# Deterministic and Low-Span Work-Efficient Parallel Batch-Dynamic Trees

Daniel Anderson
Carnegie Mellon University
Pittsburgh, USA
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, USA
guyb@cs.cmu.edu

## ABSTRACT

Dynamic trees are a well-studied and fundamental building block of dynamic graph algorithms dating back to the seminal work of Sleator and Tarjan [*STOC'81*, (1981), pp. 114-122]. The problem is to maintain a tree subject to online edge insertions and deletions while answering queries about the tree, such as the heaviest weight on a path, etc. In the parallel batch-dynamic setting, the goal is to process batches of edge updates work efficiently in low (polylog $n$) span. Two work-efficient algorithms are known: batch-parallel Euler Tour Trees by Tseng et al. [*ALENEX'19*, (2019), pp. 92–106] and parallel Rake-Compress (RC) Trees by Acar et al. [*ESA'20*, (2020), pp. 2:1–2:23]. Both however are randomized and work efficient in expectation. Several downstream results that use these data structures (and indeed to the best of our knowledge, all known work-efficient parallel batch-dynamic graph algorithms) are therefore also randomized.

In this work, we give the first deterministic work-efficient solution to the problem. Our algorithm maintains a parallel RC-Tree on $n$ vertices subject to batches of $k$ edge updates deterministically in worst-case $O(k \log(1 + n/k))$ work and $O(\log n \log \log k)$ span on the Common-CRCW PRAM. We also show how to improve the span of the randomized algorithm from $O(\log n \log^* n)$ to $O(\log n)$.

Lastly, as a result of our new deterministic algorithm, we also derandomize several downstream results that make use of parallel batch-dynamic dynamic trees, previously for which the only efficient solutions were randomized.

## CCS CONCEPTS

• **Theory of computation → Dynamic graph algorithms**.

## KEYWORDS

dynamic trees, parallel graph algorithms, batch-dynamic algorithms

**ACM Reference Format:**
Daniel Anderson and Guy E. Blelloch. 2024. Deterministic and Low-Span Work-Efficient Parallel Batch-Dynamic Trees. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626183.3659976



*SPAA '24, June 17–21, 2024, Nantes, France*

ACM ISBN 979-8-4007-0416-1/24/06.
https://doi.org/10.1145/3626183.3659976

## 1 INTRODUCTION

The dynamic trees problem dates back to the seminal work of Sleator and Tarjan [48] on *Link-Cut trees*. The problem is to maintain a forest of trees subject to the insertion and deletion of edges, while answering queries about the forest. Examples of queries include connectivity (is there a path from $u$ to $v$), the weight of all vertices in a specific subtree, and the weight of the heaviest (or lightest) edge on a path. The latter is a key ingredient in the design of efficient algorithms for the maximum flow problem [5, 29, 31, 48, 49]. Dynamic trees are also ingredients in algorithms for dynamic graph connectivity [1, 20, 33, 36, 39] dynamic minimum spanning trees [9, 20, 33, 34, 36], and minimum cuts [8, 23, 25, 40].

There are a number of efficient ($O(\log n)$ time per operation) dynamic tree algorithms, including Sleator and Tarjan's *Link-Cut Tree* [48], Henzinger and King's *Euler-Tour Trees* [33], Frederickson's *Topology Trees* [20–22], Holm and de Lichtenberg's *Top Trees* [6, 35, 50], and Acar et al.'s Rake-Compress Trees [2–4]. Most of these algorithms are sequential and handle single edge updates at a time.

Exceptions are Tseng et al.'s *Batch-Parallel Euler-Tour Trees* [51] and Acar et al.'s *Parallel Batch-Dynamic RC-Trees* [2]. These algorithms implement *batch-dynamic* updates, which take a set of $k$ edges to insert or delete with the goal of doing so in parallel. Both of these algorithms achieve work-efficient $O(k \log(1 + n/k))$ work, which matches the sequential algorithms ($O(\log n)$ work) for low values $k$, and is optimal ($O(n)$ work) for large values of $k$. Both of these algorithms, however, are randomized.

Batch-dynamic trees are an important ingredient in several recent breakthrough results in parallel graph algorithms. RC-Trees are the backbone of the first ever work-efficient parallel algorithm for minimum cuts [8] and also a key ingredient in the first nearly work-efficient highly parallel undirected depth-first search algorithm [26]. RC-Trees also underpin the first work-efficient parallel incremental minimum spanning tree algorithm [9], and batch-dynamic Euler-tour trees were an ingredient in the first nearly work-efficient parallel batch-dynamic connectivity algorithm [1].

Other graph problems have also been studied in the parallel batch-dynamic model, such as dynamic minimum spanning trees [9, 17, 18, 45, 47, 52], and approximate $k$-core decomposition [41]. However, to the best of our knowledge, all work-efficient parallel batch-dynamic graph algorithms are randomized. Indeed, avoiding randomization seems difficult even for some classic static problems. Finding a spanning forest, for instance, has a simple $O(m)$-time sequential algorithm, and an $O(m)$ work, $O(\log n)$ span randomized parallel algorithm has been known for twenty years [46], but no deterministic equivalent has been discovered. The best deterministic algorithm requires an additional $\alpha(n, m)$ factor of work [16].

The closest work to ours is Ghaffari et. al's parallel undirected DFS [26] which makes use of RC-Trees and includes a sketch of how the algorithm could be derandomized, though not work efficiently.

***Our results.*** We design a work-efficient algorithm for the batch-dynamic trees problem that is deterministic and runs in polylog $n$ span on the Common-CRCW PRAM.

Theorem 1.1. *There is a deterministic parallel batch-dynamic algorithm that maintains a balanced Rake-Compress Tree (RC-Tree) of a bounded-degree forest subject to batches of $k$ edge updates (insertions, deletions, or both) in $O(k \log(1 + n/k))$ work and $O(\log n \log \log k)$ span on the Common-CRCW PRAM. $k$ may vary between batches.*

The resulting RC-Tree is amenable to all existing query algorithms for parallel RC-Trees [2] and hence it can solve batch connectivity queries, batch subtree queries, batch path queries, and non-local queries such as nearest marked vertex queries [7].

Corollary 1.2. *There are deterministic parallel batch-dynamic algorithms for batches of $k$ queries of:*

*(1) dynamic forest connectivity,*
*(2) subtree sums of an associative operation over vertices,*
*(3) path sums of an associative and invertible operation over edges,*
*(4) nearest marked vertex,*

*running in $O(k \log(1 + n/k))$ work and $O(\log n)$ span.*

As a byproduct of our techniques, we also optimize the randomized variant of the algorithm and obtain the following improved result, which improves over the $O(\log n \log^* n)$ span of Acar et al. [2].

Theorem 1.3. *There is a randomized parallel batch-dynamic algorithm that maintains a balanced Rake-Compress Tree (RC-Tree) of a bounded-degree forest subject to updates of $k$ edges in $O(k \log(1 + n/k))$ expected work and $O(\log(n))$ span w.h.p.*[1]

Randomized parallel batch-dynamic trees have already been used as key ingredients in several other parallel batch-dynamic algorithms. As an additional result, existing algorithms for fully dynamic connectivity [1] and incremental minimum spanning trees [9] can be derandomized by using our deterministic RC-Tree, assuming that the underlying graph is ternarized (transformed to constant degree [20]). Details will be provided in the full version of the paper.

Theorem 1.4. *There is a deterministic parallel batch-dynamic algorithm which, given batches of $k$ edge insertions, deletions, and connectivity queries on a bounded-degree graph on $n$ vertices and $m$ edges, processes insertions and deletions in $O(k \log n \log(1 + n/\Delta)\alpha(n, m))$ amortized work in $O(\text{polylog}\, n)$ span, and answers all queries in $O(k \log(1 + n/k))$ work and $O(\text{polylog}\, n)$ span, where $\Delta$ is the average batch size of all deletion operations.*

Theorem 1.5. *There is a deterministic parallel batch-incremental algorithm which maintains a minimum spanning forest of a bounded-degree graph on $n$ vertices given batches of $k$ edge insertions in $O(k \log(1 + n/k) + k \log \log n)$ work in $O(\text{polylog}\, n)$ span.*

[1] We say that a statement happens *with high probability* (w.h.p.) in $n$ if for any constant $c$, the constants in the statement can be set such that the probability that the event fails to hold is $O(n^{-c})$. The constants themselves are often hidden by big-O notation.

***Overview.*** Previous implementations of RC-Trees are all based on applying self-adjusting computation to randomized parallel tree contraction [2–4], i.e., a static tree contraction algorithm is implemented in a framework that automatically tracks changes to the input values, and selectively recomputes all procedures that depend on changed data. In this setting, one has to fix the randomness upfront so that repeating precisely the same computation will always result in the same outcome.

The static tree contraction process works by contracting all vertices of degree one (leaves) and a random independent set of degree two vertices obtained by flipping coins and taking the ones that flipped heads but their neighbors did not. This process is repeated in rounds until only a singleton vertex remains. When a new edge is added to the forest, the contraction must be updated to be consistent with the presence of the new edge. This is efficient if the new contracted forest does not differ substantially from the original one, and this is shown to be true in expectation. Intuitively, this is likely because the random coin flips ensure that a constant fraction of the vertices of the forest contract in expectation, which rapidly eliminates the areas of the forest that are affected by the update. However, the downside is that this technique is difficult to make deterministic since the random coin flips are what make it unlikely that an adversary can insert an edge that causes catastrophic (i.e., expensive to update) cascading changes.

Our key insight is that to obtain an efficient deterministic algorithm, one must forgo the desire to always obtain the same contracted tree as if the algorithm were ran from scratch, since if the algorithm is only able to produce a single canonical tree, an adversary can always find an update that maximizes the difference between the old and new canonical trees. Instead, we only require that the contraction is always valid and need not correspond to a particular canonical tree. In essence, we give up history independence (the fact that the resulting data structure does not depend on the order of the updates, only the final structure [3, 43]) which was paid for by randomization, to obtain determinism.

Our dynamic algorithm is therefore not based on self-adjusting computation. Instead, our variant of tree contraction deterministically contracts a maximal independent set (MIS) of degree-one-or-two vertices each round. When an update is made to the forest, our algorithm identifies the set of *affected vertices* and then *greedily* updates the tree contraction by computing an MIS of the affected vertices and updating the contraction accordingly. Critically, these greedy changes may not be the same choices that the algorithm would have made if running from scratch. The key insights are in carefully establishing the criteria for vertices being affected such that the update is correct, while minimizing the number of such vertices so that it is efficient.

## 2 PRELIMINARIES

### 2.1 Model of computation

***The PRAM and work-span analysis.*** We use the PRAM (Parallel Random Access Machine) model with work-span analysis [11, 37]. In particular we assume an algorithm has access to an unbounded shared memory and takes sequence of $s$ steps. Each step $i$ performs $w_i$ constant-time operations in parallel (each has access to its index, $[0, \ldots, w_i)$). The steps are executed one after the other sequentially.

The *work* of the algorithm is the total number of operations performed, i.e., $\sum_{i=1}^{s} w_i$, and the *span* is the number of steps $s$. Any algorithm with $W$ work and $S$ span can be scheduled on a traditional $p$-processor PRAM [19] in $O(W/p + S)$ time [13] and any $p$ processor PRAM algorithm that takes $T$ time, does $O(pT)$ work and has $O(T)$ span.

The EREW, CREW and CRCW variants of the PRAM differ in the assumptions about whether the concurrent operations within a single step can access the same memory location or not. The EREW (*Exclusive-Read Exclusive-Write*) variant allows no concurrent access to any location, the CREW (*Concurrent-Read Exclusive-Write*) allows concurrent reads but not writes, and the CRCW (Concurrent-Read Concurrent-Write) allows both concurrent reads and writes. The CRCW model may be further subdivided by the behaviour of concurrent writes. The *Common*-CRCW PRAM permits concurrent writes but requires that all concurrent writes to the same location write the same value, else the computation is invalid. The *Arbitrary*-CRCW PRAM permits concurrent writes of different values and specifies that an arbitrary processor's write succeeds, but the algorithm may make no assumption about which processor succeeds. Our algorithms use the Common-CRCW model.

We note that the PRAM model can be mapped onto less synchronous models. For example any PRAM algorithm can be mapped onto the binary-forking model while preserving the work and increasing the span by a factor of $O(\log n)$ [12]. It is likely the span for some of the algorithms in this paper can be improved for the binary-forking model over this naive simulation.

## 2.2 Approximate prefix sums and compaction

The prefix sum operation takes an integer sequence $[a_0, \ldots, a_{n-1}]$, and returns the result $[s_0, s_1, \ldots, s_n]$ such that $s_0 = 0$ and $s_{i+1} = s_i + a_i$ for $0 \leq i < n$. Among many applications, the prefix sum can be used to compact an array so as to only keep the elements that satisfy some predicate [10]. Prefix sums, and hence compaction, can be computed in $O(n)$ work and $O(\log n)$ span. Goldberg and Zwick show how the span can be improved to $O(\log \log n)$ for an approximate varsion of the problem—i.e., one that takes an integer input sequence $[a_0, a_1, \ldots, a_{n-1}]$, returns the result $[s_0, s_1, \ldots, s_n]$ such that $s_0 = 0$, $s_{i+1} \leq s_i + a_i$ $(0 \leq i < n)$, and $s_n \leq c \sum_0^{n-1} a_i$. This can be used for approximate compaction, where if there are $m$ elements satisfying the predicate, they will be compacted into an output array of length $O(m)$ with other slots marked as empty.

Lemma 2.1 ([32]). *There exists a deterministic Common-CRCW algorithm for approximate prefix sums and approximate compaction on an array of $n$ elements in $O(n)$ work and $O(\log \log n)$ span.*

## 2.3 Approximate counting sort

We will use an approximate counting sort as a subroutine. An approximate counting sort takes $n$ integer keys (and possibly associated values) in the range $[0..m)$. It returns an array of size $O(n)$ where the keys are sorted, but possibly with gaps. The gaps are marked as such. Furthermore for each value in the range, the sort returns a pointer to a position in the array such that all keys earlier are less than the value, and all keys at the position or later are greater or equal to the value. This can be achieved by combining the counting sort of Cole and Vishkin [15] with the approximate prefix sum algorithm of Goldberg and Zwick [32] described above. This approximate counting sort works in three phases.

(1) Partition the input into chunks of size $m$, and within each chunk count the number of entries for each of the $m$ possible values. This is done sequentially within the chunk but parallel across chunks. View the output as an $m \times (n/m)$ array $S$.
(2) Flatten $S$ into row-major order (i.e. all the counts for the same value are contiguous). Now run a single approximate prefix sum across all the $n$ counts.
(3) View the output again as an $m \times (n/m)$ offset array $B$, and allocate an output array of length $cn$ (same $c$ as used in the definition of approximate prefix sums). Now go through each chunk again (sequentially within chunks and parallel across them) and use the offset array as the position to start writing keys of the given value—i.e., $B[i, j]$ gives the start offset for values $i$ in chunk $j$.

This results in a valid approximate counting sort. Clearly the output is of size $O(n)$, and all keys will be sorted (possibly with gaps due to the approximate prefix sum). The pointers to the start of the region for each key value $i$ can be found at $B[i, 0]$.

Lemma 2.2. *There exists a deterministic Common-CRCW approximate counting sort algorithm for $n$ integers in the range $[0, m)$ that runs in $O(n)$ work and $O(m + \log \log n)$ span.*

This follows since the work and span for the first and third step are $O(n)$ and $O(m)$ respectively, and the the cost of the second step is given by Lemma 2.1.

## 2.4 Colorings and maximal independent sets

Parallel algorithms for graph coloring and maximal independent sets are well studied [14, 28, 30, 38, 42]. Our algorithm uses a subroutine that finds a maximal independent set of a collection of chains, i.e., a set of vertices of degree one or two.

Goldberg and Plotkin [30] give an algorithm for $O(\log^{(c)}(n))$-coloring[2] a constant-degree graph in $O(c \cdot n)$ work and $O(c)$ span in the EREW PRAM model. Given a $c$-coloring, one can easily obtain a maximal independent set as follows. Sequentially, for each color, look at each vertex of that color in parallel. If a vertex of the current color is not adjacent to a vertex already selected for the independent set, then select it. This takes $O(c \cdot n)$ work and $O(c)$ span, hence a constant coloring yields an $O(n)$ work and constant span algorithm.

For any choice of $c$, the above algorithms do not yield a work-efficient algorithm for maximal independent set since it is not work efficient to find a constant coloring, and not work efficient to convert a non-constant coloring into an independent set. To make it work efficient, we can borrow a trick from Cole and Vishkin [15]. We first produce a $\log \log n$ coloring in $O(1)$ span, then use an approximate counting sort to sort the vertices by color, allowing us to perform the coloring-to-independent-set conversion work efficiently. This results in an $O(n)$ work, $O(\log \log n)$ span algorithm for maximal independent set in a constant-degree graph.

Lemma 2.3. *There exists a deterministic Common-CRCW algorithm that finds an MIS in a constant-degree graph in $O(n)$ work and $O(\log \log n)$ span.*

[2]The notation $\log^{(c)} n$ refers to the *repeated/nested logarithm* function, i.e., $\log^{(1)} n = \log n$ and $\log^{(c)} n = \log(\log^{(c-1)} n)$. E.g., $\log^{(2)} n = \log \log n$.

## 2.5 Parallel tree contraction

Tree contraction is a procedure for computing functions over trees in parallel in low span [44]. It involves repeatedly applying *rake* and *compress* operations to the tree while aggregating data specific to the problem. The rake operation removes a leaf from the tree and aggregates its data with its parent. The compress operation replaces a vertex of degree two and its two adjacent edges with a single edge joining its neighbors, aggregating any data associated with the vertex and its two adjacent edges.

Rake and compress operations can be applied in parallel as long as they are applied to an independent set of vertices. Miller and Reif [44] describe a linear work and $O(\log n)$ span randomized algorithm that performs a set of *rounds*, each round raking every leaf and an independent set of degree two vertices by flipping coins. They show that it takes $O(\log n)$ rounds to contract any tree to a singleton w.h.p. They also describe a deterministic algorithm but it is not work efficient. Later, Gazit, Miller, and Teng [24] obtain a work-efficient deterministic algorithm with $O(\log n)$ span.

These algorithms are defined for constant-degree trees, so non-constant-degree trees are handled by converting them into bounded-degree equivalents, e.g., by ternerization [20].

## 2.6 Rake-Compress Trees (RC-Trees)

RC-Trees [2, 4, 7] are based on viewing the process of parallel tree contraction as inducing a *clustering*. A cluster is a connected subset of edges and vertices. The base clusters are singletons containing the individual edges and vertices, so there are $n + m$ base clusters. The internal (non-base) clusters that arise have the following properties:

(1) The subgraph induced by the vertex subset is connected,
(2) the edge subset contains all of the edges in the subgraph induced by the vertex subset,
(3) every edge in the edge subset has at least one endpoint in the vertex subset.

This makes them somewhat of a hybrid of topology trees [20–22] and top trees [6, 35, 50], which cluster just the vertices or just the edges respectively. Importantly but somewhat unintuitively, an RC cluster may contain an edge without containing both endpoints of that edge. A vertex that is an endpoint of an edge, but is not contained in the same cluster as that edge is called a *boundary vertex* of the cluster containing the edge.

The clusters of the RC-Tree always have at most two boundary vertices, and hence can be classified as *unary clusters*, *binary clusters*, or *nullary clusters*. Unary clusters arise from rake operations and have one boundary vertex. Binary clusters arise from compress operations and have two boundary vertices. A binary cluster with boundary vertices $u$ and $v$ always corresponds to an edge $(u, v)$ in the corresponding round of tree contraction. Binary clusters can therefore be thought of as "generalized edges" (this notion is also used by top trees [6]). The path between the two boundary vertices of a binary cluster (in the original forest) is called its *cluster path*.

To form a recursive clustering from a tree contraction, we begin with the base clusters and the uncontracted tree. On each round, for each vertex $v$ that contracts via rake or compress (which, remember, form an independent set), we identify the set of clusters that are adjacent to $v$ (equivalently, all clusters that have $v$ as a boundary vertex). These clusters are merged into a single cluster consisting of the union of their contents. We call $v$ the *representative* vertex of the resulting cluster. The boundary vertices of the resulting cluster are the union of the boundary vertices of the constituents, minus $v$.

The rake operations always creates unary clusters and the compress operation creates binary clusters. When a vertex has no neighbors, it *finalizes* and creates a nullary cluster representing the root of its connected component. Since each vertex rakes, compresses, or finalizes exactly once, there is a one-to-one mapping between representative vertices of the original tree and internal clusters.

An RC-Tree then encodes this recursive clustering. The leaves of the RC-Tree are the base edge clusters of the tree, i.e., the singleton edges and vertices (note that the base cluster for vertex $v$ is always a direct child of the cluster for which $v$ is the representative, so omitting it from the RC-Tree loses no information). Internal nodes of the RC-Tree are clusters formed by tree contraction, such that the children of a node are the clusters that merged to form it. The root of the RC-Tree is a cluster representing the entire tree, or connected component in the case of a disconnected forest.

Queries are facilitated by storing augmented data on each cluster, which is aggregated from the child clusters at the time of its creation. Since tree contraction removes a constant fraction of the vertices at each round, the resulting RC-Tree is balanced, regardless of how balanced or imbalanced the original tree was. This allows queries to run in $O(\log n)$ time for single queries, or *batch queries* to run in $O(k \log(1 + n/k))$ work and $O(\log n)$ span [7].

An example of a tree, a recursive clustering, and the corresponding RC-Tree are depicted in Figure 1. The base vertex clusters are omitted since in practice they can be combined with the internal cluster that they represent.

# 3 DETERMINISTIC DYNAMIC CONTRACTION

## 3.1 The contraction data structure

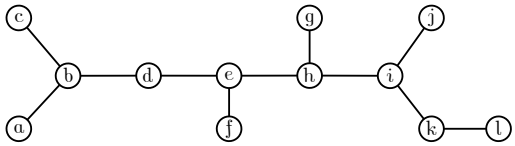
Our algorithm maintains a *contraction data structure*, containing:

- for each level of the contraction process, an adjacency list representation of the contracted tree,
- the clusters of the RC-Tree.

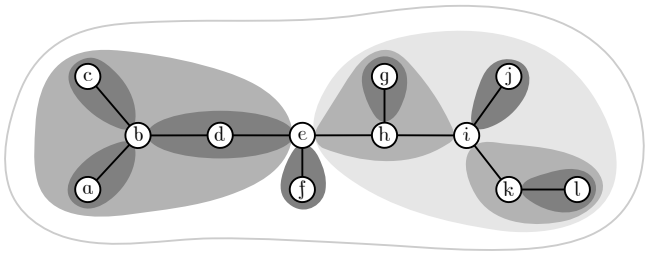
It records the history of the tree contraction process beginning from the input forest $F$ as it contracts to a forest of singletons. This is the heart of the static and dynamic algorithms; the static algorithm builds it initially and the dynamic algorithm updates it efficiently.

The data structure is organized by *levels*, each corresponding with a *round* of tree contraction. *Level 0* is the original forest $F_0 = F$. Round $i$ of tree contraction produces the forest $F_i$ (level $i$) from $F_{i-1}$. A vertex is live at level $i$ if it has not contracted yet, or dead otherwise. We say that a vertex contracts in $F_{i-1}$ (or equivalently, contracts in round $i$) if it is live at level $i - 1$ but not at level $i$. For each level in which a vertex is live, the contraction data structure stores an adjacency list for that vertex. Each entry in a vertex $v$'s adjacency list is one of three possible kinds of value:

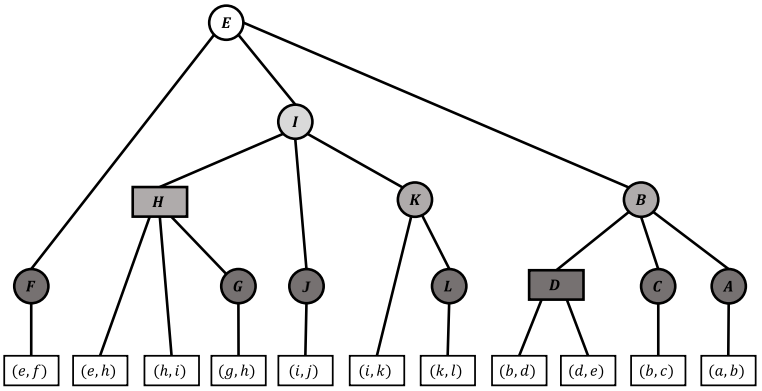
(1) a pointer to an edge $(u, v) \in F$,
(2) a pointer to a binary cluster for which $v$ is one of the boundary vertices. If $u$ is the other boundary vertex, this represents an edge $(u, v) \in F_i$,
(3) a pointer to a unary cluster for which $v$ is the boundary vertex. This does not represent any edge in the contracted tree.




**(a) An unrooted tree**




**(b) A recursive clustering of the tree produced by tree contraction. Clusters produced in earlier rounds are depicted in a darker color.**




**(c) The corresponding RC-Tree. Clusters produced from rakes (and the root cluster) are shown as filled circles, and clusters produced from compress as rectangles. The base edge clusters (edges of the original tree) are labeled in lowercase, and the composite clusters are labeled with the uppercase of their representative vertex. Base vertex clusters are omitted. The shade of a cluster corresponds to its height in the clustering. Lower heights (i.e., its representative vertex contracted earlier) are darker.**

**Figure 1: A tree, a clustering, and the corresponding RC-Tree [2].**

At level 0 the adjacency list therefore stores pointers to the edges adjacent to each vertex. At later levels, in a contracted forest, some of the edges are not edges of $F$, but are the result of a compress operation and represent a binary cluster (Case 2), which may contain augmented data (e.g., the sum of the weights in the cluster, the maximum weight edge on the cluster path, etc, depending on the application). Additionally, vertices that rake accumulate augmented data inside their resulting unary cluster that needs to be aggregated when their parent cluster is created (Case 3). Acar et al. [4] and Anderson [7] describe many examples of augmented data for various applications.

Clusters contain pointers to their child clusters alongside any augmented data. Each internal cluster corresponds uniquely to the vertex that contracted to form it, so counting them plus the base clusters, the RC-Tree contains exactly $2n + m$ clusters. It is convenient to omit the base-vertex clusters, since if the user wishes to store augmented data on the vertices, this can be stored on the unique internal cluster for which that vertex is the representative, leading to a cleaner representation with $n + m$ clusters.

## 3.2 Static deterministic tree contraction

We build an RC-Tree deterministically using a variant of Miller and Reif's tree contraction algorithm. Instead of contracting all degree-one vertices (leaves) and an independent set of degree two vertices, we instead contract *any maximal independent set of degree one and two vertices*, i.e., leaves are not all required to contract. This difference is subtle but critical for the efficiency of the update algorithm. The reason will become clear during the analysis, but essentially, not forcing leaves to contract reduces the number of vertices that need to be reconsidered during an update, since a vertex that was previously not a leaf becoming a leaf might cause a chain reaction requiring many more vertices to be updated. Such a chain reaction is undesirable, and our variant avoids it.

*Definition 3.1.* We say that a tree contraction is maximal at some round if the set of vertices that contract form a maximal independent set of degree one and two vertices. A tree contraction is maximal if it is maximal at every round.

Recall that $F_0$ denotes the initial forest and $F_i$ for $i \geq 1$, the forest obtained by applying one round of maximal tree contraction to $F_{i-1}$. To obtain a maximal tree contraction of $F_i$, consider in parallel every vertex of degree one and two. These are the *eligible* vertices. We find a maximal independent set of eligible vertices by using Lemma 2.3. This takes $O(|F_i|)$ work and $O(\log \log n)$ span. To write down the vertices of $F_{i+1}$, we can apply approximate compaction to the vertex

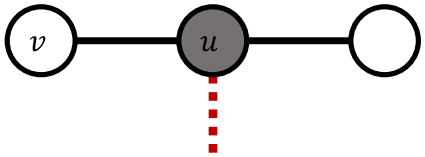



**Figure 2: A vertex $v$ is affected because its neighbor $u$ is affected, and $v$ depends on $u$ contracting in order for the contraction to be maximal. In this case $u$'s degree has increased to three so it can not contract, so $v$ must contract instead.**

set of $F_i$, filtering those which were selected to contract. This also takes $O(|F_i|)$ work and $O(\log \log n)$ span [32].

Each vertex writes itself into its neighbors' adjacency list in the next level. Since the degree of the tree is constant, this does not require complex synchronization primitives and can easily be achieved by writing into a fixed-size array, one slot for each neighbor. For vertices that do not contract, they can simply copy their corresponding entry in their neighbors' adjacency list to the next level assuming the neighbor is still alive in that round. For vertices that do contract, they create the corresponding RC cluster and write a pointer to the cluster into the adjacency list. For example, if a vertex $v$ with neighbors $u$ and $w$ compresses in round $i$, it writes a pointer to the binary cluster formed by $v$ (whose boundary vertices are $u$ and $w$) into the adjacency lists at level $i+1$ of $u$ and $w$.

To build the RC-Tree clusters, it suffices to observe that when a vertex $v$ contracts, the contents of its adjacency list are precisely the child clusters of the resulting cluster. Hence in constant time we can build the cluster by aggregating the augmented values of the children and creating a corresponding cluster. If at a round, a vertex is isolated (has no neighbors), it creates a root (nullary) cluster.

### 3.3 A deterministic update algorithm

A *dynamic update* consists of a set of $k$ edges to be added or deleted (a combination of both is valid). The update begins by modifying the adjacency lists of the $2k$ endpoints of the modified edges. Call this resulting forest $F'_0$, to denote the forest after the update (note that updates are performed in-place since copying the entire forest would immediately ruin the work bound). For each round $i$, the goal of the update algorithm is now to produce $F'_i$, an updated tree contraction for level $i$, using $F_i, F_{i-1}$ and $F'_{i-1}$. This approach is similar to the randomized change propagation algorithm [2], but the details are very different, and the devil is in the details.

***Affected vertices.*** To perform the update efficiently, we define the notion of an *affected vertex*. Affected vertices are those that need to be reprocessed since their state might need to change (from contracted to not contracted or vice-versa) to keep the invariant that the tree contraction is always maximal. The subtlety in correctly defining the affected vertices is that the set must be comprehensive enough that the algorithm is correct, but also minimal enough that it is efficient. Having too many affected vertices would be inefficient, but having too few would be incorrect. This definition strikes the right balance and achieves the best of both worlds.

*Definition 3.2 (Affected).* A vertex is *affected* at level $i$ if any of:

(1) It is live in one of $F_i$ and $F'_i$ but not the other,

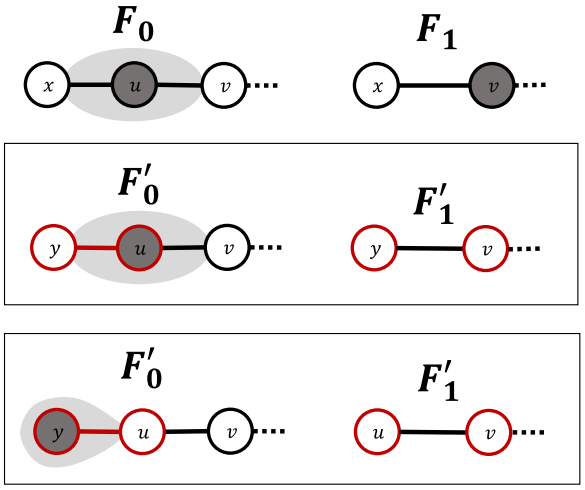



**Figure 3: Direct affection (two possibilities): A vertex $u$ directly affects its neighbor $v$. $u$ is affected at round 0 since its adjacency list was changed. Since $u$ contracts in $F_0$ and changes the adjacency list of $v$ at round 1, $v$ becomes affected. Note that this can happen either when $u$ contracts in $F'_0$ (first example) or when it does not (second example).**

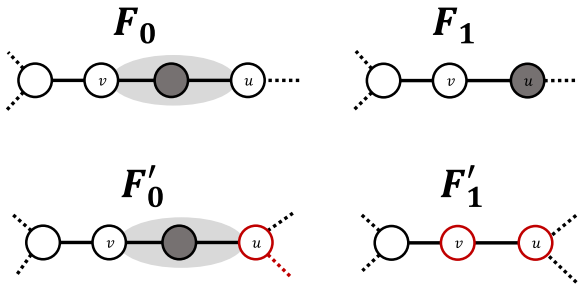



**Figure 4: Affection by dependence: A vertex $u$ affects its neighbor $v$ in $F_1$ by dependence. Even though $v$'s adjacency list is the same in both forests, it is affected because it depended on $u$ to contract in $F_1$ for maximality to be satisfied. Since $u$ can no longer contract in $F'_1$, it is important that $v$ is able to.**

(2) it is live in both $F_i$ and $F'_i$ but has different adjacency lists,
(3) it is live in both $F_i$ and $F'_i$, and still live in $F_{i+1}$ (i.e., did not contract in $F_i$), but all of its neighbors in $F_i$ that contracted in $F_i$ are affected at level $i$.

The first two cases of affected vertices are intuitive. If a vertex used to exist at level $i$ but no longer does, or vice versa, it definitely needs to be updated in level $i+1$. If a vertex has a different adjacency list than it used to, then it definitely needs to be processed because it can not possibly contract in the same manner, or may change from being eligible to ineligible to contract or vice versa.

The third case of affection is more subtle, but is important for the correctness and efficiency of the algorithm. We call the third case *affection by dependence*. Suppose an eligible vertex $v$ doesn't contract in $F_i$. Since the contraction forms a maximal independent

**Algorithm 1** Batch update

```
1: procedure BatchUpdateForest(E+, E−)
2:     Create F'_0 from F_0 by adding edges in E+ and removing edges in E−
3:     Determine the affected vertices at level 0 from the endpoints of E+ ∪ E−
4:     for each level i from 1 to log_{6/5} n do
5:         Find an MIS of update-eligible affected vertices in F_{i−1}
6:         Obtain F'_i from F_i by recomputing the adjacency lists of the affected vertices, contracting those vertices in the new MIS
7:         Determine the affected vertices at level i from the result (Definition 3.2)
8:     end for
9: end procedure
```

set, at least one of $v$'s neighbors must contract. If all such neighbors are affected at level $i$, they may no longer contract in $F'_i$ because of the update, which would leave $v$ uncontracted and without a contracting neighbor, violating maximality. Therefore $v$ must also be considered affected to prevent this scenario. Figure 2 shows an example where this matters.

Note that by the definition, vertices only become affected because they have an affected neighbor at the previous or current level. We call this *spreading* affection.

*Definition 3.3 (Spreading affection).* An affected vertex $u$ *spreads to* $v$ at level $i+1$ if $v$ was unaffected at level $i$ and is affected at level $i+1$ because either

(1) $v$ is a neighbor of $u$ at level $i$, such that $u$ is affected and contracts in round $i+1$ in either $F_i$ or $F'_i$, or
(2) $v$ is a neighbor of $u$ in $F_{i+1}$, such that $v$ does not contract in $F_{i+1}$, but $u$, which is affected, does.

We call Case (1), *spreading directly* and Case (2) *spreading by dependence*. Figure 3 shows two examples of directly spreading affection. Figure 4 shows an example of spreading affection by dependence.

***The algorithm.*** With the definition of affected vertices, the update algorithm can be summarized as stated in Algorithm 1. The levels are processed one after the other, but the subroutines that run on each level are parallel. In the update algorithm, a vertex is *update-eligible* if it has degree one or two, and it is not adjacent to an unaffected vertex that contracts. Line 5 is accomplished using Lemma 2.3. This finds a maximal independent set of update-eligible affected vertices in $O(k)$ work and $O(\log\log k)$ span.

Line 6 is implemented by looking at each affected vertex in parallel and updating it to reflect its new behaviour in $F'_i$. This entails writing the corresponding adjacent edges into the adjacency lists of its neighbors in level $i$ if it did not contract, or writing the appropriate cluster values if it did. At the same time, for each contracted vertex, the algorithm computes the augmented value on the resulting RC cluster from the values of the children.

Line 7 is implemented by looking at each affected vertex at the previous level and any vertex within distance two of those in parallel, then filtering those which do not satisfy the definition of affected. Note that by the definition of affected, looking at vertices within distance two is sufficient since at worst, affection can only spread to neighbors and possibly those neighbors' uncontracted neighbors. Using approximate compaction [32], this step takes linear work in the number of affected vertices and $O(\log\log n)$ span. In Section 4, we show that the number of affected vertices at each level is $O(k)$, so this is efficient.

***Correctness.*** We now argue that the algorithm is correct. This consists in proving the invariant that after each update, the contraction is maximal, i.e., for every level of the tree contraction and every vertex $v$ of degree one or two, either $v$ contracts or $v$ has at least one neighbor that contracts.

Lemma 3.4. *After running the update algorithm, the contraction is still maximal, i.e., the contracted vertices at each level form a maximal independent set of degree one and two vertices.*

Proof. Our goal is to prove that after an update, every eligible vertex (vertices of degree one or two) either contracts or is adjacent to a vertex that contracts. The proof is by casework based on whether $v$ is unaffected and contracts or does not contract, or is affected and update-eligible or not update-eligible. Suppose the tree contraction was maximal at level $i$ before the update. We will argue that after the update the contraction is still maximal at level $i$.

Consider any eligible vertex $v \in F'_i$ that was unaffected at level $i$ and contracts in $F'_i$. Since it was unaffected it exists in $F_i$ and it also contracted in $F_i$. We need to argue that no neighbor of $v$ contracts in $F'_i$. For any unaffected neighbor $u$, it didn't contract in $F_i$ and hence still doesn't contract in $F'_i$. If $u$ is an affected neighbor, it is not considered update-eligible because $v$ is unaffected and contracted and hence $u$ does not contract. Therefore none of $v$'s neighbors contract in $F'_i$ and $v$ satisfies the invariant.

Now suppose $v \in F'_i$ was eligible and unaffected at level $i$ and didn't contract in $F'_i$. We need to argue that $v$ has a neighbor in $F'_i$ that contracted. Since $v$ is unaffected it exists and also didn't contract in $F_i$. Therefore it has a neighbor $u \in F_i$ that contracted. If $u$ is unaffected, then $u \in F'_i$ and contracts. If $u$ were affected, then $v$ would be affected by dependence, but $v$ is unaffected. Therefore all eligible unaffected vertices $v \in F'_i$ satisfy the invariant.

Consider any eligible affected vertex $v \in F'_i$. If $v$ has an unaffected contracted neighbor, then $v$ is not update-eligible and hence does not contract in $F'_i$, and has a contracted neighbor. Otherwise, $v$ is update-eligible and participates in the MIS. Since the algorithm finds a maximal independent set on the update-eligible affected vertices, $v$ either contracts and has no contracted neighbor, or doesn't contract and has a contracted neighbor. Therefore $v$ satisfies the invariant.

Together, we can conclude that every eligible vertex satisfies the invariant and hence the contraction is maximal. □

# 4 ANALYSIS

We start by proving some general and useful lemmas about the contraction process, from which the efficiency of the static algorithm immediately follows, and which will later be used in the analysis of the update algorithm.

## 4.1 Round and tree-size bounds

Lemma 4.1. *Consider a (non-singleton vertex) tree $T$ and suppose a maximal independent set of degree one and two vertices is contracted via* rake *and* compress *contractions to obtain $T'$. Then*

$$|T'| \leq \frac{5}{6}|T|$$

Proof. More than half of the vertices in any tree have degree one or two [53]. A maximal independent set among them is at least one third of them since every adjacent run of three vertices must have at least one selected, so at least one sixth of the vertices contract. Therefore the new tree has at most $\frac{5}{6}$ as many vertices. □

The lemma also applies to forests since it can simply be applied independently to each component. Note that singleton vertices finalize and hence the bound is true for the total size of the forest including the singletons. Three important corollaries follow that allow us to bound the cost of various parts of the algorithm. Corollary 4.2 gives the number of rounds required to fully contract a forest, and Corollary 4.3 gives a bound on the number of rounds required to shrink a forest to size $n/\log n$. Lastly, Corollary 4.4 gives a bound on the number of rounds required to shrink a tree to size $k$, which is useful in bounding the work of the batched update and query algorithms.

Corollary 4.2. *Given a forest on $n$ vertices, maximal tree contraction completely contracts a forest of $n$ vertices in $\log_{6/5} n$ rounds.*

Corollary 4.3. *Given a forest on $n$ vertices, after performing $\log_{6/5} \log n$ rounds of maximal tree contraction, the number of vertices in the resulting forest is at most $n/\log n$.*

Corollary 4.4. *Given a forest on $n$ vertices and any integer $k \geq 1$, after performing $\log_{6/5}(1 + n/k)$ rounds of maximal tree contraction, the number of vertices in the resulting forest is at most $k$.*

Proof. By Lemma 4.1, the number of vertices in each round is at most $5/6^{\text{th}}$s of the previous round, so the number remaining after round $r$ is at most $n\,(5/6)^r$. The three corollaries follow. □

## 4.2 Analysis of the static algorithm

Armed with the lemmas and corollaries of Section 4.1, we can now analyze the static algorithm.

Theorem 4.5. *The static maximal tree contraction algorithm can be implemented in $O(n)$ work and $O(\log n \log\log n)$ span for a forest of $n$ vertices.*

Proof. The work performed at each round is $O(|F_i|)$, i.e., the number of live vertices in the forest at that round. By Lemma 4.1, the total work is therefore at most

$$\sum_{i=0}^{\infty} n \left(\frac{5}{6}\right)^i = n \sum_{i=0}^{\infty} \left(\frac{5}{6}\right)^i = 6n.$$

The span of the algorithm is $O(\log\log n)$ per round to perform the maximal independent set and approximate compaction operations by Lemmas 2.3 and 2.1. By Corollary 4.2 there are $O(\log n)$ rounds, hence the total span is $O(\log n \log\log n)$. □

## 4.3 Analysis of the update algorithm

We first sketch a summary of the proof then present the full analysis.

***Summary.*** We begin by establishing the criteria for vertices becoming affected. Initially, the endpoints of the updated edges and a small neighborhood around them are affected. We call these the *origin vertices*. For each of these vertices, it may *spread* its affection to nearby vertices in the next round. Those vertices may subsequently spread to other nearby vertices in the following round and so on. As affection spreads, the affected vertices form an *affected component*, a connected set of affected vertices whose affection originated from a common origin vertex. An affected vertex that is adjacent to an unaffected vertex is called a *frontier vertex*. Frontier vertices are those which are capable of spreading affection. Note that it is possible that in a given round, a vertex that becomes affected was adjacent to multiple frontier vertices of different affected components, and is subsequently counted by both of them, and might therefore be double counted in the analysis. This is okay since it only overestimates the number of affected vertices in the end.

With these definitions established, our results show that each affected component consists of at most two frontier vertices, and that at most four new vertices can be added to each affected component in each round. Given these facts, since a constant fraction of the vertices in any forest must contract in each round, we show that the size of each affected component shrinks by a constant fraction, while only growing by a small additive factor. This leads to the conclusion that each affected component never grows beyond a *constant size*, and since there are initially $O(k)$ origin vertices, that there are never more than $O(k)$ affected vertices in any round. This fact allows us to establish that the update algorithm is efficient.

The analysis of the update algorithm follows a similar pattern to the analysis of the randomized change propagation algorithm [2], though once again with substantial tweaks to the details.

***The proofs.*** We now prove the aforementioned facts.

Lemma 4.6. *If $v$ is unaffected at level $i$, then $v$ contracts in round $i$ in $F$ if and only if $v$ contracts in round $i$ in $F'$.*

Proof. Unaffected vertices are ignored by the update algorithm, and hence remain the same before and after an update. □

We now establish that Definition 3.3 indeed encompasses all possibilities for affection to spread. If a vertex is not affected at level $i$ but is affected at level $i+1$, we say that $v$ *becomes affected in round $i$*, which we prove can happen in just two ways.

Lemma 4.7. *If $v$ becomes affected in round $i$, then one of the following is true:*

*(1) $v$ has an affected neighbor $u$ at round $i$ which contracted in either $F_i$ or $F'_i$*
*(2) $v$ does not contract by round $i+1$, and has an affected neighbor $u$ at round $i+1$ that contracts in $F_{i+1}$.*

Proof. First, since $v$ becomes affected in round $i$, it is not already affected at level $i$. Therefore, due to Lemma 4.6, $v$ does not contract, otherwise it would do so in both $F$ and $F'$ and hence be unaffected at level $i+1$. Since $v$ does not contract, $v$ has at least one neighbor, otherwise it would finalize.

Suppose that (1) is not true, i.e., that $v$ has no affected neighbors that contract in $F_i$ or $F'_i$ in round $i$. Then either none of $v$'s neighbors contract in $F_i$, or only unaffected neighbors of $v$ contract in $F_i$. By Lemma 4.6, in either case, $v$ has the same set of neighbors in $F_{i+1}$ and $F'_{i+1}$. Therefore, since $v$ is affected in level $i+1$, does not contract in either $F_i$ or $F'_i$, and has the same neighbors in both, it must be in Case (3) in the definition of affected. Therefore, $v$ has an affected neighbor that contracts in $F_{i+1}$.

Since $\neg(1) \implies (2)$, we have that $(1) \vee (2)$ is true. □

Our end goal is to bound the number of affected vertices at each level, since this corresponds to the amount of work required to update the contraction after an edge update.

Let $\mathcal{A}^i$ denote the set of affected vertices at level $i$.

Lemma 4.8. *For a batch update of size $k$ (insertion or deletion of $k$ edges), we have $|\mathcal{A}^0| \leq 6k$.*

Proof. An edge changes the adjacency list of its two endpoints. These two endpoints might contract in the first round, which affects their uncontracted neighbors by dependence. However, vertices that contract have degree at most two, so this is at most two additional vertices per endpoint. Therefore there are up to 6 affected vertices per edge modification, and hence up to $6k$ affected vertices. □

Each edge modified at level 0 affects some set of vertices, which spread to some set of vertices at level 1, which spread to some set of vertices at level 2 and so on. We will therefore partition the set of affected vertices into $s = |\mathcal{A}^0|$ *affected components*, indicting the "origin" of the affection. When a vertex $u$ spreads to $v$, it will add $v$ to its component on the next level.

More formally, we will construct $\mathcal{A}^i_1, \mathcal{A}^i_2, \ldots, \mathcal{A}^i_s$, which form a partition of $\mathcal{A}^i$. We start by arbitrarily partitioning $\mathcal{A}^0$ into $s$ singleton sets $\mathcal{A}^0_1, \mathcal{A}^0_2, \ldots, \mathcal{A}^0_s$. Given $\mathcal{A}^i_1, \mathcal{A}^i_2, \ldots, \mathcal{A}^i_s$, we construct $\mathcal{A}^{i+1}_1, \mathcal{A}^{i+1}_2, \ldots, \mathcal{A}^{i+1}_s$ such that $\mathcal{A}^{i+1}_j$ contains the affected vertices $v \in A^{i+1}$ that were either already affected in $\mathcal{A}^i_j$ or were spread to by a vertex $u \in \mathcal{A}^i_j$. Note that it is possible, under the given definition, for multiple vertices to spread to another, so this may overcount by duplicating vertices. Vertices are de-duplicated by only adding them to the affected component that they spread from via the lowest ID vertex as a tiebreaker.

*Definition 4.9 (Frontier).* A vertex $v$ is a *frontier* at level $i$ if $v$ is affected at level $i$ and a neighbor of $v$ in $F_i$ is unaffected at level $i$.

Lemma 4.10. *If $v$ is a frontier vertex at level $i$, then it is alive in both $F_i$ and $F'_i$ at level $i$, and is adjacent to the same set of unaffected vertices in both.*

Proof. If $v$ were dead in both forests it would not be affected and hence not a frontier vertex. If $v$ were live in one forest but dead in the other, then all of its neighbors would have a different set of neighbors in $F_i$ and $F'_i$ (they must be missing $v$) and hence all of them would be affected, so $v$ would have no unaffected neighbors and hence not be a frontier.

Similarly, consider an unaffected neighbor $u$ of $v$ in either forest. If $u$ was not adjacent to $v$ in the other forest, it would have a different set of neighbors and hence be affected. □

If a $v$ spreads affection to a vertex in round $i$, then by definition $v$ must be a frontier. Our next goal is to analyze the structure of the affected sets and show that the number of frontier vertices is small.

Lemma 4.11. *For all $i, j$, the forest induced by $\mathcal{A}^i_j$ in $F_i$ is a tree.*

Proof. When $i = 0$ the components are isolated vertices which are trivially trees. For $i > 0$, the rake and compress operations both preserve the connectedness of the underlying tree, and Lemma 4.7 shows that affection only spreads to neighboring vertices. □

Lemma 4.12. *$\mathcal{A}^i_j$ has at most two frontiers and $|\mathcal{A}^{i+1}_j \setminus \mathcal{A}^i_j| \leq 4$.*

Proof. We proceed by induction on $i$. At level 0, each group contains one vertex, so it definitely contains at most 2 frontier vertices. Consider some $\mathcal{A}^i_j$ and suppose it contains one frontier vertex $u$, which may spread directly by contracting (Definition 3.3). If $u$ spreads directly, then it either compresses or rakes in $F_i$ or $F'_i$. This means it has degree at most two in $F_i$ or $F'_i$, and by Lemma 4.10, it is therefore adjacent to at most two unaffected vertices, and hence may spread to at most these two vertices. Since $u$ contracts, it is no longer a frontier by Lemma 4.10, but its newly affected neighbors may become frontiers, so the number of frontiers is at most two.

Suppose $u$ spreads via dependency in round $i$ (Case 2 in Definition 3.3) in $\mathcal{A}^{i+1}_j$ and contracts in $F_{i+1}$. Since $u$ contracts in $F_{i+1}$, it has at most two neighbors, and by Lemma 4.10, it is also adjacent to at most two unaffected vertices, and may spread to at most these two vertices. If it spreads to one of them, that vertex may become a frontier and hence there are at most two frontier vertices. If it spreads to both of them, $u$ is no longer adjacent to any unaffected vertices and hence is no longer a frontier, so there are still at most two frontier vertices, and $|\mathcal{A}^{i+1}_j \setminus \mathcal{A}^i_j| \leq 3$.

Now consider some $\mathcal{A}^i_j$ that contains two frontier vertices $u_1, u_2$. By Lemma 4.11, $u_1$ and $u_2$ each have at least one affected neighbor. If either contract, it would no longer be a frontier, and would have at most one unaffected neighbor which might become affected and a frontier. Therefore the number of frontiers is preserved when affection is spread directly.

Lastly, suppose $u_1$ or $u_2$ spreads via dependency in round $i$. Since it would contract in $F_{i+1}$, it has at most one unaffected neighbor which might become affected and become a frontier. It would subsequently have no unaffected neighbor and therefore no longer be a frontier. Therefore the number of frontiers remains at most two and $|\mathcal{A}^{i+1}_j \setminus \mathcal{A}^i_j| \leq 4$. □

Now define $\mathcal{A}^i_{F,j} = \mathcal{A}^i_j \cap V^i_F$, the set of affected vertices from $\mathcal{A}^i_j$ that are live in $F$ at level $i$, and similarly define $\mathcal{A}^i_{F',j}$ for $F'$.

Lemma 4.13. *For every $i, j$ we have*

$$|\mathcal{A}^i_{F,j}| \leq 26.$$

Proof. Consider the subforest induced by the set of affected vertices $\mathcal{A}^i_{F,j}$. By Lemmas 4.11 and 4.12, this is a tree with two frontier vertices. The update algorithm finds and contracts a maximal independent set of affected degree-one-or-two vertices that are not adjacent to an unaffected vertex that contracts in $F_i$. There can be at most two vertices (the frontiers) that are adjacent to an unaffected vertex, and at most four new affected vertices appear by Lemma 4.11, so by Lemma 4.1, the size of the new affected set is

$$\begin{aligned}|\mathcal{A}^{i+1}_{F,j}| &\leq 4 + \frac{5}{6}\left(|\mathcal{A}^i_{F,j}| - 2\right) + 2 \\ &= \frac{26}{6} + \frac{5}{6}|\mathcal{A}^i_{F,j}|.\end{aligned}$$

Since $|\mathcal{A}^0_{F,j}| = 1$, we obtain

$$|\mathcal{A}^{i+1}_{F,j}| \leq \frac{26}{6}\sum_{r=0}^{\infty}\left(\frac{5}{6}\right)^r = \frac{\frac{26}{6}}{1-\frac{5}{6}} = \frac{\frac{26}{6}}{\frac{1}{6}} = 26,$$

just as desired. □

Lemma 4.14. *Given a batch update of $k$ edges, for every $i$*

$$|\mathcal{A}^i| \leq 312k.$$

Proof. By Lemma 4.8, there are at most $6k$ affected components. At any level, every affected vertex must be live in either $F$ or $F'$, so $\mathcal{A}^i_j = \mathcal{A}^i_{F,j} \cup \mathcal{A}^i_{F',j}$, and hence

$$|\mathcal{A}^i| \leq \sum_{j=1}^{6k}\left(|\mathcal{A}^i_{F,j}| + |\mathcal{A}^i_{F',j}|\right) \leq 6k \times 26 \times 2 = 312k,$$

which completes the proof. □

We can conclude that given an update of $k$ edges, the number of affected vertices at each level of the algorithm is $O(k)$.

***Putting it all together.*** Given the series of lemmas above, we now have the power to analyze the performance of the update algorithm.

Theorem 4.15 (Update performance). *A batch update consisting of $k$ edge insertions or deletions takes $O\left(k \log\left(1 + n/k\right)\right)$ work and $O(\log n \log\log k)$ span.*

Proof. The update algorithm performs work proportional to the number of affected vertices at each level. Consider separately the work performed processing the levels up to and including level $r = \log_{6/5}(1 + n/k)$. By Lemma 4.14, there are $O(k)$ affected vertices per level, so the work performed on levels up to including $r$ is

$$O(kr) = O\left(k \log\left(1 + \frac{n}{k}\right)\right).$$

By Corollary 4.4, after $r$ rounds, there are at most $k$ vertices alive in $F_r$ or $F'_r$. The number of affected vertices is at most the number of live vertices in either forest, and hence at most $2k$. The amount of affected vertices in all subsequent levels is therefore at most

$$\sum_{i=0}^{\infty}\left(\frac{5}{6}\right)^i 2k = \frac{2k}{1-\frac{5}{6}} = 12k,$$

and hence the remaining work is $O(k)$. Therefore the total work across all rounds is at most

$$O\left(k \log\left(1 + \frac{n}{k}\right)\right) + O(k) = O\left(k \log\left(1 + \frac{n}{k}\right)\right).$$

In each round, it takes $O(\log\log k)$ span to find a maximal independent set of the affected vertices and to perform the approximate compaction required to filter out the vertices that are no longer affected in the next round. Therefore, over $O(\log n)$ rounds, this results in $O(\log n \log\log k)$ span. □

# 5 A LOW-SPAN RANDOMIZED ALGORITHM

The algorithm we have presented in this paper is somewhat generic. As presented it makes use of an MIS of affected vertices since this is convenient for determinism, but one could substitute the MIS algorithm for any other independent set, provided that it satisfied the equivalents of Lemma 4.1 (contracting a constant fraction of the vertices) and Lemma 4.14 (only a small number of affected vertices), and still obtain a correct and efficient algorithm.

The randomized tree contraction algorithm of Miller and Reif [44] which is used as the key ingredient in the randomized RC-Tree algorithm [2, 3] indeed satisfies both of these properties (as is required by the algorithm of Acar et al. [2]), and hence the algorithm presented in this paper would be suitable for the randomized variant as well (of course it would no longer be deterministic). The major upside of the randomized algorithm is that computing the independent set takes just $O(1)$ span rather than $O(\log\log n)$ span, which is one of the two span bottlenecks of the algorithm. Instead of guaranteeing that the number of vertices eliminated is at least $1/6^{\text{th}}$ of them, it eliminates at least $1/8^{\text{th}}$ of the vertices w.h.p. [2, 44]

The downside of the original randomized algorithm of Acar et al. [2] is that it is based on self-adjusting computation and hence it is difficult to optimize. Our algorithm in this paper on the other hand is a direct implementation of dynamic tree contraction and hence is much more amenable to optimizations. The remaining bottleneck of the randomized algorithm is the span of performing approximate compaction on each level to eliminate the contracted vertices, which costs $O(\log^* n)$. Here, we present a technique to eliminate this span overhead. The same techniques could be applied to the deterministic algorithm, but in that case, the span of deterministically computing an MIS remains as the bottleneck.

## 5.1 A lower span static algorithm

The basic static algorithm uses approximate compaction after each round to filter out the vertices that have contracted. This is important, since without this step, every round would take $\Theta(n)$ work, for a total of $\Theta(n \log n)$ work. This leads to the deterministic $O(n)$ work and $O(\log n \log\log n)$ span algorithm, or the randomized $O(n)$ work and $O(\log n \log^* n)$ span algorithm using randomized approximate compaction [27], which has $O(\log^* n)$ span w.h.p. We can improve the span by splitting the algorithm into two phases.

***Phase One.*** Note that the purpose of compaction is to avoid performing wasteful work on dead vertices each round. However, if the forest being contracted has just $O(n/\log n)$ vertices, then a "wasteful" algorithm which avoids performing compaction takes at most $O(n)$ work anyway. So, the strategy for phase one is to contract the forest to size $O(n/\log n)$, which, by Corollary 4.3 takes at most $O(\log\log n)$ rounds (w.h.p. with the randomized algorithm). This is essentially the same strategy used by Gazit, Miller, and

Teng [24]. The work of the first phase is therefore $O(n)$ in expectation and the span, using randomized approximate compaction, is just $O(\log\log n \log^* n)$ w.h.p.

***Phase Two.*** In the second phase, we run the "wasteful" algorithm, which is simply the original algorithm but without performing any compaction. Since the forest begins with $O(n/\log n)$ vertices in this phase, this takes $O(n)$ work in expectation and completes in $O(\log n)$ additional rounds w.h.p. The algorithm no longer needs to pay for compaction, and hence the span is just $O(\log n)$ w.h.p. Putting these together, the total work of both phases is $O(n)$, and the span w.h.p. is $O(\log\log n \log^* n) + O(\log n) = O(\log n)$.

## 5.2 A lower span dynamic algorithm

We optimize the dynamic algorithm similarly to the static algorithm, by splitting it into three phases this time.

***Phase One.*** The algorithm runs Phase One for $\log_{8/7}(1 + n/k)$ rounds (the $8/7$ comes from the randomized independent set which eliminates $1/8^{\text{th}}$ of the remaining vertices w.h.p. [2]). Note importantly that this depends on the batch size $k$, so the number of rounds each phase runs is not always the same for each update operation.

Similarly to the optimized static algorithm without concurrent writes, we attack the problem by splitting the affected vertices into groups. Specifically, we will group the affected vertices into *affected components* based on their origin vertex as defined in Section 4. There are $O(k)$ affected components, each of which is initially a singleton defined by an affected vertex at round 0.

In each round, the algorithm processes each affected component and each affected vertex within in parallel. At the end of the round, the newly affected vertices are identified for each component. To tiebreak, and ensure that only one copy of an affected vertex exists, if multiple vertices spread to the same vertex, only the one with the lowest identifier adds the newly affected vertex to its component. Since the forest has constant degree, the one with the lowest identifier can be identified in constant time.

Given the set of affected vertices, new and old, we can then filter each component independently in parallel to remove vertices that are no longer affected in the next round. The critical insight is that according to Lemma 4.13 (or its equivalent in the randomized algorithm, Lemma 23 of [2]), each affected component has *constant size* (w.h.p.), so this filtering takes constant work and span w.h.p.

Having to maintain this set of $k$ affected components adds an additional $O(k)$ work to each round, but since we run Phase One for only $O(\log(1 + n/k))$ rounds, this is still work efficient.

***Phase Two.*** Using the randomized contraction algorithm, by the time Phase Two begins, the forest will have contracted to the point that at most $k$ vertices remain w.h.p. From this point onwards, we use an algorithm very similar to the static algorithm to complete the remaining rounds, and thus split into two more phases. First, we can collect the contents of each of the $O(k)$ affected components back into a single array of $O(k)$ affected vertices w.h.p. This can be done in at most $O(k)$ work and $O(\log k)$ span w.h.p.

Given the affected vertices, we logically partition them into $k/\log k$ groups of size $O(\log k)$ w.h.p. We then run the basic dynamic update algorithm for $\log\log k$ rounds, using a filter algorithm (not approximate compaction) at each round to remove vertices that are no longer affected. The span of this phase is therefore $O((\log\log k)^2)$ w.h.p, and costs at most $O(k)$ additional work w.h.p.

***Phase Three.*** After completing Phase Two, there will be at most $O(k/\log k)$ vertices alive in the forest w.h.p, and hence at most twice that many affected vertices (affected vertices may be alive in either the new or old forest). Phase Three simply collects the remaining affected vertices and performs the same steps as Phase One. We create up to $O(k/\log k)$ singleton affected components, and then in each round, process each vertex in each component in parallel, then spread to any newly affected vertices. Each affected component remains constant size w.h.p. and the work performed in each round is at most $O(k/\log k)$ for $O(\log k)$ rounds w.h.p., adding to a total of $O(k)$ work. Since each affected component is constant size w.h.p, maintaining them takes constant time w.h.p. After $O(\log k)$ rounds w.h.p, the forest is fully contracted.

In total, at most $O(k\log(1 + n/k))$ additional work is added, so the algorithm is still work efficient. The span of Phases One and Three is constant per round, and hence the total span w.h.p. is

$$O\left(\log\left(1 + \frac{n}{k}\right) + (\log\log k)^2 + \log(k)\right) = O(\log n).$$

# 6 CONCLUSION

We presented the first deterministic work-efficient parallel algorithm for the batch-dynamic trees problem. We showed that parallel RC-Trees [2, 4] can be derandomized using a variant of parallel tree contraction that contracts a maximal independent set of degree one and two vertices. Our algorithm performs $O(k\log(1 + n/k))$ work for a batch of $k$ updates and runs in $O(\log n \log\log k)$ span. We also improve the span of the randomized variant of the algorithm from $O(\log n \log^* n)$ to just $O(\log n)$ w.h.p.

Several interesting questions still remain open. Our deterministic algorithm requires $O(\log n \log\log k)$ span, while our randomized variant requires just $O(\log n)$. Can we obtain a deterministic algorithm with $O(\log n)$ span? It seems unlikely that the exact algorithm that we present here could be optimized to that point, since that would imply finding a maximal independent set in $O(1)$ span work efficiently, and the fastest known algorithms run in $O(\log^* n)$ span but are not even work efficient. This doesn't rule out using other techniques instead of a maximal independent set, however.

Lastly, it would be interesting to explore which other parallel dynamic graph problems can be derandomized, either using our deterministic RC-Trees as an ingredient, or independently.

# ACKNOWLEDGMENTS

We thank the reviewers for their feedback and for spotting a bug in an earlier version. This work was supported by the National Science Foundation grants CCF-2119352 and CCF-1919223.

# REFERENCES


[1] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. 2019. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[2] Umut A Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-dynamic Trees via Change Propagation. In *European Symposium on Algorithms (ESA)*.

[3] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[4] Umut A Acar, Guy E Blelloch, and Jorge L Vittes. 2005. An experimental analysis of change propagation in dynamic trees. In *Algorithm Engineering and Experiments (ALENEX)*.

[5] Ravindra K Ahuja, James B Orlin, and Robert E Tarjan. 1989. Improved time bounds for the maximum flow problem. *SIAM J. on Computing* 18, 5 (1989), 939–954.

[6] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2005. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms (TALG)* 1, 2 (2005), 243–264.

[7] Daniel Anderson. 2023. *Parallel Batch-Dynamic Algorithms.* Ph. D. Dissertation. Department of Computer Science, Carnegie Mellon University.

[8] Daniel Anderson and Guy E Blelloch. 2021. Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[9] Daniel Anderson, Guy E Blelloch, and Kanat Tangwongsan. 2020. Work-Efficient Batch-Incremental Minimum Spanning Trees with Applications to the Sliding-Window Model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[10] Guy E. Blelloch. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann.

[11] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (March 1996).

[12] Guy E. Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[13] Richard P Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

[14] Richard Cole and Uzi Vishkin. 1986. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.

[15] Richard Cole and Uzi Vishkin. 1986. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control* 70, 1 (1986), 32–53.

[16] Richard Cole and Uzi Vishkin. 1991. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and computation* 92, 1 (1991), 1–47.

[17] Paolo Ferragina and Fabrizio Luccio. 1994. Batch dynamic algorithms for two graph problems. In *International Conference on Parallel Architectures and Languages Europe*.

[18] Paolo Ferragina and Fabrizio Luccio. 1996. Three techniques for parallel maintenance of a minimum spanning tree under batch of updates. *Parallel processing letters* 6, 02 (1996), 213–222.

[19] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *ACM Symposium on Theory of Computing (STOC)*.

[20] Greg N Frederickson. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. on Computing* 14, 4 (1985), 781–798.

[21] Greg N Frederickson. 1997. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. on Computing* 26, 2 (1997), 484–538.

[22] Greg N Frederickson. 1997. A data structure for dynamically maintaining rooted trees. *J. Algorithms* 24, 1 (1997), 37–65.

[23] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2020. Minimum Cut in $O(m \log^2 n)$ Time. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*.

[24] Hillel Gazit, Gary L Miller, and Shang-Hua Teng. 1988. Optimal tree contraction in the EREW model. In *Concurrent Computations*. Springer, 139–156.

[25] Barbara Geissmann and Lukas Gianinazzi. 2018. Parallel minimum cuts in near-linear work and low depth. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[26] Mohsen Ghaffari, Christoph Grunau, and Jiahao Qu. 2023. Nearly Work-Efficient Parallel DFS in Undirected Graphs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[27] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.

[28] Andrew Goldberg, Serge Plotkin, and Gregory Shannon. 1987. Parallel symmetry-breaking in sparse graphs. In *ACM Symposium on Theory of Computing (STOC)*.

[29] Andrew V Goldberg, Michael D Grigoriadis, and Robert E Tarjan. 1991. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Mathematical Programming* 50, 1 (1991), 277–290.

[30] Andrew V Goldberg and Serge A Plotkin. 1987. Parallel (Δ+ 1)-coloring of constant-degree graphs. *Inf. Process. Lett.* 25, 4 (1987), 241–245.

[31] Andrew V Goldberg and Robert E Tarjan. 1988. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)* 35, 4 (1988), 921–940.

[32] Tal Goldberg and Uri Zwick. 1995. Optimal deterministic approximate parallel prefix sums and their applications. In *Proceedings of the Third Israel Symposium on the Theory of Computing and Systems*.

[33] Monika Rauch Henzinger and Valerie King. 1995. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *ACM Symposium on Theory of Computing (STOC)*. ACM.

[34] Monika R Henzinger and Valerie King. 2001. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. on Computing* 31, 2 (2001), 364–374.

[35] Jacob Holm and Kristian de Lichtenberg. 1998. *Top-trees and dynamic graph algorithms.* Master's thesis. Citeseer.

[36] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760.

[37] J. JaJa. 1992. *Introduction to Parallel Algorithms.* Addison-Wesley Professional.

[38] Hermann Jung and Kurt Mehlhorn. 1988. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Inf. Process. Lett.* 27, 5 (1988), 227–236.

[39] Bruce M Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[40] David R Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76.

[41] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems.

[42] Michael Luby. 1985. A simple parallel algorithm for the maximal independent set problem. In *ACM Symposium on Theory of Computing (STOC)*.

[43] Daniele Micciancio. 1997. Oblivious data structures: applications to cryptography. In *ACM Symposium on Theory of Computing (STOC)*.

[44] Gary L. Miller and John H. Reif. 1985. Parallel Tree Contraction and Its Application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE.

[45] Shaunak Pawagi and Owen Kaser. 1993. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica* 9, 4 (1993), 357–381.

[46] Seth Pettie and Vijaya Ramachandran. 2002. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. on Computing* 31, 6 (2002), 1879–1895.

[47] Xiaojun Shen and Weifa Liang. 1993. A parallel algorithm for multiple edge updates of minimum spanning trees. In *International Parallel Processing Symposium (IPPS)*.

[48] Daniel D Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci.* 26, 3 (1983), 362–391.

[49] Robert E Tarjan. 1997. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming* 78, 2 (1997), 169–177.

[50] Robert E Tarjan and Renato F Werneck. 2005. Self-adjusting top trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[51] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. 2019. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 92–106.

[52] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Minimum Spanning Forest and the Efficiency of Dynamic Agglomerative Graph Clustering. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[53] Renato F Werneck. 2006. *Design and analysis of data structures for dynamic trees.* Ph. D. Dissertation. Princeton University.