Word-packing Algorithms for Dynamic Connectivity and Dynamic Sets

Casper Kejlberg-Rasmussen∗ Tsvi Kopelowitz † Seth Pettie ‡ Ely Porat §

Abstract

We examine several (dynamic) graph and set intersection problems in the word-RAM model with word size $w$. We begin with Dynamic Connectivity where we need to maintain a fully dynamic graph $G = (V,E)$ with $n = |V|$ vertices while supporting $(s,t)$-connectivity queries. To do this, we provide a new simplified worst-case solution for the famous Dynamic Connectivity (which is interesting on its own merit), and then show how in the word-RAM model the query and update cost can be reduced to $O(\sqrt{n} \cdot \log n \log(w \log n))$, assuming $w < n^{1-\Omega(1)}$. Since $w = \Omega(\log n)$, this bound is always $O(\sqrt{n})$ and it is $o(\sqrt{n})$ when $w = \omega(\log n)$.

We then examine the task of maintaining a family $F$ of dynamic sets where insertions and deletions into the sets are allowed, while enabling a set intersection reporting query on sets $S_1, S_2 \in F$ where we wish to report all of the elements in $S_1 \cap S_2$. We first show that given a known upper-bound $d$ on the size of any set, we can maintain $F$ so that a set intersection reporting query costs $O(\frac{d}{w/\log^2 w})$ expected time, and updates cost $O(\log w)$ expected time. Using this algorithm we can list all triangles of a graph $G = (V,E)$ in $O(\frac{m\alpha}{w/\log^2 w} + t)$ expected time where $m = |E|$, $\alpha$ is the arboricity of $G$, and $t$ is the size of the output. This is comparable with known algorithms that run in $O(m\alpha)$ time.

Next, we provide an incremental data structure on $F$ that supports intersection proof queries in which given $S_1$ and $S_2$ we wish to determine if they intersect, and if they do we require a proof (an element in the intersection). Both queries and insertions of elements into sets take $O(\sqrt{\frac{N}{w/\log^2 w}})$ expected time, where $N = \sum_{S \in F} |S|$. Finally, we provide time/space tradeoffs for the fully dynamic set intersection listing problem so that using $M$ words of space each update costs $O(\sqrt{M \log N})$ expected time, each reporting query costs $O(N \sqrt{\frac{\log N}{M}} \sqrt{\log \log N} + 1)$ expected time where $op$ is the size of the output, and each proof query costs $O(N \sqrt{\frac{\log N}{M}} + \log N)$ expected time.

1 Introduction

In this paper we explore the power of word level parallelism to speed up algorithms for dynamic set intersection, dynamic connectivity, and triangle enumeration. We assume a $w$-bit word-RAM model, $w > \log n$, with the standard repertoire of unit-time operations on $w$-bit words: bitwise Boolean operations, left/right shifts, addition, comparison, and dereferencing. Using the modest

∗MADALGO, Aarhus University
†University of Michigan, Email: kopelow@gmail.com
‡University of Michigan, Email: pettie@gmail.com. Supported by NSF grants CCF-1217338 and CNS-1318294 and a grant from the US-Israel Binational Science Foundation. This research was performed in part at the Center for Massive Data Algorithmics (MADALGO) at Aarhus University, which is supported by the Danish National Research Foundation grant DNRF84.
§Bar-Ilan University Email: porat@cs.biu.ac.il
parallelism intrinsic in this model (sometimes in conjunction with tabulation) it is often possible to obtain a nearly factor-$w$ (or factor-log $n$) speedup over traditional algorithms. The Four Russians algorithm for boolean matrix multiplication is perhaps the oldest algorithm to use this technique. Since then it has been applied to computing edit distance [MP80], regular expression pattern matching [Mye92], APSP in dense weighted graphs [Cha10], APSP and transitive closure in sparse graphs [Cha12, Cha08], and more recently, to computing the Fréchet distance [BBMM14] and solving 3SUM in subquadratic time [BDP08, GP14]. Refer to [Cha13] for more examples.

**Dynamic Connectivity.** The problem is to answer connectivity queries in a graph subject to insertion and deletion of edges. The best known worst-case algorithm (sparsification [EGIN97] applied to Frederickson’s [Fre85] algorithm) performs edge updates in $O(\sqrt{n})$ time and queries in $O(1)$ time. All subsequent dynamic connectivity algorithms are either amortized [HK99, HT97, HdT01, Tho00, WN13] or use Monte Carlo randomization [KKM13] and therefore have some probability of error. See Table 1 in the appendix for a summary of dynamic connectivity structures. We present a new worst-case dynamic connectivity structure that is conceptually simpler than Frederickson’s and potentially faster. The update time is $O(\sqrt{n} \cdot \log \frac{n}{\log w})$, which matches Frederickson [Fre85] when $w = \Theta(\log n)$ and is faster when $w = \omega(\log n)$. In the external memory model, where memory is accessed in blocks of $B$ $w$-bit words, the running time becomes $O(\sqrt{n} \cdot \log \frac{n}{Bw} \log(\frac{Bw}{\log n}))$.

**Set Intersection.** The problem is to represent a (possibly dynamic) family of sets $F$ with total size $N = \sum_{S \in F} |S|$ so that given $S, S' \in F$, one can quickly determine if $S \cap S' = \emptyset$ (emptiness query) or report some $x \in S \cap S'$ (witness query) or report all members of $S \cap S'$. Let $d$ be an \emph{a priori} bound on the size of any set. We give a randomized algorithm to preprocess $F$ in $O(N)$ time such that reporting queries can be answered in $O(d/\log w + |S \cap S'| \log \log w)$ expected time. Subsequent insertion and deletion of elements can be handled in $O(\log w)$ time.

We give $O(N)$-space structures for the three types of queries when there is no restriction on the size of sets. For emptiness queries the worst-case update/query times are $O(\sqrt{N})$, for witness queries the update/query times are $O(\sqrt{N \log N})$, and for reporting queries the update time is $O(\sqrt{N \log N})$ and the query time $O(\sqrt{N \log N (1 + |S \cap S'|)}).$ These fully dynamic structures do not benefit from word-level parallelism. When only insertions are allowed we give another structure that handles both insertions and emptiness/witness queries in $O(\sqrt{N/w} \log^{2+1/w} n)$ time\footnote{These bounds assume $B$ and $w$ are not too large. The running time is always $\Omega(\log n)$ regardless of how large $B$ and $w$ are.}

**Related work on set intersection.** Most existing set intersection data structures, e.g., [DLOM00, BK02, BY04], work in the comparison model, where sets are represented as sorted lists or arrays. In these data structures the main benchmark is the minimum number of comparisons needed to certify the answer. When restricting the size of our sets, the most related structure is that of Bille, Pagh, and Pagh [BPP07] where similar yet more complicated techniques are used. They gave an algorithm for preprocessing $m$ sets with a total of $n$ elements so that the intersection of all $m$ sets can be computed in $O(n/(\log n/w + km))$ time, where $k$ is the size of the output. The structure most closely related to our structures with no upper bound on the size of sets is from Cohen and

\begin{equation}
\frac{1}{2} \sum_{i=1}^{m} \frac{1}{\log_{2} (\frac{n}{\log_{2} (\frac{n}{\log_{2} (\frac{n}{\log_{2} (\frac{n}{\log_{2} (\frac{n}{\log_{2} n})})})})})}
\end{equation}
Porat [CP10]. They gave a static $O(N)$-space data structure for answering reporting queries in $O(\sqrt{N(1 + |S \cap S'|)})$ time.

**Triangle Enumeration.** Itai and Rodeh [IR78] showed that all $t$ triangles in a graph could be enumerated in $O(m^{3/2})$ time. Thirty years ago Chiba and Nishizeki [CN85] generalized [IR78] to show that $O(m\alpha)$ time suffices, where $\alpha$ is the arboricity of the graph. This algorithm has only been improved for dense graphs using fast matrix multiplication. The recent algorithm of Bjorklund, Pagh, Williams, and Zwick [BPWZ14] shows that when the matrix multiplication exponent $\omega = 2$, triangle enumeration takes $\tilde{O}(\min\{n^2 + nt^{2/3}, m^{4/3} + mt^{1/3}\})$ time. (The actual running time is expressed in terms of $\omega$.) We give the first asymptotic improvement to Chiba and Nishizeki’s algorithm for graphs that are too sparse to benefit from fast matrix multiplication. Using our set intersection data structure, we can enumerate $t$ triangles in $O(m + m\alpha/\log^2 w + t \log \log w)$ expected time.

For simplicity we have stated all bounds in terms of an arbitrary word size $w$. When $w = O(\log n)$ the $w/\log^2 w$ factors become $\log n/\log \log n$ and the $\log \log w$ factors disappear entirely.

**Overview of the paper.** The paper is structured as follows. In Section 2 we present our new algorithm for dynamic connectivity. Notice that one can skip directly to Section 3 without reading Section 2. In Section 3 we discuss a packing algorithm for (dynamic) set intersection, and in Section 4 we show how the packing algorithm for set interseciton can be used to speed up triangle listing. In Section 5 we present our data structure for emptiness queries on a fully dynamic family of sets. In Section 6 we combine the packing algorithm for set intersection with the emptiness query data structure to obtain a packed data structure for set intersection proof queries on an incremental family of sets. Finally, in the Appendix we present non-packed data structures for emptiness, proof, and reporting set intersection queries on a fully dynamic family of sets, with memory/time tradeoffs.

## 2 A Simplified 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 $\text{Euler}(T)$, which is not uniquely determined by the structure of $T$. Vertices may appear in $\text{Euler}(T)$ several times. We designate one copy of each vertex the *principle copy*.

When an edge $(u, v)$ is inserted that connects distinct witness trees $T$ and $T'$, $(u, v)$ becomes a tree edge and we need to construct $\text{Euler}(T \cup \{(u, v)\} \cup T')$ from $\text{Euler}(T)$ and $\text{Euler}(T')$. In the reverse situation, if a tree edge $(u, v)$ is deleted from $T$, we need to find a replacement edge $(\hat{u}, \hat{v})$ (if one exists) and construct $\text{Euler}(T \setminus \{(u, v)\} \cup \{(\hat{u}, \hat{v})\})$. Lemma 2.1 establishes the nearly obvious fact that the new Euler tours can be obtained from the old Euler tours using $O(1)$ surgical operations.

**Lemma 2.1.** After each edge insertion or deletion, the Euler tours of all witness trees can be updated with $O(1)$ surgical operations: splitting a list in two, concatenating two lists, and creating/destroying lists containing non-principle copies of vertices. In particular, if $T$ is a witness tree, $e \in T$, and $\hat{e}$ a replacement edge joining the two trees of $T \setminus \{e\}$, a tour $\text{Euler}(T \setminus \{e\} \cup \{\hat{e}\})$ can be obtained from $\text{Euler}(T)$ by $O(1)$ surgical operations.


Proof. Let $e = (u, v)$ and $\hat{e} = (\hat{u}, \hat{v})$. Suppose without loss of generality that $\hat{v}$ is in $v$’s component of $T \setminus \{ e \}$ and an occurrence of $\hat{u}$ appears after $(v, u)$ in Euler($T$). See Figure 1. We can write Euler($T$) as \((P_1, u, v, P_2, \hat{v}, P_3, v, u, P_4, \hat{u}, P_5)\), where \((P_1, u, \hat{v}, P_4, \hat{u}, \hat{v}, P_5)\) and \((v, P_2, \hat{v}, P_3, v, P_2, \hat{v}, \hat{u}, P_5)\) are the Euler tours of the two components in $T \setminus \{ e \}$. One valid tour for Euler($T \setminus \{ e \} \cup \{ \hat{e} \}$) is \((P_1, u, P_4, \hat{v}, \hat{v}, P_3, v, P_2, \hat{v}, \hat{u}, P_5)\), which can be obtained from Euler($T$) using $O(1)$ surgical operations. The other cases (inserting an edge, deleting a tree edge without substituting a replacement) are handled similarly.

Remark 2.1. Note that in Euler($T \setminus \{ e \} \cup \{ \hat{e} \}$), the multiplicities of $\hat{u}$ and $\hat{v}$ are incremented and those of $u$ and $v$ decremented, relative to their respective multiplicities in Euler($T$). We have a choice of which copy of $u$ and $v$ to destroy. For example, the $u$ between $P_1$ and $P_4$ in Euler($T \setminus \{ e \} \cup \{ \hat{e} \}$) can either be the $u$ following $P_1$ or the $u$ preceding $P_4$ in Euler($T$). When manipulating these Euler tours we always retain the principle copy of each vertex and discard only non-principle copies.

The worst case running time of our edge insertion/deletion procedure will ultimately depend on $m$, the current number of edges in the graph. For the sake of simplicity we let $\hat{m}$ be an upper bound on the number of edges and analyze the running time in terms of $\hat{m}$. It is straightforward to obtain bounds in terms of $m$ rather than $\hat{m}$ by progressively rebuilding the data structure when $m$ gets close $\hat{m}$ or $\hat{m}/4$; this is explained in Section 2.3. Using the sparsification method of [EGIN97] the running time can be made to depend on $n$ rather than $m$.

2.2 A Dynamic List Data Structure

We have reduced dynamic connectivity in graphs to the following structurally simpler problem on dynamic lists. The problem is to maintain a pair \((\mathcal{L}, \mathcal{E})\), where $\mathcal{L}$ is a set of lists and $\mathcal{E}$ a set of edges joining elements of the lists. In our situation $\mathcal{L}$ is the set of Euler tours and $\mathcal{E}$ are only incident to the principle copies of vertices. By assumption $|\mathcal{E}| \leq \hat{m}$.
JOIN($L_0, L_1$) : Set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L_0, L_1\} \cup \{L_0L_1\}$, that is, replace $L_0$ and $L_1$ with their concatenation $L_0L_1$.

SPLIT($u$) : Let $L = L_0L_1 \in \mathcal{L}$ be the list containing $u$, where $u$ is the last element in $L_0$. Set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L\} \cup \{L_0, L_1\}$.

INSERT($e$) : Set $\mathcal{E} \leftarrow \mathcal{E} \cup \{e\}$.

DELETE($e$) : Set $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$.

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

CUT($u$, $v$) : A precondition is that LIST($u$) = LIST($v$). Let $L = L_0L_1L_2$ be the list containing $u$ and $v$, and $L_1$ the minimal sublist containing $u$ and $v$. Report any edge $(\hat{u}, \hat{v}) \in \mathcal{E}$ with $\hat{u} \in L_1$ and $\hat{v} \in L_0L_2$.

If $L'$ is a sublist of a member of $\mathcal{L}$, let $\text{mass}(L')$ 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, $\sum_{L \in \mathcal{L}} \text{mass}(L)$, is clearly at most $M \overset{\text{def}}{=} 2\bar{m}$. 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 $M$, but may deviate from this ideal (below or above) out of necessity.

**Invariant 2.1.** Each $L \in \mathcal{L}$ is partitioned into chunks $L = I_0I_1 \cdots I_{l-1}$ such that for all $i \in [l]$, exactly one of the following mass criteria is satisfied.

1. **Criterion 1.** $K \leq \text{mass}(I_i) \leq 4K$ and all elements of $I_i$ have mass at most $2K$.
2. **Criterion 2.** $\text{mass}(I_i) > 2K$ and $I_i$ consists of a single element, or
3. **Criterion 3.** $\text{mass}(I_i) < K$ and both $I_{i-1}$ and $I_{i+1}$, if they exist, satisfy Criterion 2.

**Lemma 2.2.** Let $\mathcal{L}$ be a set of lists whose total mass is at most $M$. Any partition of its lists into chunks that conforms to Invariant 2.1 uses at most $M/K + |\mathcal{L}|$ 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.

Motivated by Lemma 2.2, we maintain a chunk identifier function $\text{ID} : \text{Chunks} \rightarrow [M/K] \cup \{\bot\}$. The last chunk in a list may have no id ($\bot$) only if it satisfies Criterion 3. Aside from these chunks the ID function is injective. Extending the notation, let $\text{ID}(u)$ be the id of the chunk containing list element $u$. Each element $u$ stores $\text{ID}(u)$ and each chunk $i$ maintains two length-$M/K$ bit-vectors.

- $\text{Adj}_i(j) = \begin{cases} 1 & \text{if there is a } (u, v) \in \mathcal{E} \text{ such that } \text{ID}(u) = i \text{ and } \text{ID}(v) = j \\ 0 & \text{otherwise} \end{cases}$
- $\text{Memb}_i(j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$

At this point we can begin to describe how each operation works.

---

3Note that in this abstract data structure, there is no prohibition against edges between distinct lists, though our dynamic connectivity data structure does, in fact, maintain this prohibition since lists are identified with connected components. Inserting and deleting edges does not automatically trigger any restructuring of the lists.
Performing \textbf{INSERT}(u,v) or \textbf{DELETE}(u,v). First consider an \textbf{INSERT}(u,v) operation and let 
\((i,j) = (\text{ID}(u), \text{ID}(v))\). If either \(i\) or \(j\) is \(\perp\) then we do nothing. If \(\text{Adj}_j(j) \overset{\text{def}}{=} \text{Adj}_j(i) = 0\) we set \(\text{Adj}_j(j), \text{Adj}_j(i) \leftarrow 1\); otherwise we do nothing. When performing a \textbf{DELETE}(u,v) operation we need to determine if \(\text{Adj}_j(j)\) and \(\text{Adj}_j(i)\) should be 0 or 1. If the chunks with ids \(i\) and \(j\) satisfy Criterion 2 then they each consist of a single element (namely \(u\) and \(v\)) so we can set \(\text{Adj}_j(j), \text{Adj}_j(i) \leftarrow 0\). If either is a Criterion 1 or Criterion 3 chunk, say chunk \(I\) (or if mass(\(I\)) = 2 then they each consist of a single element (namely \(u\) and \(v\)) so we can set \(\text{Adj}_j(j), \text{Adj}_j(i) \leftarrow 0\). If either is a Criterion 1 or Criterion 3 chunk, say chunk \(I\), then it has mass \(O(K)\) and we can afford to scan all its incident edges, looking for one connected to chunk \(i\) either is a Criterion 1 or Criterion 3 chunk, say chunk \(I\). If mass(\(I\)) > 2 \(K\), we temporarily split \(I\) into three chunks \(I'_1I'_2I'_3\), where \(I'_1 = (u)\) now satisfies Criterion 2. If mass(\(I'_1\)) < \(K\) and \(I_{i-1}\) satisfies Criterion 2 (or if mass(\(I'_2\)) < \(K\) and \(I_{i+1}\) satisfies Criterion 2) then \(I'_3\) (and \(I'_2\)) is a valid Criterion 3 chunk. If \(I'_3\) or \(I'_2\) do not satisfy Criteria 1 or 3 we need to correct this violation. If mass(\(I_{i-1}\)) + mass(\(I'_1\)) ≤ \(4K\) then we can join \(I_{i-1}\), \(I'_1\) into a single Criterion 1 chunk. Otherwise we split \(I_{i-1}\) as evenly as possible into chunks \(I'_{i-1}I''_{i-1}\). Since mass(\(I_{i-1}\)) + mass(\(I'_1\)) ≤ \(6K\) and none of their elements has mass more than \(2K\), we can ensure that \(K \leq \text{mass}(I'_{i-1}), \text{mass}(I''_{i-1}) \leq 4K\), 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 \(I\) to \(2K\) violates Criterion 2 locally, and may violate \(I_{i-1}\) and/or \(I_{i+1}\) if they are Criterion 3 chunks. However, the total mass of these violated chunks is between \(2K\) and \(4K\), so they can be joined into a single Criterion 1 chunk.

Creating and Destroying Chunks. We maintain an unsorted list of unused chunk ids. Re-linquishing an id and claiming an unused id take \(O(1)\) time. In the procedure above, Criterion 2 chunks are never destroyed, so we can assume a chunk \(I\) to be destroyed has mass(\(I\)) = \(O(K)\). We begin by retiring \(\text{ID}(I)\), then proceed to zero-out \(\text{Adj}_{\text{ID}(I)}(\ast)\) and the \(O(K)\) non-zero locations in \(\text{Adj}_{\ast}(\text{ID}(I))\). To create a chunk \(I\) we just reverse that process. We retrieve a new id for \(I\), then scan the edges incident to \(I\) elements and update the \(O(\text{mass}(I)) = O(K)\) relevant locations of \(\text{Adj}_{\text{ID}(I)}(\ast)\) and \(\text{Adj}_{\ast}(\text{ID}(I))\).

Performing \textbf{JOIN}(L_0,L_1). Write \(L_0 = I_0,\ldots,I_{p-1}\) and \(L_1 = J_0,\ldots,J_{q-1}\) as a list of chunks, then form \(L = I_0,\ldots,I_{p-1},J_0,\ldots,J_{q-1}\) by concatenating the lists. If \(L\) satisfies the mass criteria of Invariant \(2.1\) then we are done. In particular, no adjustments are necessary if \(I_{p-1}\) and \(J_0\) both satisfy Criteria 1 or 2. In general we create/destroy \(O(1)\) chunks around \(I_{q-1}J_0\) to restore Invariant \(2.1\) as described earlier. Note that even if no chunk boundaries need to be modified, we
may need to create a full-fledged chunk out of $I_{p-1}$ if it satisfies Criterion 3, $J_0$ satisfies Criterion 2, and ID($I_{p-1}$) = ⊥.

**Performing** Split($u$). Suppose $u$ is contained in chunk $I_i$ in $L = I_0 \cdots I_{i-1} I_{i+1} \cdots I_{p-1}$. We temporarily replace $I_i$ with two chunks $I'_i I''_i$, then split $L$ into $L_0 = I_0 \cdots I_{i-1} I'_i$ and $L_1 = I''_i I_{i+1} \cdots I_{p-1}$. To restore Invariant 2.1 we adjust $O(1)$ chunks at the end of $L_0$ and the beginning of $L_1$, as described earlier.

**Performing** Cut($u, v$). Let $I_k$ and $I_l$ be the chunks containing $u$ and $v$ in the list $L = I_0 \cdots I_{p-1}$. Partition $L$ into parts $L = L_0(I'_k I''_k) L_1(I'_l I''_l) L_2(I_{p-1})$, where the parenthesized parts may not exist. If $I_k$ is not a Criterion 2 chunk then $I'_k$ is the prefix of $I_k$ preceding $u$ and $I''_k$ the suffix beginning with $u$. If $I_k$ is a Criterion 2 chunk then $I_k = (u)$ consists of one element and we include it in $L_1$. Chunk $I_l$ is handled similarly. If ID($I_{p-1}$) ≠ ⊥ then $I_{p-1}$ is included in $L_2$, otherwise it is separate.

Edges crossing the cut are of the following three types. See Figure 2.

(i) those incident to elements in $I''_k$ or $I'_l$,

(ii) those incident to elements in $I_{p-1}$, if ID($I_{p-1}$) = ⊥, and

(iii) those incident to elements in $L_1$.

The number of edges of types (i) and (ii) is $O(K)$; these edges are checked individually. Searching for edges of type (iii) reduces to an interval-sum query. Consider the length $M/K$ bit-vector

$$C = \bigvee_{I \in L_1} \text{Adj}_{\text{ID}(I)} \land \bigvee_{I \in L_0 \cup L_2} \text{Memb}_{\text{ID}(I)},$$

where $\lor$ and $\land$ are bit-wise Boolean operations. Note that $C(\text{ID}(I)) = 1$ precisely if $I$ is in $L_0L_2$ and there is some edge between $I$ and a chunk in $L_1$. Therefore, there is an edge crossing the cut if and only if $C$ is non-zero.

In order to implement Cut efficiently we impose on each list $L \in \mathcal{L}$ a (dynamic) balanced tree $T(L)$. Which tree structure we use is not relevant, so long as it supports all the standard operations in logarithmic time: splits and joins, insertions and deletions (corresponding to the creation and destruction and chunks) and interval-sum queries. For the sake of specificity we suggest using augmented (2,4)-trees. Each node $v \in T(L)$ stores two bit-vectors $\text{Adj}^v$ and $\text{Memb}^v$. When $v$ is

---

$^4$All internal nodes have between two and four children and all leaves are at the same distance from the root.
a leaf identified with chunk $I$, $\text{Adj}^u = \text{Adj}_{\text{ID}(I)}$ and $\text{Memb}^u = \text{Memb}_{\text{ID}(I)}$. In general, when $v$ is internal,

$$\text{Adj}^v = \bigvee_{w \text{ a leaf descend. of } v} \text{Adj}^w \quad \text{and} \quad \text{Memb}^v = \bigvee_{w \text{ a leaf descend. of } v} \text{Memb}^w.$$ 

If $C(\text{ID}(I)) = 1$ then we know there is an edge crossing the cut but still need to find one. We can locate a chunk $I'$ in $L_1$ such that $\text{Adj}_{\text{ID}(I')}(\text{ID}(I)) = 1$ as follows. The interval $L_1$ is covered by $O(\log n)$ internal nodes of $\mathcal{T}(L)$. Find one such node $v$ for which $\text{Adj}^v(\text{ID}(I)) = 1$ and walk from $v$ to a descendant leaf, in each step moving to a child $w$ for which $\text{Adj}^w(\text{ID}(I)) = 1$. We will eventually discover a leaf $I'$ for which $\text{Adj}_{\text{ID}(I')}(\text{ID}(I)) = 1$. If both $I$ and $I'$ are Criterion 2 chunks then we can immediately retrieve the edge between their singleton elements. If either chunk satisfies Criteria 1 or 3 (and therefore has mass $O(K)$), we can find an edge crossing the cut by scanning its $O(K)$ incident edges.

**Performing** \textsc{List}(u). Retrieve the chunk id $\text{ID}(u)$ in $O(1)$ time, then follow $O(\log(M/K))$ pointers from $u$’s chunk to the root of $\mathcal{T}(L)$, where $L$ is the list containing $u$. By complementing the tree with additional pointers we can maintain the property that all leaves have a path to their root with length $O(\log_{K/\log n}(M/K))$. As we will see shortly, this is $O(1)$ for any reasonable value of $K$, which depends on $m$ and $w$.

### 2.3 Running Time Analysis

**Periodic Rebuilding.** Recall that $\hat{m}$ is an upper bound on $m$, the current number of edges. When $m = \hat{m} - \sqrt{\hat{m}}$ or $m = \hat{m}/4 + \sqrt{\hat{m}}$ we rebuild the data structure over the next $\sqrt{\hat{m}}$ updates and set $\hat{m} \leftarrow 2\hat{m}$ or $\hat{m} \leftarrow \hat{m}/2$. At this point we also update $K \approx \sqrt{\hat{m}}$, which is a function of $M \overset{\text{def}}{=} 2\hat{m}$ and $w$. See below for the exact expression.

To prepare for the moment when $\hat{m}$ and $K$ are updated we need to replace the bit-vectors stored in the nodes of $\{\mathcal{T}(L)\}_{L \in \mathcal{L}}$ with longer or shorter ones, and increase or decrease the size of chunks to conform to Invariant 2.1. Over $\Theta(\sqrt{\hat{m}})$ updates we continue to reconfigure the chunks with respect to the old Invariant 2.1 and supplement nodes of $\{\mathcal{T}(L)\}_{L \in \mathcal{L}}$ with new bit-vectors, still maintaining the old ones. Over the following $\Theta(\sqrt{\hat{m}})$ updates we update all the chunks to conform to the new Invariant 2.1. During this rebuilding phase $m$ is roughly half the new $\hat{m}$, so there is always ample room in the new id-space $[M/K]$. After all chunks have been rebuilt to satisfy the new Invariant 2.1 we can destroy the old bit-vectors at nodes of $\{\mathcal{T}(L)\}_{L \in \mathcal{L}}$ over the next $\Theta(\sqrt{\hat{m}})$ updates.

**Inserting/Deleting Edges.** Since the nodes of $\{\mathcal{T}(L)\}_{L \in \mathcal{L}}$ carry two $M/K$-bit vectors, each stored in $[M/(Kw)]$ machine words, all “logarithmic” time operations on these trees actually take $O\left(\frac{M \log n}{Kw}\right)$ time. Creating or destroying a chunk $I$ in $L$ involves $O(K)$ modifications to the leaves of $\mathcal{T}(L)$, which may induce changes at all their ancestors. Note that only bit number $\text{ID}(I)$ of their bit-vectors can be affected. Since the tree has height at most $\log(M/K)$, the total number of affected ancestor nodes is $O(K \log(M/K^2))$. By the same argument, the time spent in $\text{Cut}(u, v)$ to search for an edge crossing the cut is also $O(K \log(M/K^2))$.

\footnote{In this interim period Invariant 2.1 is necessarily slightly weaker. We ensure that any chunks modified in the interim satisfy the new mass invariants. A chunk satisfies Criterion 3 if its mass is less than either the old or new $K$ and the neighboring chunks satisfy either the old or new Criterion 2.}
Inserting or deleting an edge may require $O(1)$ surgical operations (Splits and Joins), the creation and destruction of $O(1)$ chunks, and a single call to Cut. The total time is therefore $O\left(\frac{M \log n}{K w} + K \log \left(\frac{M}{K w}\right)\right)$, which is $O\left(\sqrt{M \cdot \frac{\log n}{w} \log \left(\frac{w}{\log n}\right)}\right)$ when $K = \sqrt{M \cdot \frac{\log n}{w \log(w/\log n)}}$. Using the sparsification method of Eppstein, Galil, Italiano, and Nissenzweig [EGIN97] this bound can be improved to $O\left(\sqrt{n \cdot \frac{\log n}{w} \log \left(\frac{w}{\log n}\right)}\right)$, assuming $w < n^{1-\Omega(1)}$. Since $w = \Omega(\log n)$, this bound is always $O(\sqrt{n})$ and it is $o(\sqrt{n})$ when $w = \omega(\log n)$.

### 3 Packing Sets

**Theorem 3.1.** There exists an algorithm in the word-RAM model that preprocesses a family of sets $F = \{S_1, \ldots, S_t\}$ in linear time so that given a query of two sets $S, S' \in F$ one can either determine the emptiness of $S \cap S'$ in $O\left(\frac{d \log^2 w}{w}\right)$ expected time, or list all of the elements in $S \cap S'$ in $O\left(\frac{d \log^2 w}{w} + k \log \log w\right)$ expected time and space where $d$ is a known upper bound on $\max_{S \in F} |S|$, $k = |S \cap S'|$, and $w$ is the size of the memory word. If $w = O(\log n)$ then the time costs are reduce to expected $O\left(\frac{d \log \log n}{\log n}\right)$ for emptiness and $O\left(\frac{d \log \log n}{\log n} + k\right)$ for listing. Furthermore, insertions and deletions of elements to a set $S \in F$ (without allowing $|S| > d$) can be supported in $O(\log w)$ expected time or $O(1)$ expected time when $w = O(\log n)$.

**Proof.** Every set $S \in F$ is split into $\ell$ buckets $B^S_1, \ldots, B^S_\ell$ where $\ell = \frac{d \log^2 w}{w}$. We use a pairwise independent hash function $\ell h$ to assign each element $e \in S$ into a bucket $B^S_{h(e)}$. The expected number of elements from a set $S$ in each bucket is $\frac{w}{\log w}$. We use a second hash function $h'$ which reduces the universe size to $w^2$. An $h'(e)$ value is represented with $2 \log w + 1$ bits, the extra control bit being necessary for certain manipulations described below. For each $S$ and $i$ we represent $h'(B^S_i)$ as a packed, sorted sequence of $h'$-values. In expectation each $h'(B^S_i)$ occupies $O(1)$ words, though some buckets may be significantly larger. Finally, for each bucket $B^S_i$ we maintain a lookup table that translates from $h'(e)$ to $e$. If there is more than one element that is hashed to $h'(e)$ then all such elements are maintained in the lookup table via a linked list.

Notice that $S \cap S' = \bigcup_{i=1}^\ell B^S_i \cap B^{S'}_i$. Thus, we can enumerate $S \cap S'$ by enumerating the intersections of all $B^S_i \cap B^{S'}_i$. Fix one such $i$. We first merge the packed sorted lists $h'(B^S_i)$ and $h'(B^{S'}_i)$. Albers and Hagerup [AH97] showed that two words of sorted numbers (separated by control bits) can be merged using Batcher’s algorithm in $O(\log w)$ time. Using this as a primitive we can merge the sorted lists $h'(B^S_i)$ and $h'(B^{S'}_i)$ in time $O(|B^S_i + B^{S'}_i|/(w/\log w))$. Let $C$ be the resulting list, with control bits set to 0. Our task is now to enumerate all numbers that appear twice (necessarily consecutively) in $C$. Let $C'$ be $C$ with control bits set to 1. We shift $C$ one field to the right ($2 \log w + 1$ bit positions) and subtract it from $C'$. Let $C''$ be the resulting list, with all control bits reset to 0. A field is zero in $C''$ iff it and its predecessor were identical, so the problem now is to enumerate zero fields. By repeated halving, we can distill each field to a single bit (0 for zero, 1 for non-zero) in $O(\log \log w)$ time and then take the complement of these bits (1 for zero, 0 for non-zero). We have now reduced the problem to reading off all the 1s in a $w$-bit word, which can be done in $O(\log \log w)$ time per 1 bit. See [BMM97]. For each repeated $h'$-value we lookup

---

6Regardless of how big $w$ is (relative to $n$), the running time for updates is always $\Omega(\log n)$ without sparsification or $\Omega(\log^2 n)$ with sparsification.

7We emphasize that multiplication is not needed for hashing as one can use tabulation hashing. See [PTT16].

8The control bits stop carries from crossing field boundaries.

9The $\log \log w$ factor can be removed with a variety of assumptions. If $w = O(\log n)$ then we can do $O(1)$-time
all elements in \( B^S_i \) and \( B^{S'}_i \) with that value and report any occurring in both sets. Every unit of time spent in this step corresponds to an element in the intersection or a false positive.

The cost of intersecting buckets \( B^S_i \) and \( B^{S'}_i \) is \( O(1 + \left\lfloor \frac{|B^S_i|}{w/\log w} \right\rfloor + \left\lfloor \frac{|B^{S'}_i|}{w/\log w} \right\rfloor) \log w + (|B^S_i \cap B^{S'}_i| + f_i) \log log w \) where \( f_i \) is the number of false positives. The expected value of \( f_i \) is \( o(1) \) since the expected sizes of \( B^S_i \) and \( B^{S'}_i \) are \( w/\log w \) and for \( e \in B^S_i, e' \in B^{S'}_i \), \( Pr(h'(e) = h'(e')) = 1/w^2 \). Thus, the expected runtime for a query is

\[
\sum_{i=1}^{\ell} O(1 + \left\lfloor \frac{|B^S_i|}{w/\log w} \right\rfloor + \left\lfloor \frac{|B^{S'}_i|}{w/\log w} \right\rfloor) \log w + (|B^S_i \cap B^{S'}_i| + f_i) \log log w \\
= O(\ell \log w + |S \cap S'| \log log w) \\
= O\left(\frac{d \log^2 w}{w} + k \log \log w\right).
\]

It is straightforward to implement insertions and deletions in \( O(\log w) \) time. When \( w = O(\log n) \) many operations can be performed by table lookup, in \( O(1) \) time. For example, we can sort a packed word of hash values in \( O(1) \) time, or insert/delete a hash value from a word in \( O(1) \) time, or enumerate all 1s in a word in \( O(1) \) time per 1.

\[\square\]

4 A Faster Triangle Enumeration Algorithm

**Theorem 4.1.** Given a graph \( G = (V, E) \) with \( m = |E| \) edges and \( n = |V| \) vertices and arboricity \( \alpha \), all \( t \) triangles can be enumerated in \( O(m + \frac{m\alpha}{\log n/\log \log n} + t \log \log w) \) expected time or in \( O(m + \frac{m\alpha}{\log n/\log \log n} + t) \) expected time if \( w = O(\log n) \).

**Proof.** We will make use of the data structure in Theorem 3.1. To do this we first find an acyclic orientation of \( E \) in which the out-degree of any vertex is \( O(\alpha) \). Such an orientation can be found in linear time using the peeling algorithm of Chiba and Nishizeki [CN85]. Begin by preprocessing the set family \( F = \{\Gamma^+(u) \mid u \in V\} \), where all sets have size \( O(\alpha) \). For each edge \((u, v)\), enumerate all elements in the intersection \( \Gamma^+(u) \cap \Gamma^+(v) \). For each vertex \( w \) in the intersection output the triangle \( \{u, v, w\} \). Since the orientation is acyclic, every triangle is output exactly once. There are \( m \) set intersection queries, each taking \( O(\alpha/\max\{\frac{w}{\log^2 w}, \frac{\log n}{\log \log n}\}) \) time, aside from the cost of reporting the output, which is \( O(\log \log w) \) per element or \( O(1) \) if \( w = O(\log n) \).

\[\square\]

5 Emptiness Queries

Each set \( S \in F \) maintains its elements in a lookup table using a perfect dynamic hash function. So the cost of inserting a new element into \( S \), deleting an element from \( S \), or determining whether some element \( x \) is in \( S \) is expected \( O(1) \) time. Let \( N = \sum_{S \in F} |S| \). We make the standard assumption that \( N \) is always at least \( N'/2 \) and at most \( 2N' \) for some natural number \( N' \). Standard rebuilding de-amortization techniques are used if this is not the case.

---

lookup into \( o(n) \)-size precomputed tables. If unit-time multiplication is available we can use the most-significant-bit algorithm of [FW93]. The most reasonable assumption is that there is a unit-time operation to convert a number from integer to floating point representation, which produces the MSB as a byproduct.
The Structure. We say a set \( S \) is large if at some point \( |S| > 2\sqrt{N} \), and since the last time \( S \) was at least that large, its size was never less than \( \sqrt{N} \). If a set is not large, and is size is at least \( \sqrt{N} \) then we say it is medium. If a set is neither large nor medium then it is small. Notice that the size of a small set is less than \( \sqrt{N} = O(\sqrt{N}) \). Let \( S \subseteq F \) be the sub-family of large and medium sets, and let \( \ell = |S| \). Notice that \( \ell \leq \sqrt{N} \). For each set \( S \in L \) we maintain a unique integer \( 1 \leq i_S \leq \ell \), and an intersection-size integer array \( A_S \) of size \( \ell \) such that \( A_S[i_S] = |S \cap S'| \). We will sometimes add or delete the last slot at the end of an intersection-set array. Using standard techniques these operations can be done in worst-case \( O(1) \) time. Due to the nature of our algorithm we cannot guarantee that all of the intersection-size arrays will always be fully updated. However, we will guarantee the following invariant.

Invariant 5.1. For every two large sets \( S \) and \( S' \), \( A_S[i_S] \) and \( A_{S'}[i_S] \) are correctly updated.

Query. For two sets \( S, S' \in F \) where either \( S \) or \( S' \) is not large, say \( S \), we can check if they intersect by scanning the elements in \( S \) and for each one we can check if it is in \( S' \) via the lookup table. The time cost is \( O(|S|) = O(\sqrt{N}) \). If both sets are large, then we look at \( A_S[i_S] \) to see if it is zero or more, which by Invariant 5.1 is correct. This takes \( O(1) \) time.

Insertions. When inserting a new element \( x \) into \( S \), we first need to update the lookup table of \( S \) to include \( x \). Next, if \( S \) was small and remained small then no additional work is done. Otherwise, for each \( S' \in L \) we must update the size of \( S \cap S' \) in the appropriate intersection-size arrays. This is done directly in \( O(\sqrt{N}) \) time by checking if \( x \) is in \( S' \) for each \( S' \). We briefly recall, as mentioned above, that it is possible that some of the intersection-size arrays will not be fully updated, and so updating the size of intersections may be impossible just from checking if \( x \) is in \( S' \) for those appropriate locations. Nevertheless, as explained soon, Invariant 5.1 will still hold, which suffices for the correctness of the algorithm.

The more challenging case is when \( S \) becomes medium. If this happens we need to increase \( \ell \) by 1, assign \( i_S \) to be the new \( \ell \), construct \( A_S \) in \( O(\sqrt{N}) \) time, and for each \( S' \in L \) we need to increase the size of \( A_{S'} \) by 1 in \( O(\sqrt{N}) \) time and compute \( A_{S'}[i_S] \). This entire process is dominated by the need to compute \( |S \cap S'| \) for each \( S' \in L \), taking a total time of \( O(\sum_{S' \in L} |S|) \) which could be as large as \( O(N) \) and is too costly. However, this work can be spread over the following \( \sqrt{N} \) insertions made into \( S \) until \( S \) become large. This is done as follows. When \( S \) becomes medium it copies \( L \) into \( L_S \). Next, for every insertion into \( S \) we compute the value of \( O(1) \) locations in \( A_S \) by computing the intersection size of \( S \) and each of \( O(1) \) sets from \( L_S \) in \( O(\sqrt{N}) \) time. By the time \( S \) becomes large we will have correctly computed the indicator in \( A_S \) for all \( \sqrt{N} \) sets. It is possible that between the time \( S \) became medium to the time \( S \) became large, there may have been other sets such as \( S' \) which became medium and perhaps even large, but \( S' \notin L_S \). Notice that in such a case \( S \in L_{S} \) and so it is guaranteed that by the time both \( S \) and \( S' \) are large, the indicators \( A_S[i_S] \) and \( A_{S'}[i_S] \) are correctly updated, and so Invariant 5.1 is maintained. Thus the total cost of performing an insertion is \( O(\sqrt{N}) \) expected time.

Deletions. When deleting an element \( x \) from \( S \), we first need to update the lookup table of \( S \) to remove \( x \) in \( O(1) \) expected time. If \( S \) was small and remained small then no additional work is done. If \( S \) was in \( L \) then we must scan all of the \( S' \in L \) and check if \( x \) is in \( S' \) in order to update the appropriate locations in the intersection-size arrays. This takes \( O(\sqrt{N}) \) time.

If \( S \) was medium and now became small, we need to decrease \( \ell \) by 1, remove the assignment to \( i_S \) to be the new \( \ell \), delete \( A_S \), and for each \( S' \in L \) we need to remove \( A_{S'}[i_S] \) and decrease the
size of \( A_{S'} \) by 1. To facilitate these requirements, let \( \hat{S} \in L \) be such that \( i_{\hat{S}} = \ell \). If \( \hat{S} \neq S' \) then we swap \( i_{\hat{S}} \) with \( i_S \) in \( O(\sqrt{N}) \) time by swapping the intersection sizes in all the arrays. Now we are guaranteed that \( i_S = \ell \), so we delete the last location from each of the arrays in total \( O(\sqrt{N}) \) time. In addition, in order to accommodate the update process of medium sized sets, for each medium set \( S' \) we must remove \( S \) from \( L_{S'} \) if it was in there.

**Theorem 5.1.** There exists an algorithm that maintains a family \( F \) of dynamic sets using \( O(N) \) space where each update costs \( O(\sqrt{N}) \) expected time, and each emptiness query costs \( O(\sqrt{N}) \) expected time.

**6 Finding Proofs Efficiently**

**Theorem 6.1.** If there exists an algorithm \( A \) that maintains a family of incremental sets \( F \) so that set intersection proof queries can be answered in \( O\left(\frac{d}{\tau_q}\right) \) expected time and an element can be added to a set in \( O(\tau_q) \) expected time, where \( d \) is an a-priori upper bound on the size of the sets in \( F \), \( \tau_q > 0 \) is some speed-up factor, and \( k \) is the size of the output, then there exists an algorithm in the word-RAM model that maintains a family \( F \) of incremental sets using \( O(N) \) space where each insertion and proof query costs \( O\left(\frac{N}{\tau_q}\right) \) expected time.

**Proof.** In our context, we say that a set is large if its size is at least \( \sqrt{N\tau_q} \), and is medium if its size is between \( \sqrt{N\tau_q} \) and \( \sqrt{N\tau_q} \). Similar to the algorithm of Theorem 5.1 we maintain for each medium and large set \( S \) a proof array \( P_S \) such that for any large set \( S' \) we have that \( P_S[i_{S'}] \) is either an element (proof) in the intersection of \( S \) and \( S' \), or null if no such element exists. This works in our setting since only insertions are allowed so once we have a proof that two sets intersect it will never change. Since there are at most \( \sqrt{N\tau_q} \) large sets and at most \( \sqrt{N\tau_q} \) medium sets, the space usage is \( O(N) \).

We use the algorithm for \( A \) to maintain all of the medium and small sets, with \( d = \sqrt{N\tau_q} \). So a set intersection query between two such sets will cost \( O\left(\frac{\sqrt{N}}{\tau_q}\right) \) expected time. In addition, each large set \( S \) maintains a stash of some elements that were recently inserted into it. This stash is a secondary set of size at most \( \sqrt{N\tau_q} \) which is also maintained with algorithm \( A \). If a query is between \( S_1 \) and \( S_2 \) and \( S_1 \) is large, then: (1) if \( S_2 \) is small we lookup each element in \( S_2 \) to see if it is in \( S_1 \), (2) if \( S_2 \) is medium then we use the proof array to see if there is a proof of an intersection, and if there is none then we intersect \( S_2 \) with the stash of \( S_1 \) using algorithm \( A \), and (3) if \( S_2 \) is large we use the proof array to see if there is a proof of an intersection. Notice that in the last case there is no need to intersect the stash of \( S_2 \) with the stash of \( S_1 \) as the proof is in proof array if it exists. In any case, the cost of a query is \( O\left(\frac{\sqrt{N}}{\tau_q}\right) \) expected time.

**Insertion.** When inserting an element \( x \) into \( S \), if \( S \) is small then we do nothing. If \( S \) is medium then we add \( x \) to the structure of \( S \) in algorithm \( A \). If \( S \) is large then we add \( x \) to the stash of \( S \) and check for every other large set if \( x \) is in that set, updating the table accordingly. If \( S \) became medium then we add it to the structure of algorithm \( A \). Since the size of \( S \) is \( O\left(\frac{N}{\tau_q}\right) \) this takes \( O\left(\frac{\sqrt{N}}{\tau_q}\right) \) expected time. If \( S \) became large then we no longer need algorithm \( A \) to maintain it (however there is no need to remove the part of \( S \) that is already in algorithm \( A \)). The affect of
growth of the proof tables is treated using the same techniques as in Theorem 5.1 and so we omit their description. This will cost $O(\sqrt{N/\tau_q} + \tau_u)$ expected time.

Finally, if $S$ is large then the stash of $S$ may overflow. To overcome this, we need to combine $S$ with its stash, while updating the the proof arrays. We describe an amortized solution for this process, which a direct lazy approach will deamortize. When combining $S$ with its stash, we need to update the proof arrays for set intersection proofs between medium sets and $S$ as it is possible that the proof is in the stash. To do this, we directly scan all of the medium sets and check if a new proof can be obtained from the stash. The number of medium sets is $O(\sqrt{N/\tau_q})$ and the cost of each intersection will be $O(\sqrt{N/\tau_q})$ one for a total of $O(N)$ time. Since this operation only takes place once every $\Omega(\sqrt{N/\tau_q})$ insertions into $S$ the amortized cost is $O(\sqrt{N/\tau_q})$ time.

Combining Theorem 3.1 with Theorem 6.1 we obtain the following.

**Corollary 6.2.** There exists an algorithm in the word-RAM model that maintains a family $F$ of incremental sets using $O(N)$ space where each insertion costs $O(\sqrt{N/w/\log^* w} + \log w)$ expected time and a proof query costs $O(\sqrt{N/w/\log^* w})$ expected time.
References

[AH97] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. Inf. Comput., 136(1):25–51, 1997.

[BBMM14] K. Buchin, M. Buchin, W. Meulemans, and W. Mulzer. Four Soviets walk the dog – with an application to Alt’s conjecture. In Proceedings 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 1399–1413, 2014.

[BDP08] I. Baran, E. D. Demaine, and M. Pătraşcu. Subquadratic algorithms for 3SUM. Algorithmica, 50(4):584–596, 2008.

[BK02] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In Proceedings 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 390–399, 2002.

[BMM97] A. Brodnik, P. B. Miltersen, and J. I. Munro. Trans-dichotomous algorithms without multiplication – some upper and lower bounds. In Proceedings 5th Int’l Workshop on Algorithms and Data Structures (WADS), volume 1272 of Lecture Notes in Computer Science, pages 426–439. Springer Berlin Heidelberg, 1997.

[BPP07] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In Algorithms and Computation, 18th International Symposium, ISAAC 2007, pages 739–750, 2007.

[BPWZ14] A. Bjorklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. In Proceedings 41st Int’l Colloquium on Automata, Languages, and Programming (ICALP), page ?, 2014.

[BY04] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In Combinatorial Pattern Matching, 15th Annual Symposium, CPM, pages 400–408, 2004.

[Cha08] T. M. Chan. All-pairs shortest paths with real weights in $o(n^3/\log n)$ time. Algorithmica, 50(2):236–243, 2008.

[Cha10] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. SIAM J. Comput., 39(5):2075–2089, 2010.

[Cha12] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. ACM Transactions on Algorithms, 8(4):34, 2012.

[Cha13] T. M. Chan. The art of shaving logs. In Proceedings 13th Int’l Symposium on Algorithms and Data Structures (WADS), volume 8037 of Lecture Notes in Computer Science, pages 231–231. Springer, 2013.

[CN85] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. SIAM J. Comput., 14(1):210–223, 1985.

[CP10] H. Cohen and E. Porat. Fast set intersection and two-patterns matching. Theor. Comput. Sci., 411(40-42):3795–3800, 2010.

[DLOM00] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, pages 743–752, 2000.

[EGIN97] 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.

[Fre85] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. SIAM J. Comput., 14(4):781–798, 1985.

[FS89] 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.
[FW93] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci., 47(3):424–436, 1993.

[GP14] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In Proceedings 55th IEEE Symposium on Foundations of Computer Science (FOCS), 2014. Full manuscript available as arXiv:1404.0799.

[HdT01] 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.

[HF98] M. R. Henzinger and M. L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. Algorithmica, 22(3):351–362, 1998.

[HK99] M. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J. ACM, 46(4):502–516, 1999.

[HT97] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. J. Random Structures and Alg., 11(4):369–379, 1997.

[IR78] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. SIAM J. Comput., 7(4):413–423, 1978.

[KKM13] 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.

[MP80] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. J. Comput. Syst. Sci., 20(1):18–31, 1980.

[MSVT94] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. Theoretical Computer Science, 130(1):203–236, 1994.

[Mye92] G. Myers. A Four Russians algorithm for regular expression pattern matching. J. ACM, 39(2):432–448, 1992.

[PD06] M. Pătrașcu and E. Demaine. Logarithmic lower bounds in the cell-probe model. SIAM J. Comput., 35(4):932–963, 2006.

[PT11a] 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.

[PT11b] M. Patrascu and M. Thorup. The power of simple tabulation hashing. In STOC, pages 1–10, 2011.

[Tho00] M. Thorup. Near-optimal fully-dynamic graph connectivity. In Proceedings 32nd ACM Symposium on Theory of Computing (STOC), pages 343–350, 2000.

[WN13] 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.
A Emptiness Queries with Time/Memory Tradeoff

We assume without loss of generality that all of the sets are from an ordered integer universe $U = \{1, 2, \ldots, u\}$, as one can always consider the binary presentation of the keys differentiating the elements as representation of integers. Nevertheless, our results will not depend on $u$. Each set $S \in F$ maintains its elements in a lookup table using a perfect dynamic hash function. So the cost of inserting a new element into $S$, deleting an element from $S$, or determining whether some element $x$ is in $S$ is expected $O(1)$ time. Let $N = \sum_{S \in F} |S|$. We make the standard assumption that $N$ is always at least $N'/2$ and at most $2N'$ for some natural number $N'$. Rebuilding de-amortization techniques are used if this is not the case.

The Structure. We say a set $S$ is large if at some point $|S| > \frac{2N'}{\sqrt{M}}$, and since the last time $S$ was at least that large, its size was never less than $\frac{N'}{\sqrt{M}}$. If a set is not large, and is size is at least $\frac{N'}{\sqrt{M}}$, then we say it is medium. If a set is neither large nor medium then it is small. Notice that the size of a small set is less than $\frac{N'}{\sqrt{M}} = O(\frac{N}{\sqrt{M}})$. Let $L \subseteq F$ be the sub-family of large and medium sets, and let $\ell = |L|$. Notice that $\ell \leq \sqrt{M}$. For each set $S \in L$ we maintain a unique integer $1 \leq i_S \leq \ell$, and an intersection-size integer array $A_S$ of size $\ell$ such that $A_S[i_S] = |S \cap S'|$. We will sometimes add or delete the last slot at the end of an intersection-set array. Using standard techniques these operations can be done in worst-case $O(1)$ time. Due to the nature of our algorithm we cannot guarantee that all of the intersection-size arrays will always be fully updated. However, we will guarantee the following invariant.

Invariant A.1. For every two large sets $S$ and $S'$, $A_S[i_S']$ and $A_S'[i_S]$ are correctly updated.

Query. For two sets $S, S' \in F$ where either $S$ or $S'$ is not large, say $S$, we can check if they intersect by scanning the elements in $S$ and for each one we can check if it is in $S'$ via the lookup table. The time cost is $O(|S|) = O(\frac{N}{\sqrt{M}})$. If both sets are large, then we look at $A_S[i_S']$ to see if it is zero or more, which by Invariant A.1 is correct. This takes $O(1)$ time.

Insertions. When inserting a new element $x$ into $S$, we first need to update the lookup table of $S$ to include $x$. Next, if $S$ was small and remained small then no additional work is done. Otherwise, for each $S' \in L$ we must update the size of $S \cap S'$ in the appropriate intersection-size arrays. This is done directly in $O(\sqrt{M})$ time by checking if $x$ is in $S'$ for each $S'$. We briefly recall, as mentioned above, that it is possible that some of the intersection-size arrays will not be fully updated, and so updating the size of intersections may be impossible just from checking if $x$ is in $S'$ for those appropriate locations. Nevertheless, as explained soon, Invariant A.1 will still hold, which suffices for the correctness of the algorithm.

The more challenging case is when $S$ becomes medium. If this happens we need to increase $\ell$ by 1, assign $i_S$ to be the new $\ell$, construct $A_S$ in $O(\sqrt{M})$ time, and for each $S' \in L$ we need to increase the size of $A_{S'}$ by 1 in $O(\sqrt{M})$ time and compute $A_S[i_S]$. This entire process is dominated by the need to compute $|S \cap S'|$ for each $S' \in L$, taking a total time of $O(\sum_{S' \in L} |S|)$ which could be as large as $O(N)$ and is too costly. However, this work can be spread over the following $\frac{N'}{\sqrt{M}}$ insertions made into $S$ until $S$ become large. This is done as follows. When $S$ becomes medium it copies $L$ into $L_S$. Next, for every insertion into $S$ we compute the value of $\lceil \frac{M}{N'} \rceil$ locations in $A_S$ by computing the intersection size of $S$ and each of $\lceil \frac{M}{N'} \rceil$ sets from $L_S$ in $O(\frac{M}{N'} \cdot \sqrt{M}) = O(\sqrt{M})$ time. By the time $S$ becomes large we will have correctly computed the indicator in $A_S$ for all
### Worst Case Data Structures

| Ref.       | Update Time | Query Time | Notes                                      |
|------------|-------------|------------|--------------------------------------------|
| [Fre85]    | $O(\sqrt{m})$ | $O(1)$     |                                            |
| [EGIN97, Fre85] | $O(\sqrt{n})$ | $O(1)$     | [Fre85] + sparsification [EGIN97].         |
| [KKM13]    | $O(c\log^5 n)$ | $O\left(\frac{\log n}{\log \log n}\right)$ | Rand. Monte Carlo; no witness.              |
| new        | $O\left(\sqrt{n} \cdot \frac{\log n}{w} \log\left(\frac{w}{\log n}\right)\right)$ | $O(1)$    | Assumes word size $w < n^{1-\epsilon}$. |

### Amortized Data Structures

| Ref.        | Amort. Update | W.C. Query | Notes                                      |
|-------------|---------------|------------|--------------------------------------------|
| [HK99]      | $O(\log^3 n)$ | $O\left(\frac{\log n}{\log \log n}\right)$ | Randomized Las Vegas.                      |
| [HT97]      | $O(\log^2 n)$ | $O\left(\frac{\log n}{\log \log n}\right)$ | Randomized Las Vegas.                      |
| [HdT01]     | $O(\log^2 n)$ | $O\left(\frac{\log n}{\log \log n}\right)$ |                                            |
| [Tho00]     | $O(\log n (\log \log n)^3)$ | $O\left(\frac{\log n}{\log \log \log n}\right)$ | Randomized Las Vegas.                      |
| [WN13]      | $O(\log^2 n/\log \log n)$ | $O\left(\frac{\log n}{\log \log \log n}\right)$ |                                            |

### Amort./Worst Case Lower Bounds

| Ref.        | Update Time $t_u$ | Query Time $t_q$ | Notes                                      |
|-------------|-------------------|------------------|--------------------------------------------|
| [FS89, HF98] | $t_u = \Omega\left(\frac{\log n}{\log \log n + \log t_u}\right)$ | $t_q = \Omega\left(\frac{\log n}{\log(t_u/t_q)}\right)$ | Implies $\max\{t_u, t_q\} = \Omega(\log n)$. |
| [PD06]      | $o(\log n)$      | $\Omega\left(1-\Theta(1)\right)$ |                                            |

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

16
\[ \frac{M}{N} \cdot \frac{N'}{M} \geq \sqrt{M} \] sets. It is possible that between the time \( S \) became medium to the time \( S \) became large, there may have been other sets such as \( S' \) which became medium and perhaps even large, but \( S' \not\in L_S \). Notice that in such a case \( S \in L_S \) and so it is guaranteed that by the time both \( S \) and \( S' \) are large, the indicators \( A_S[i_S] \) and \( A_{S'}[i_{S'}] \) are correctly updated, and so Invariant A.1 is maintained. Thus the total cost of performing an insertion is \( O(\frac{N}{\sqrt{M}} + \sqrt{M}) = O(\sqrt{M}) \) expected time.

**Deletions.** When deleting an element \( x \) from \( S \), we first need to update the lookup table of \( S \) to remove \( x \) in \( O(1) \) expected time. If \( S \) was small and remained small then no additional work is done. If \( S \) was in \( L \) then we must scan all of the \( S' \in L \) and check if \( x \) is in \( S' \) in order to update the appropriate locations in the intersection-size arrays. This takes \( O(\sqrt{M}) \) expected time.

If \( S \) was medium and now became small, we need to decrease \( \ell \) by 1, remove the assignment to \( i_S \) to be the new \( \ell \), delete \( A_S \), and for each \( S' \in L \) we need to remove \( A_{S'}[i_S] \) and decrease the size of \( A_{S'} \) by 1. To facilitate these requirements, let \( \hat{S} \in L \) be such that \( i_{\hat{S}} = \ell \). If \( \hat{S} \neq S' \) then we swap \( i_{\hat{S}} \) with \( i_S \) in \( O(\sqrt{M}) \) time by swapping the intersection sizes in all the arrays. Now we are guaranteed that \( i_S = \ell \), so we delete the last location from each of the arrays in total \( O(\sqrt{M}) \) time. In addition, in order to accommodate the update process of medium sized sets, for each medium set \( S' \) we must remove \( S \) from \( L_{S'} \) if it was in there.

**Theorem A.1.** There exists an algorithm that maintains a family \( F \) of dynamic sets using \( O(M) \) space where each update costs \( O(\sqrt{M}) \) expected time, and each emptiness query costs \( O(\frac{N}{\sqrt{M}}) \) expected time.

**B Fully Dynamic Set Intersection**

In order to avoid having to deal with the universe size as insertions of new elements take place we assign in advance integers in the range of \([2N']\). Each element in \( \bigcup_{S \in F} S \) is assigned an integer in this range. When a new element not appearing in \( \bigcup_{S \in F} S \) arrives, it is assigned to the smallest available integer, and that integer is used as its key. When keys are deleted (no longer in use), we do not remove their assignment, and instead, we conduct a standard rebuilding technique in order to reassign the elements. Finally, we use a random permutation so we may assume that the elements are uniformly spread in the universe.

**The structure.** Consider the following binary tree \( T \) of height \( O(\log N') \) where each vertex \( v \) covers some range from \( U \), denoted by \([\alpha_v, \beta_v]\), such that the range of the root covers all of \( U \), and the left (right) child of \( v \) covers the first (second) half of \([\alpha_v, \beta_v]\). A vertex at depth \( i \) covers \( \frac{[2N']}{2^i} \) elements of \( U \). For a vertex \( v \) let \( S^v = S \cap [\alpha_v, \beta_v] \). Let \( N_v = \sum_{S \in F} |S^v| \). Let \( M_v = \frac{N_v M}{N} \).

We say a set \( S \) is \( v \)-large if at some point \( |S^v| > \frac{2N}{\sqrt{M}} \), and since the last time \( S^v \) was at least that large, its size was never less than \( \frac{N_v}{\sqrt{M}} \). If \( S \) is not \( v \)-large, and \( |S^v| \geq \frac{N_v}{\sqrt{M}} \) then we say it is \( v \)-medium. If \( S \) is neither \( v \)-large nor \( v \)-medium then it is \( v \)-small. Notice that if \( S \) is \( v \)-small then \( |S^v| < \frac{N_v}{\sqrt{M}} = O(\frac{N_v}{\sqrt{M}}) \).

Each vertex \( v \in T \) with children \( v_0 \) and \( v_1 \) maintains a structure for emptiness queries as in Section A with size parameter \( N_v \) and \( M_v \) on the family \( F^v = \{ S^v : S \in F \} \). In addition, we add auxiliary data to the intersection-size arrays as follows. For sets \( S_1, S_2 \in F \) the set of all vertices in which \( S_1 \) and \( S_2 \) intersect under them defines a tree \( T' \). This tree has some branching vertices which
have 2 children, some non-branching internal vertices with only 1 child, and some leaves. Consider the vertices in \( T \) for which \( S_1 \) and \( S_2 \) are relatively large, and then consider only the connected component of these vertices which includes \( r \), assuming such a component exists. This connected component defines a tree \( \hat{T} \) on which we will perform the computation during query time. To assist in a fast traversal of \( \hat{T} \) we maintain shortcut pointers for every two sets \( S_1, S_2 \in F \) and for every vertex \( v \in T \) such that \( S_1 \) and \( S_2 \) are \( v \)-large. To this end, we say \( v \) is a branching-\((S_1, S_2)\)-vertex if both \( S_1^0 \cap S_2^0 \neq \emptyset \) and \( S_1^i \cap S_2^i \neq \emptyset \).

Consider the path starting from the left (right) child of \( v \) and ending at the first descendant \( v' \) of \( v \) such that:(1) \( S_1 \) and \( S_2 \) are relatively large to all of the vertices on the path, (2) \( S_1^{v'} \cap S_2^{v'} \neq \emptyset \), and (3) either \( v' \) is a branching-\((S_1, S_2)\)-vertex or one of the sets \( S_1 \) and \( S_2 \) is not \( v' \)-large. The left (right) shortcut pointer of \( v \) will point to \( v' \). Notice that the shortcut pointers are maintained for every vertex \( v \) even if on the path from \( r \) to \( v \) there are some vertices for which either \( S_1 \) or \( S_2 \) are not relatively large. Also notice that using these pointers it is straightforward to check in \( O(1) \) time if \( S_1^0 \cap S_2^0 \) and \( S_1^i \cap S_2^i \) are empty or not.

The space complexity of the structure is as follows. Each vertex \( v \) uses \( O(M_v) \) words of memory which is \( O(MN_v/N') \). So the memory usage is \( \sum_v M_v = O(M \log N) \) since in each level of \( T \) the sum of all \( M_v \) for the vertices in that level is \( O(M) \), and there are \( O(\log N) \) levels.

**Reporting queries.** For a reporting query on \( S_1 \) and \( S_2 \), if \( op = 0 \) then either the emptiness test at the root will conclude in \( O(1) \) time, or we spend \( O\left(\frac{N'}{\sqrt{M'}}\right) = O\left(\frac{N}{\sqrt{M}}\right) \) time. Otherwise, we recursively examine vertices \( v \) in \( T \) starting with the root \( r \). If both \( S_1 \) and \( S_2 \) are \( v \)-large and \( S_1^i \cap S_2^i \neq \emptyset \), then we continue recursively to the vertices pointed to by the appropriate shortcut pointers. If either \( S_1 \) or \( S_2 \) is not \( v \)-large then we wish to output all of the elements in the intersection of \( S_1^i \) and \( S_2^i \), called a local reporting procedure. To do this, we check for each element in the smaller set if it is contained within the bigger set using the lookup table which takes \( O\left(\frac{N}{\sqrt{M'}}\right) \) time.

**Lemma B.1.** If \( \sum_{i=1}^{t} x_i \leq k \) then \( \sum_{i=1}^{t} \sqrt{x_i} \leq \sqrt{k \cdot t} \).

*Proof.* Since \( \sum_{i=1}^{t} \sqrt{x_i} \) is maximized whenever all the \( x_i \) are equal, we have that \( \sum_{i=1}^{t} \sqrt{x_i} \leq t \sqrt{\frac{k}{t}} = \sqrt{kt} \).

**Lemma B.2.** The time to answer a reporting query is \( O\left(\frac{N}{\sqrt{M}} \sqrt{op + 1}\right) \) where \( op \) is the size of the output.

*Proof.* Assume at first that \( op > 0 \). The algorithm encounters two types of vertices. The first type are vertices \( v \) for which both \( S_1 \) and \( S_2 \) are \( v \)-large, and the second type are vertices \( v \) for which either \( S_1 \) or \( S_2 \) is not \( v \)-large. Each vertex of the first type performs \( O(1) \) work, and the number of such vertices is at most the number of vertices of the second type, due to the branching nature of the shortcut pointers. For vertices of the second type, the intersection of \( S_1 \) and \( S_2 \) must both be non-empty relative to such vertices and so the \( O\left(\frac{N'}{\sqrt{M'}}\right) \) time cost can be charged to at least one element in the output. Denote the vertices of the second type by \( v_1, v_2, \ldots, v_t \). Notice that \( t \leq op \) as each \( v_i \) contains at least one element from the intersection, and that \( \sum_i N_{v_i} < 2N' \) since the vertices are not ancestors of each other. Therefore, due to Lemma [B.1], the total time cost is

\[
\sum_{i} \frac{N'_{v_i}}{\sqrt{M_{v_i}}} = \sum_{i} \frac{N'_{v_i} \sqrt{N'}}{\sqrt{M N_{v_i}}} = \sqrt{\frac{N'}{M}} \sum_{i} \sqrt{N'_{v_i}} \leq \sqrt{\frac{N'}{M}} \sqrt{2N' \sqrt{t}} \leq O\left(\frac{N \sqrt{op}}{\sqrt{M}}\right).
\]

\( \square \)
Proof queries. A proof query is answered by traversing down $T$ using shortcut pointers, but instead of recursively looking at both shortcut pointers for each vertex, we only consider one. Thus the total time it takes until we reach a vertex $v$ for which either $S_1$ or $S_2$ is not $v$-large is $O(\log N)$. Next, we use the hash function to find an element in the intersection in $O\left(\frac{N}{\sqrt{M}}\right)$ time, for a total of $O(\log N + \frac{N}{\sqrt{M}})$ time to answer a proof query.

Insertions and Deletions. When inserting a new element $x$ into $S_1$, we first locate the leaf $\ell$ of $T$ which covers $x$. Next, we update our structure on the path from $\ell$ to $r$ as follows. Starting from $\ell$, for each vertex $v$ on the path we insert $x$ into $S^v_1$. This incurs a cost of $\sqrt{M_v}$ for updating the emptiness query structure at $v$. If there exists some set $S_2$ such that $|S^v_1 \cap S^v_2|$ becomes non-zero, then we may need to update some shortcut pointers on the path from $\ell$ to $r$ relative to $S_1$ and $S_2$. Being that such a set $S_2$ must be large, the number of such sets is at most $\frac{N_\ell}{\sqrt{2M}}$. To analyze the expected running time of an insertion notice that since the elements in the universe are randomly distributed, the expected value of $N_v$ and $M_v$ for a vertex $v$ at depth $i$ are $\frac{N}{2^i}$ and $\frac{M}{2^i}$ respectively. So the number of $v$-large sets is at most $\frac{N_\ell}{\sqrt{2M}} = \frac{N}{\sqrt{2M}}$. The expected time costs of updating the emptiness structure is at most $\sum_{i=0}^{\infty} \log N \cdot \frac{\sqrt{2M}}{N_\ell} = O\left(\frac{N}{\sqrt{M}}\right)$. The same analysis holds for the shortcut pointer.

The deletion process is exactly the reverse of the insertions process, and also costs $O\left(\frac{N}{\sqrt{M}}\right)$ expected time.

Finally, being that the space usage is $O(M \log N)$, but we are interested in results that depend directly on $M$, we substitute $M/\log N$ for $M$ and obtain the following.

**Theorem B.3.** There exists an algorithm that maintains a family $F$ of dynamic sets using $O(M)$ space where each update costs $O(\sqrt{M \log N})$ expected time, each report query costs $O\left(\frac{N\log N}{\sqrt{M}} \sqrt{op + 1}\right)$ time, and each proof query costs $O\left(\frac{N\log N}{\sqrt{M}} + \log N\right)$ expected time.