UNIVERSITY OF COPENHAGEN

Københavns Universitet

Faster worst case deterministic dynamic connectivity

Kejlberg-Rasmussen, Casper; Kopelowitz, Tsvi; Pettie, Seth; Thorup, Mikkel

Faster Worst Case Deterministic Dynamic Connectivity[*]

Casper Kejlberg-Rasmussen[1], Tsvi Kopelowitz[2], Seth Pettie[3], and Mikkel Thorup[†4]

1   **MADALGO, Aarhus University, Aarhus, Denmark**
    `ckr@cs.au.dk`
2   **University of Michigan, Ann Arbor, USA**
    `pettie,tsvi@umich.edu`
3   **University of Michigan, Ann Arbor, USA**
    `pettie,tsvi@umich.edu`
4   **University of Copenhagen, Copenhagen, Denmark**
    `mthorup@di.ku.dk`

Abstract

We present a deterministic dynamic connectivity data structure for undirected graphs with worst case update time $O\left(\sqrt{\frac{n(\log\log n)^2}{\log n}}\right)$ and constant query time. This improves on the previous best deterministic worst case algorithm of Frederickson (*SIAM J. Comput.*, 1985) and Eppstein Galil, Italiano, and Nissenzweig (*J. ACM*, 1997), which had update time $O(\sqrt{n})$. All other algorithms for dynamic connectivity are either randomized (Monte Carlo) or have only amortized performance guarantees.

1   Introduction

Dynamic Connectivity is perhaps the single most fundamental unsolved problem in the area of dynamic graph algorithms. The problem is simply to maintain a dynamic undirected graph $G = (V, E)$ subject to edge updates and connectivity queries:

INSERT(u, v) :   Set $E \leftarrow E \cup \{(u, v)\}$.
DELETE(u, v) :   Set $E \leftarrow E \setminus \{(u, v)\}$.
CONN?(u, v) :   Determine whether u and v are in the same connected component in G.

Over thirty years ago Frederickson [10] introduced *topology trees* and *2-dimensional topology trees*, which gave the first non-trivial solution to the problem. Each edge insertion/deletion is handled in $O(\sqrt{m})$ time and each query is handled in $O(1)$ time. Here m is the current number of edges and n the number of vertices. On sparse graphs (where

---

$m = O(n)$) Frederickson's data structure has not been improved by any deterministic worst case algorithm. However, when the graph is dense Frederickson's data structure can be improved using the general sparsification method of Eppstein, Galil, Italiano, and Nissenzweig [7]. Using simple sparsification [6] the update time becomes $O(\sqrt{n} \log(m/n))$ and using more sophisticated sparsification [7] the running time becomes $O(\sqrt{n})$. This last bound has not been improved in twenty years.

1.1 New Results

In this paper we return to the classical model of deterministic worst case complexity. We give a new dynamic connectivity structure with worst case update time on the order of

$$\min \left\{ \sqrt{\frac{m(\log \log n)^2}{\log n}}, \quad \sqrt{\frac{m \log^5 w}{w}} \right\},$$

where $w = \Omega(\log n)$ is the word size.[1] These are the first improvements to Frederickson's 2D-topology trees [10] in over 30 years. Using the sparsification reduction of Eppstein et al. [7] the running time expressions can be made to depend on 'n' rather than 'm', so we obtain $O(\sqrt{\frac{n(\log \log n)^2}{\log n}})$ bounds (or faster) for all graph densities.

1.2 Related Work

Most research on the dynamic connectivity problem has settled for *amortized* update time guarantees. Following [15, 16], Holm et al. [17] gave a very simple deterministic algorithm with amortized update time $O(\log^2 n)$ and query time $O(\log n/\log \log n)$.[2] However, in the worst case Holm et al.'s [17] update takes $\Omega(m)$ time, the same as computing a spanning tree from scratch! Recently Wulff-Nilsen [25] improved the update time of [17] to $O(\log^2 n/\log \log n)$. Using Las Vegas randomization, Thorup [24] gave a dynamic connectivity data structure with an $O(\log n(\log \log n)^3)$ amortized update time. In other words, the algorithm answers all connectivity queries correctly but the amortized update time holds with high probability.

In a major breakthrough Kapron, King, and Mountjoy [18] used Monte Carlo randomization to achieve a worst case update time of $O(\log^5 n)$. However, this algorithm has three notable drawbacks. The first is that it is susceptible to undetected false negatives: CONN?(u, v) may report that u, v are disconnected when they are, in fact, connected. The second is that even when CONN?(u, v) (correctly) reports that u, v are connected, it is forbidden from exhibiting a connectivity witness, i.e., a spanning forest in which u, v are joined by a path. The Kapron et al. [18] algorithm <u>does</u> maintain such a spanning forest internally, but if this witness were made public, a very simple attack could force the algorithm to answer connectivity queries incorrectly. Lastly, the algorithm uses $\Omega(n \log^2 n)$ space, which for sparse graphs is superlinear in m. Very recently Gibb et al. [13] reduced the update time of [18] to $O(\log^4 n)$.

On special graph classes, dynamic connectivity can often be handled more efficiently. For example, Sleator and Tarjan [22] maintain a dynamic set of trees in $O(\log n)$ worst-case update time subject to $O(\log n)$ time connectivity queries. (See also [1, 3, 15, 23].)

---

[1] Our algorithms 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.

[2] Any connectivity structure that maintains (internally) a spanning forest can have query time $O(\log_{t_u/\log n} n)$ if the update time is $t_u = \Omega(\log n)$.

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

WORST CASE DATA STRUCTURES

REF.	UPDATE TIME	QUERY TIME	NOTES
[10]	$O(\sqrt{m})$	$O(1)$	
[7, 10]	$O(\sqrt{n})$	$O(1)$	[10] + sparsification [7].
[18]	$O(c \log^5 n)$	$O\left(\dfrac{\log n}{\log\log n}\right)$	Randomized Monte Carlo; no connectivity witness;
[13]	$O(c \log^4 n)$	$O\left(\dfrac{\log n}{\log\log n}\right)$	n^c opers. err with prob. n^{-c}.
new	$O\left(\sqrt{\dfrac{n(\log\log n)^2}{\log n}}\right)$ $O\left(\sqrt{\dfrac{n \log^5 w}{w}}\right)$	$O(1)$	$w = \Omega(\log n)$

AMORTIZED DATA STRUCTURES

REF.	AMORT. UPDATE	W.C. QUERY	NOTES
[15]	$O(\log^3 n)$	$O\left(\dfrac{\log n}{\log\log n}\right)$	Randomized Las Vegas.
[16]	$O(\log^2 n)$	$O\left(\dfrac{\log n}{\log\log n}\right)$	Randomized Las Vegas.
[17]	$O(\log^2 n)$	$O\left(\dfrac{\log n}{\log\log n}\right)$	
[24]	$O(\log n(\log\log n)^3)$	$O\left(\dfrac{\log n}{\log\log\log n}\right)$	Randomized Las Vegas.
[25]	$O\left(\dfrac{\log^2 n}{\log\log n}\right)$	$O\left(\dfrac{\log n}{\log\log n}\right)$	

AMORT./WORST CASE LOWER BOUNDS

REF.	UPDATE TIME t_u	QUERY TIME t_q	NOTES
[11, 14, 19]		$t_q = \Omega\left(\dfrac{\log n}{\log(t_u \log n)}\right)$	
[20]	$t_u = \Omega\left(\dfrac{\log n}{\log(t_q/t_u)}\right)$	$t_q = \Omega\left(\dfrac{\log n}{\log(t_u/t_q)}\right)$	Implies $\max\{t_u, t_q\} = \Omega(\log n)$.
[21]	$o(\log n)$ implies	$\Omega\left(n^{1-o(1)}\right)$	

Connectivity in dynamic planar graphs can be reduced to the dynamic tree problem [8, 9], and therefore solved in $O(\log n)$ time per operation. The cell probe lower bounds of Pătraşcu and Demaine [20] show that Sleator and Tarjan's algorithm is optimal in the sense that *some* operation must take $\Omega(\log n)$ time. Superlogarithmic updates can be used to get modestly sublogarithmic queries, but Pătraşcu and Thorup [21] 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 history of upper and lower bounds for dynamic connectivity.

Compared to the amortized algorithms [17, 24, 25], ours is better suited to *online* applications that demand a bound on the latency of *every* operation.[3] Compared to the Monte Carlo algorithms [13, 18], ours is attractive in applications that demand linear space, zero probability of error, and a public witness of connectivity.

2   The High Level Algorithm

The algorithm maintains a spanning tree of each connected component of the graph as a *witness* of connectivity. Each such witness tree T is represented as an Euler tour $\mathrm{Euler}(T)$.[4] $\mathrm{Euler}(T)$ 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 $2(|V(T)| - 1)$ if $|V(T)| \geq 2$ (the last vertex is excluded from the list, which is necessarily the same as the first) or length 1 if $|V(T)| = 1$. Vertices may appear in $\mathrm{Euler}(T)$ several times. We designate one copy of each vertex the *principal copy*, which is responsible for all edges incident to the vertex. Each vertex in the graph maintains a pointer to its principal copy. Each T-edge (u, v) maintains two pointers to the (possibly non-principal) copies of u and v that precede the oriented occurrences of (u, v) and (v, u) in $\mathrm{Euler}(T)$, respectively. Note that cyclic rotations of $\mathrm{Euler}(T)$ are also valid Euler tours; if $\mathrm{Euler}(T) = (u, \ldots, v)$ 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 $\mathrm{Euler}(T_0 \cup \{(u, v)\} \cup T_1)$ from $\mathrm{Euler}(T_0)$ and $\mathrm{Euler}(T_1)$. 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 $\mathrm{Euler}(T_0)$ and $\mathrm{Euler}(T_1)$ from $\mathrm{Euler}(T)$, then look for a *replacement edge*, $(\hat{u}, \hat{v})$ with $\hat{u} \in V(T_0)$ and $\hat{v} \in V(T_1)$. If a replacement is found we construct $\mathrm{Euler}(T_0 \cup \{(\hat{u}, \hat{v})\} \cup T_1)$ from $\mathrm{Euler}(T_0)$ and $\mathrm{Euler}(T_1)$. Lemma 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-principal copies of vertices.

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

Proof. Recall that cyclic shifts of Euler tours are valid Euler tours. Suppose without loss of generality that $\mathrm{Euler}(T) = (P_0, u, v, P_1, v, u, P_2)$ where $P_0, P_1,$ and P_2 are sequences of

---

[3] Amortized data structures are most useful when employed by offline algorithms that do not care about individual operation times. The canonical example is the use of amortized Fibonacci heaps [12] to implement Dijkstra's algorithm [5].

[4] Henzinger and King [15] 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.

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-principal copies of u and v; at least one of the two copies must be non-principal. 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 principal 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 principal 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-principal 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-principal copy of v (or a non-principal copy of u.)   ◄

Thus, we have *reduced* dynamic connectivity in graphs to implementing several simple operations on dynamic lists. Our algorithm maintains a pair $(\mathcal{L}, E)$, where $\mathcal{L}$ is a set of lists (containing principal and non-principal copies of vertices) and E is the dynamic set of edges joining principal 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(L_0, L_1) :   Set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L_0, L_1\} \cup \{L_0 L_1\}$, that is, replace L_0 and L_1 with their concatenation $L_0 L_1$.

SPLIT(x) :   Let $L = L_0 L_1 \in \mathcal{L}$, where x is the last element of L_0. Set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L\} \cup \{L_0, L_1\}$.

REPLACEMENTEDGE(L_0, L_1) :   Return any edge joining elements in L_0 and L_1.

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.

▶ **Invariant 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) \neq $ LIST(v) then make (u, v) a tree edge and perform suitable SPLITs and JOINs 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 $T = T_0 \cup \{(u, v)\} \cup T_1$, perform suitable SPLITs and JOINs to create Euler(T_0) and Euler(T_1) from Euler(T). At this point Invariant 2 may be violated as there could be an edge joining T_0 and T_1. We call REPLACEMENTEDGE(Euler(T_0), Euler(T_1)) and if it finds an edge, say $(\hat{u}, \hat{v})$, we perform more SPLITs and JOINs to form Euler$(T_0 \cup \{(\hat{u}, \hat{v})\} \cup T_1)$.

Henzinger and King [15] 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.

Section 3 gives a relatively simple instantiation of the high-level approach with update time $O(\sqrt{n}/w^{1/4})$, $w = \Omega(\log n)$ being the word size. This is slightly slower than our claimed result. In Section 4 we describe the modifications needed to achieve the claimed bounds.

3   A New Data Structure for Dynamic Lists

3.1   Chunks and Superchunks

In order to simplify the maintenance of Invariant 3, stated below, we shall make two simplifying assumptions. We assume that we have a fixed upper bound $\hat{m}$ on the number of edges and that the maximum degree never exceeds K, where $K \approx \sqrt{\hat{m}/\operatorname{poly}(w)}$. The first assumption is justified by the fact that the sparsification method of [7] creates instances in which $\hat{m}$ is known to be linear in the number of vertices. (It can also be removed by the standard technique of periodic rebuilding.) Refer to Section 5.1 for clean ways to remove the degree-bound assumption.

If L' is a sublist of a list $L \in \mathcal{L}$, define $\operatorname{mass}(L')$ to be the number of edges incident to elements of L', counting an edge twice if both endpoints are in L'.[5] The sum of list masses, $\sum_{L \in \mathcal{L}} \operatorname{mass}(L)$, is clearly at most $2\hat{m}$, where $\hat{m}$ is the fixed upper bound on the number of edges. We maintain a partition of each list $L \in \mathcal{L}$ into *chunks* satisfying Invariant 3.

▶ **Invariant 3.** *Let $L \in \mathcal{L}$ be an Euler tour. If $\operatorname{mass}(L) < K$ then L consists of a single chunk. Otherwise $L = C_0 C_1 \cdots C_{p-1}$ is partitioned into $\Theta(\operatorname{mass}(L)/K)$ chunks such that $\operatorname{mass}(C_l) \in [K, 3K]$ for all $l \in [p] \stackrel{\text{def}}{=} \{0, \ldots, p-1\}$.*

The chunks are partitioned into contiguous sequences of $\Theta(h)$ superchunks according to Invariant 4. For the time being define $h = 2\lfloor \sqrt{w}/2 \rfloor$, where w is the word size.

▶ **Invariant 4.** *A list in $\mathcal{L}$ having fewer than $h/2$ chunks forms a single superchunk with ID $\perp$. A list in $\mathcal{L}$ with at least $h/2$ chunks is partitioned into superchunks, each consisting of between $h/2$ and $h-1$ consecutive chunks. Each such superchunk has a unique ID in $[J] \stackrel{\text{def}}{=} \{0, \ldots, J-1\}$, where $J = 4\hat{m}/(Kh)$. (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 $h/2$ 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 $[J]$. Refer to Section 5.2 for a description of how to handle $\perp$ IDs and short lists.

3.2   Word Operations

When $h \leq \lfloor \sqrt{w} \rfloor$, Invariant 4 implies that we can store a matrix $A \in \{0,1\}^{h \times h}$ in one word that represents the adjacency between the chunks within two superchunks i and j. This matrix will always be represented in row-major order; rows and columns are indexed by $[h] = \{0, \ldots, h-1\}$. In this format it is straightforward to insert a new all-zero row above a specified row k (and destroy row $h-1$) by shifting the old rows $k, \ldots, h-2$ down by one. It is also easy to copy an interval of rows from one matrix to another. Lemma 5 shows that the corresponding operations on *columns* can also be effected in $O(1)$ time with a fixed mask μ precomputable in $O(\log w)$ time.

▶ **Lemma 5.** *Let $h = 2\lfloor \sqrt{w}/2 \rfloor$ and let μ be the word $(1^h 0^h)^{h/2}$. Given μ we can in $O(1)$ time copy/paste any interval of columns from/to a matrix $A \in \{0,1\}^{h \times h}$, represented in row-major order.*

---

5   Remember that edges are only incident to principal copies of vertices, so non-principal copies never contribute any mass.

Proof. Recall that the rows and columns are indexed by integers in $[h] = \{0, \ldots, h-1\}$. We first describe how to build a mask ν_k for columns $k, \ldots, h-1$ then illustrate how it is used to copy/paste intervals of columns. In C notation,[6] the word $\nu'_k = (\mu \gg k) \,\&\, \mu$ is a mask for the intersection of the even rows and columns $k, \ldots, h-1$, so $\nu_k = \nu'_k \mid (\nu'_k \gg h)$ is a mask for columns k through $h-1$.

To insert an all-zero column before column k of A (and delete column $h-1$) we first copy columns $k, \ldots, h-2$ to $A' = A \,\&\, (\nu_{k+1} \ll 1)$ then set $A = (A \,\&\, (\sim\nu_k)) \mid (A' \gg 1)$. 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 h columns together.   ◄

3.3 Adjacency Data Structures

In order to facilitate the efficient implementation of REPLACEMENTEDGE we maintain an $O(\hat{m}/K) \times O(\hat{m}/K)$ 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 $[h]$ indicating its position within the superchunk. Each superchunk maintains its ID in $[J] \cup \{\bot\}$ and a pointer to the list containing it.

- ChAdj is a $J \times J$ array of h^2-bit words ($h^2 \leq w$) indexed by superchunk IDs. The entry ChAdj(i, j) is interpreted as an $h \times h$ 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 kth chunk of superchunk i and the lth 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 $ID(S) = \bot$. By Invariants 2 and 4, S is not incident to any other superchunks and has fewer than $h/2$ 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

    SupAdj$_i(j) = 1$ if ChAdj$(i, j) \neq 0$ and 0 otherwise, whereas
    Memb$_i(j) = 1$ if $j = i$ and 0 otherwise.

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

- We maintain a *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 $\mathcal{S}$ of disjoint lists of superchunks. Each $S \in \mathcal{S}$ (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 $\mathcal{S}$.

---

[6] The operations $\&$, $\mid$, and $\sim$ are bit-wise AND, OR, and NOT; $\ll$ and $\gg$ are left and right shift.

SCINSERT(i) :   Retrieve an unused ID, say i', and allocate a new superchunk with ID i' and all-zero vector $\text{SupAdj}_{i'}$. Insert superchunk i' immediately after superchunk i in i's list in $\mathcal{S}$. If no i is given, create a new list in $\mathcal{S}$ 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 $S_0, S_1 \in \mathcal{S}$ with their concatenation $S_0 S_1$.

SCSPLIT(i) :   Let $S = S_0 S_1 \in \mathcal{S}$ and i be the last superchunk in S_0. Replace $S_0 S_1$ with two lists S_0, S_1.

UPDATEADJ($i, x \in \{0,1\}^J$) :   Set $\text{SupAdj}_i \leftarrow x$ and update $\text{SupAdj}_j(i) \leftarrow x(j)$ 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 $O(\lceil J/w \rceil)$ 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 ith 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.

3.4   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 3 and 4 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 3 and 4.

3.4.1   Splitting Chunks

Suppose we want to split the kth chunk of superchunk i into two pieces, both of which will (at least temporarily) stay within superchunk i.[7] We first zero-out all bits of $\text{ChAdj}(i, \star)(k, \star)$

---

[7] Remember 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.

and $\text{ChAdj}(\star, i)(\star, k)$ in $O(J)$ time. For each j we need to insert an all-zero row below row k in $\text{ChAdj}(i, j)$ and an all-zero column after column k of $\text{ChAdj}(j, i)$. This can be done in $O(1)$ time for each j, or $O(J)$ in total; see Lemma 5.

In $O(K)$ time we scan the edges incident to the new chunks k and $k+1$ and update the corresponding bits in $\text{ChAdj}(i, \star)(k', \star)$ and $\text{ChAdj}(\star, i)(\star, k')$, for $k' \in \{k, k+1\}$.

3.4.2 Merging Adjacent Chunks

In order to merge chunks k and $k+1$ of superchunk i we need to replace row k of $\text{ChAdj}(i, j)$, for all j, with the bit-wise OR of rows k and $k+1$ of $\text{ChAdj}(i, j)$, 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 $\text{ChAdj}(j, i)$, which takes $O(1)$ time per j, by Lemma 5. In total the time is $O(J)$, independent of K.

3.4.3 Splitting Superchunks

Suppose we want to split superchunk i after its kth chunk. We first call $\text{SCINSERT}(i)$, which allocates an empty superchunk with ID i' and inserts i' after i in its superchunk list in $\mathcal{S}$. In $O(J)$ time we transfer rows $k+1, \ldots, h-1$ from $\text{ChAdj}(i, j)$ to $\text{ChAdj}(i', j)$ and transfer columns $k+1, \ldots, h-1$ from $\text{ChAdj}(j, i)$ to $\text{ChAdj}(j, i')$. By Lemma 5 this takes $O(1)$ time per j.

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

3.4.4 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 h chunks together. In $O(J)$ time we transfer the non-zero rows of $\text{ChAdj}(i', j)$ to $\text{ChAdj}(i, j)$ and transfer the non-zero columns of $\text{ChAdj}(j, i')$ to $\text{ChAdj}(j, i)$. A call to $\text{SCDELETE}(i')$ deletes superchunk i' from its list in $\mathcal{S}$ and retires ID i'. We then call $\text{UPDATEADJ}(i, x)$ 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_i and $\text{SupAdj}_{i'}$, with bit i' set to zero. Updating the list-sum data structure takes $O(J)$ time.

3.5 Joining and Splitting Lists

Once we have routines for splitting and merging adjacent (super)chunks, implementing JOIN and SPLIT on lists in $\mathcal{L}$ is much easier. The goal is to restore Invariant 3 governing chunk masses and Invariant 4 on the number of chunks per superchunk.

3.5.1 Performing $\text{JOIN}(L_0, L_1)$

Write $L_0 = C_0, \ldots, C_{p-1}$ and $L_1 = D_0, \ldots, D_{q-1}$ as a list of chunks. If both L_0 and L_1 are not short then they have corresponding superchunk lists $S_0, S_1 \in \mathcal{S}$. Call $\text{SCJOIN}(S_0, S_1)$ to join S_0, S_1 in $\mathcal{S}$, in $O(J)$ time.

3.5.2   Performing $\mathrm{SPLIT}(x)$

Suppose x is contained in chunk C_l of $L = C_0 \cdots C_{l-1} C_l C_{l+1} \cdots C_{p-1}$. We split C_l into two chunks $C_l' C_l''$, and split the superchunk containing C_l along this line. Let S be the superchunk list corresponding to L and i be the ID of the superchunk ending at C_l'. We split S using a call to $\mathrm{SCSPLIT}(i)$, which corresponds to splitting L into $L_0 = C_0 \cdots C_{l-1} C_l'$ and $L_1 = C_l'' C_{l+1} \cdots C_{p-1}$. At this point C_l' or C_l'' may violate Invariant 3 if $\mathrm{mass}(C_l') < K$ or $\mathrm{mass}(C_l'') < K$. Furthermore, Invariant 4 may be violated if the number of chunks in the superchunks containing C_l' and C_l'' is too small. We first correct Invariant 3 by possibly merging and resplitting $C_{l-1} C_l'$ and $C_l'' C_{l+1}$ along new boundaries. If the superchunk containing C_l' has fewer than $h/2$ chunks, it and the superchunk to its left have strictly between $h/2$ and $3h/2$ chunks together, and so can be merged (and possibly resplit) into one or two superchunks satisfying Invariant 4. The same method can correct a violation of C_l''''s superchunk. This takes $O(K + J)$ time.

3.5.3   Performing $\mathrm{REPLACEMENTEDGE}(L_0, L_1)$

The list-sum data structure makes implementing the $\mathrm{REPLACEMENTEDGE}(L_0, L_1)$ 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 $\alpha \leftarrow \mathrm{ADJQUERY}(S_0)$ and $\beta \leftarrow \mathrm{MEMBQUERY}(S_1)$ and their bit-wise AND $\alpha \wedge \beta$ with a linear scan of both vectors. If $\alpha \wedge \beta$ is the all-zero vector then there is no edge between L_0 and L_1. On the other hand, if $(\alpha \wedge \beta)(j) = 1$, 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 $\mathrm{SupAdj}^z(j) = 1$. Once i and j are known we retrieve any 1-bit in the matrix $\mathrm{ChAdj}(i, j)$, say at position (k, l), indicating that the kth chunk of superchunk i and the lth chunk of superchunk j are adjacent. We scan all its adjacent edges in $O(K)$ time and retrieve an edge joining L_0 and L_1. The total time is $O(J/w + \log J + K) = O(J/w + K)$.

3.5.4   Performing $\mathrm{INSERT}(u, v)$

If $\mathrm{LIST}(u) \neq \mathrm{LIST}(v)$, first perform $O(1)$ SPLITs and JOINs to restore the Euler tour Invariant 2. Now u and v are in the same list in $\mathcal{L}$. Let i, j be the IDs of the superchunks containing the principal copies of u and v and let k, l be the positions of u and v's chunks within their respective superchunks. We set $\mathrm{ChAdj}(i, j)(k, l) \leftarrow 1$. If $\mathrm{ChAdj}(i, j)$ was formerly the all-zero matrix, we call $\mathrm{UPDATEADJ}(i, x)$ to update superchunk i's adjacency information with the correct vector x.[8] Inserting one edge changes the mass of the chunks containing u and v, which could violate Invariant 3. Invariants 3 and 4 are restored by splitting/merging $O(1)$ chunks and superchunks.

3.5.5   Performing $\mathrm{DELETE}(u, v)$

Compute i, j, k, l as defined above, in $O(1)$ time. After we delete (u, v) the correct value of the bit $\mathrm{ChAdj}(i, j)(k, l)$ is uncertain. We scan chunk k of superchunk i in $O(K)$ time, looking for an edge connected to chunk l of superchunk j. If we do *not* find such an edge we set $\mathrm{ChAdj}(i, j)(k, l) \leftarrow 0$, and if that makes $\mathrm{ChAdj}(i, j) = 0$ (the all-zero matrix), we call

---

[8] Since x only differs from the former SupAdj_i at position $\mathrm{SupAdj}_i(j)$, this update to the list-sum tree takes just $O(\log J)$ time since it only affects ancestors of leaves i and j.

UPDATEADJ(i, x), where x is the new adjacency vector of superchunk i; it only differs from the former SupAdj$_i$ 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 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.

3.6 Running Time Analysis

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

By the sparsification transformation of Eppstein, Galil, Italiano, and Nissenzweig [7] this implies an update time of $O\left(\frac{\sqrt{n}}{w^{1/4}}\right)$. Each instance of dynamic connectivity created by [7] 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.

4 Speeding Up the Algorithm

Observe that there are $\Theta((\hat{m}/(Kh))^2)$ matrices (ChAdj(i, j)) but only $\hat{m}$ edges, so for $K = \sqrt{\hat{m}/h}$, the average $h \times h$ matrix has $O(h)$ 1s. Thus, storing each such matrix verbatim, using h^2 bits, is information theoretically inefficient on average. By storing only the locations of the 1s in each matrix we can represent each matrix in $O(h \log h)$ bits on average and thereby hope to solve dynamic connectivity faster with a larger 'h' parameter.

4.1 The Encoding

In this encoding we index rows and columns by indices in $\{1, \ldots, h\}$ rather than $[h]$. Let $m_{i,j} = m_{j,i}$ be the number of 1s in ChAdj(i, j). We encode ChAdj(i, j) by listing its 1 positions in $O(m_{i,j} \log h/w)$ lightly packed words. Each word is partitioned into *fields* of $1 + 2\lceil \log(h + 1) \rceil$ bits: each field consists of a control bit (normally 0), a row index, and a column index. Each word is between half-full and full, the fields in use being packed contiguously in the word. This invariant allows us to insert a new field after a given field in $O(1)$ time. We list the 1s of *either* ChAdj(i, j) *or* ChAdj$(j, i) = $ ChAdj$(i, j)^\top$ in row-major order, with a bit indicating which of the two representations is used.

4.2 Fast Operations

Given ChAdj(i, j) in row-major order, we can determine if ChAdj$(i, j)(k, l) = 1$ in $O(\log h)$ time by doing a binary search to find the correct word in the list, then a binary search within the word to find an entry $\langle k, l \rangle$, if any.[9]

---

[9] The time for a search can actually be improved to $O(\log((m_{i,j} \log h)/w))$, which is faster when ChAdj(i, j) has average density ($m_{i,j}$ is close to h) but is still $O(\log h)$ in the worst case. We still do a binary search over the first field in each word to determine which word (if any) has a field containing $\langle k, l \rangle$: the binary encoding of (k, l). This takes $O(\log((m_{i,j} \log h)/w))$ time since there are $\Theta((m_{i,j} \log h)/w)$ lightly packed words. If we add $2^{2\lceil \log(h+1) \rceil} - \langle k, l \rangle$ to each field in the word, the control bits for all fields that are equal to or greater than $\langle k, l \rangle$ will be flipped to 1. Similarly, if we set all control bits to 1

In the same time bound we can also identify the positions of the first and last 1s in row k. Thus, we can perform the following operations on $\mathrm{ChAdj}(i,j)$ in $O((m_{i,j}\log h)/w)$ time: setting a row to zero, incrementing/decrementing the row-index of some interval of rows, or copying an interval of rows.

The operations sketched above are only efficient if $\mathrm{ChAdj}(i,j)$ is in row-major order. If we have $\mathrm{ChAdj}(i,j)^\top$ in row-major order we can effect a transpose by (1) swapping the row and column indices in each field using masks and shifts, and (2) sorting the fields. In general, sorting x words of $O(w/\log h)$ fields takes $O(x(\log^2(w/\log h) + \log x \log(w/\log h)))$ time using Albers and Hagerup's implementation [2] of Batcher's bitonic mergesort [4].[10] We sort each word in $O(\log^2(w/\log h))$ time, resulting in x sorted lists, then iteratively merge the two shortest lists until one list remains. Merging two lists containing y words takes $O(y \log(w/\log h))$ time: we can merge the next $w/\log h$ fields of each list in $O(\log(w/\log h))$ time [2] and output at least $w/\log h$ items to the merged list.

Alternatively, if $w = \log n$ we can sort and merge lists of $\epsilon \log n/\log h$ fields in unit time using table lookup to precomputed tables of size $O(n^\epsilon)$. In this case sorting x packed words takes $O(x \log x)$ time.

4.3   Splitting and Joining

The cost of splitting and joining (super)chunks is now slightly more expensive. When handling superchunk i (or any chunk within it) we first put each $\mathrm{ChAdj}(i,j)$ in row-major order, in $\sum_{j=1}^{J} O(\lceil \frac{m_{i,j}\log h}{w}\rceil \log^2 h) = O(J\log^2 h + (Kh/w)\log^3 h)$ since, by Invariants 3 and 4, $\sum_j m_{i,j} = O(Kh)$. Once the relevant superchunks are in the correct format, splitting or joining $O(1)$ (super)chunks takes $O(K\log h + J + (Kh/w)\log h)$ time. Since $J = O(\hat{m}/(Kh))$, the overall update time is

$$O\left(K\log h + \frac{\hat{m}\log^2 h}{Kh} + \frac{Kh\log^3 h}{w}\right).$$

Setting $h = w$ and $K = \sqrt{\frac{\hat{m}}{w\log w}}$, the overall time is $O(\sqrt{\frac{\hat{m}\log^5 w}{w}})$. When $w = O(\log n)$ the cost of taking the transpose is cheaper since sorting and merging a packed word takes unit time via table lookup. Setting $h = \log n$, the total time is

$$O\left(K\log\log n + \frac{\hat{m}}{K\log n} + K(\log\log n)^2\right),$$

which is $O(\sqrt{\frac{\hat{m}(\log\log n)^2}{\log n}})$ when $K = \sqrt{\frac{\hat{m}}{\log n(\log\log n)^2}}$.

---

and subtract $\langle k,l\rangle + 1$ from each field, the control bits of fields that are equal to or less than $\langle k,l\rangle$ will be flipped to 0. Thus, we can single out the control bit for an occurrence of $\langle k,l\rangle$ (if any) with $O(1)$ bit-wise operations. If $\langle k,l\rangle$ is not present, the control bits reveal the field in the word after which it could be inserted, if we need to set $\mathrm{ChAdj}(i,j)(k,l) \leftarrow 1$.

[10] Albers and Hagerup also require that the fields to be sorted begin with control bits.

5   Loose Ends

5.1   Removing the Bounded Degree Assumption

Invariants 3 and 4 imply that there are $J = \Theta(\hat{m}/(Kh))$ superchunks with non-$\perp$ IDs. However, Invariant 3 cannot be satisfied (as stated) unless the maximum degree is bounded by $O(K)$. One way to guarantee this is to physically split up high degree vertices, replacing each v with a cycle on new vertices $v_1, \ldots, v_{\lceil \deg(v)/\Theta(K) \rceil}$, each of which is responsible for $\Theta(K)$ of v's edges. This is the method used by Frederickson [10], who actually demanded that the maximum degree be 3 at all times!

This vertex-splitting can be effectively simulated in our algorithm as follows. If $\deg(v) \geq K/2$, replace the principal copy of v in its Euler tour with an interval of artificial principal vertices $v_1, \ldots, v_{\lceil \deg(v)/(K/2) \rceil}$, each of which is responsible for between $K/2$ and K of v's edges. Invariant 3 is therefore maintained w.r.t. this modified tour. To keep the mass of artificial vertices between $K/2$ and K, each edge insertion/deletion may require splitting an artificial vertex or merging two consecutive artificial vertices. When the Euler tour changes we always preserve the invariant that v's artificial vertices form a contiguous interval in the tour.

5.2   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 $h/2$ 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 ID$(S) = \perp$. 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 ID$(S) \leftarrow i$, set ChAdj$(i, i) \leftarrow$ ChAdj$_S$, and destroy ChAdj$_S$. By Invariant 2, S was not incident to any other superchunk, so ChAdj$(i, j) = 0$ (the all-zero matrix) for all $j \neq i$. At this point S violates Invariant 4 (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 having associated Euler tours L_0 and L_1, and REPLACEMENTEDGE(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 ChAdj$_S \leftarrow$ ChAdj$(\text{ID}(S), \text{ID}(S))$, then set ChAdj$(\text{ID}(S), \text{ID}(S)) \leftarrow 0$ and finally retire ID(S).

The implementation of REPLACEMENTEDGE(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 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 ChAdj$_S$. Of course ChAdj$_S$ is represented as a list of its 1 positions in row-major order, so we can isolate the correct intervals of rows and columns in $O(h^2 \log^3 h/w)$ time. If there is any 1 there, say at location 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 ChAdj$_S$ must be updated to reflect any splits and joins that take place, and if no replacement edge is discovered, ChAdj$_S$ must be split into two lists representing matrices ChAdj$_{S_0}$ and ChAdj$_{S_1}$, to be identified with the single superchunks S_0 and S_1 in L_0 and L_1, respectively.

──── **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. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997. `doi:10.1006/inco.1997.2632`.

3   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. `doi: 10.1145/1103963.1103966`.

4   T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd ed.* MIT Press, 2009.

5   E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

6   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.

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

8   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.

9   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.

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

11 M. L. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

12 M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

13 D. Gibb, B. M. Kapron, V. King, and N. Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

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

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

16 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.

17 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.

18 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.

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

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

21   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.

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

23   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.

24   M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings 32nd ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2000.

25   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.