random Structures and Algs.*, 11(4):369–379, 1997.

[13] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[14] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013.

[15] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[16] M. Pătraşcu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[17] M. Patrascu and M. Thorup. Don’t rush into a union: take time to find your roots. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*, pages 559–568, 2011. Technical report available as [arXiv:1102.1783](https://arxiv.org/abs/1102.1783).

[18] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[19] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings 32nd ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2000.
[20] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1757–1769, 2013. Full version available as arXiv:1209.5608.