# Parallel Batch-Dynamic Algorithms

**Dynamic Trees, Graphs, and Self-Adjusting Computation**

*Daniel Anderson*

CMU-CS-23-120

June 2023

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the National Science Foundation under award numbers 1408940, 1910030, 1901381, and 2119352. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

**Keywords:** Dynamic algorithms, Parallel algorithms, Graph algorithms

## Abstract

The defining feature of many modern large-scale computer systems is the sheer amount of data that they generate and process. Google's MapReduce clusters process over twenty petabytes of data *per day*, and Facebook has reported processing over 4 petabytes of data per day and running over one million map reduce computations over this data. Modern systems for processing this kind of data make extensive use of parallel and distributed computation to achieve the necessary level of throughput. An important property of the datasets involved in such computations, however, is that they are not static. Rather, they are frequently *evolving*. Large social networks, for example, undergo rapid changes; users are added, and links between uses are created and deleted rapidly, but each such update affects a relatively tiny portion of the data.

This thesis explores a relatively under-explored area of algorithm design intended to tackle these kinds of problems: *parallel batch-dynamic algorithms*. Classic algorithms for handling dynamic data support a single update at a time, but this model is insufficient for handling the scale of modern rapidly changing datasets, and furthermore yields little room to exploit parallelism within the updates. A *batch-dynamic* algorithm consumes multiple updates at a time, which allows for increased throughput and more opportunities for parallelism.

In the first part of the thesis, we will design algorithms for parallel batch-dynamic trees based on sequential Rake-Compress Trees (RC Trees). In the second part of this thesis, we will design algorithms for batch-dynamic graph connectivity and batch-incremental minimum spanning trees. Then, in Part Three, we show that batch-dynamic algorithms can be used as ingredients in designing highly-efficient parallel static algorithms. Using our parallel Rake-Compress Tree data structure as an ingredient, we obtain the first ever work-efficient parallel algorithm for minimum cuts.

The final part of this thesis will explore the implementation of systems for processing batch-dynamic data. One factor that hampers the adoption of efficient dynamic algorithms in practice is that they are often extremely complicated and difficult to implement. To address this issue in the sequential setting, *self-adjusting computation* is a technique for automatically dynamizing static algorithms to create dynamic ones. This lowers the implementation burden on programmers. In this thesis, we show that self-adjusting computation can be generalized to automatically dynamize parallel algorithms, allowing programmers to reap the benefits of parallel batch-dynamic algorithms without the burden of implementing them from scratch.

## Acknowledgments

First and foremost, this thesis would not have been possible without the excellent advising of Professor Guy Blelloch. I came to CMU in 2018 not knowing a single thing about parallel algorithms, so the fact that I made it this far is a testament to his patience, knowledge, and encouragement. Guy was truly the ideal Ph.D. advisor; he was always available and willing to help with anything, has an encyclopedic knowledge of parallel algorithms, and allowed me to pursue whatever I was interested in. In particular, he was very supportive of my desire to seek out extra teaching responsibilities in preparation for a teaching-focused career.

Throughout my candidature, I have also been lucky enough to work with many other wonderful collaborators. Thanks to Pierre Le Bodic, Gregor Hendel, Merlin Viernickel, Umut Acar, Laxman Dhulipala, Sam Westrick, Kanat Tangwongsan, Marc Pfetsch, Ticha Sethapakdi, Stefanie Mueller, Hao Wei, Mike Rainey, Yihan Sun, and Magdalen Dobson. Many extra thanks are owed to Jérôme Droniou and Pierre Le Bodic at Monash University who mentored me in research before I applied to graduate school. Without their mentorship, support, and the opportunities they provided me, I would not have started a Ph.D. let alone completed one. The members of my thesis committee also deserve a great deal of gratitude for being willing to read this thesis and for providing useful feedback on improving it. Julian, Valerie, Phil, and Danny, thank you for your advice and contributions.

To Danny Sleator, David Woodruff, Elaine Shi, and all of the many TAs that I worked with over the years, thank you for allowing me to help teach 15-451 and for helping me achieve my dream of becoming a teaching professor. It was a dream come true to be able to work on such a great course and I am beyond excited that I will get to continue doing so.

To my friends that I have lived with over the past five years, thank you for finding us a place to call home and helping me navigate this strange new country. Siva, David, Arjun, Jalani, Victor, and Sam, you made Pittsburgh feel like home. Thanks to David Wajc for inviting me to share an office with you and for valiantly defending it against attempts to add another desk to it. Thanks to Goran Zuzic for being my gym buddy and making me wake up at 7 am for it, which would not have happened otherwise. An especially big thank you to everyone involved in the SCS Graduate Student Musicals over the years. They were some of the most fun experiences of my time at CMU. To all of my friends back home in Australia, thank you for welcoming me back with open arms every time I came to visit, even if it wasn't as often as I hoped.

To my mother and aunt, Diane and Jill, thank you for being willing to travel ten thousand miles to visit me. Last but not least, thank you to my wonderful girlfriend Qianou. You helped to turn what should have been the most stressful year of my life, in which I taught a class while applying for faculty jobs while writing this thesis, into the most exciting and happy year of my life.

# Contents

# Chapter 1
# Introduction and Overview

## 1.1 Introduction

The defining feature of many modern large-scale computer systems is the sheer amount of data that they generate and process. In 2008, it was reported that Google's MapReduce clusters process over twenty petabytes of data *per day*. Facebook has reported processing over 4 petabytes of data per day and running over one million map reduce computations over this data. The dawning of the era of "big data" led to countless systems being developed that attempt to make processing information of this scale feasible. Such examples include MapReduce [49], Dryad [105], Pregel [128], and Dremel [131], among many more. These systems make extensive use of parallel and distributed computing to achieve high throughput. A branch of theory has also emerged that attempts to study algorithms suitable for the MapReduce paradigm, called the Massively Parallel Computation (MPC) model [119].

These systems are primarily designed for processing *static*, i.e., unchanging data; any update to the dataset will require desired queries and analyses to be re-ran from scratch. However, most of today's largest datasets are not static, but are frequently *evolving*. Large social networks and the web hyperlink graph are two ubiquitous examples of massive data sets that undergo rapid changes. Motivated by the dynamic nature of large data sets, researchers and practitioners have worked on extending these systems with support for handling dynamic updates to the data. Examples include MapReduce Online [43], Incoop [23], Nectar [85], and Naiad [136]. Although these systems exhibit strong empirical performance, the theoretical properties of such systems and the algorithms behind them remain underexplored.

In this thesis, we will apply modern theoretical techniques for parallel computing to the design, analysis, and implementation of parallel systems for processing large-scale dynamic datasets, with strong guarantees on their performance. The key idea that will underpin our efforts is that of *batch-dynamic algorithms*, which, unlike classic dynamic algorithms, process updates to a dataset in *batches*, which both has the potential to reduce the work and to create opportunities for parallelism. While early work in the literature has studied parallel batch-dynamic algorithms (e.g., [61, 62, 109, 141, 142, 159]), these algorithms all perform work that is polynomial in the size of the total dataset, regardless of the size of the updates. In our work, we focus on designing *work-efficient* parallel batch-dynamic algorithms, which perform work that is polynomial in the update size and just polylogarithmic in the total size of the data. We will also see how batch-dynamic algorithms can be used as a tool to design efficient algorithms for static problems.

At a high level, this thesis will explore four main areas. First, we will design parallel batch-dynamic algorithms for dynamic trees (Chapter 3), with both randomized (Chapter 4) and deterministic (Chapter 5) updates. Then, we will apply dynamic trees to solve two fundamental graph problems in the batch-dynamic setting, fully dynamic graph connectivity (Chapter 6) and incremental minimum spanning trees (Chapter 7). Thirdly, we will apply our data structure for batch-dynamic trees to the classic minimum cut problem, obtaining the first ever work-efficient parallel algorithm for the problem (Chapters 8–9). Lastly, we will implement a system for processing dynamic data sets in parallel, where we design the first theoretically efficient system for parallel self-adjusting computation (Chapter 10).

## 1.2 Overview of Results

This thesis presents several results on parallel batch-dynamic algorithms, with applications to the minimum cut problem, streaming algorithms, and self-adjusting computation. Here, we give an overview of the problems we solve and a summary of our primary results.

### 1.2.1 Parallel Rake-Compress Trees

The dynamic trees problem, first posed by Sleator and Tarjan [163] is to maintain a forest of trees subject to the insertion and deletion of edges, also known as *links* and *cuts*. Dynamic trees are used as a building block in a multitude of applications, including maximum flows [171], dynamic connectivity and minimum spanning trees [63], and minimum cuts [114], making them a fruitful line of work with a rich history.

There are many efficient ($O(\log n)$ time per operation) dynamic tree algorithms, including Sleator and Tarjan's *link/cut tree* [163, 164], Henzinger and King's *Euler-Tour Trees* [96], Frederickson's *topology trees* [63, 64, 65], Holm and de Lichtenberg's *top trees* [16, 98, 167], and Acar et al.'s Rake-Compress Trees (RC-Trees) [4, 5]. Most of these algorithms are sequential and handle single edge updates at a time.

In addition to handling updates, a dynamic tree algorithm should support some kinds of *queries*. Examples of queries include asking whether two vertices are connected (is there a path between them in the forest), asking for the sum of the edge weights in a given subtree, asking for the minimum weight edge on a given path, or asking for the lowest common ancestor (LCA) of two vertices.

Since they already perform such little work, there is often little to gain by processing single updates in parallel, hence parallel applications often process *batches* of updates. We are therefore concerned with the design of parallel *batch-dynamic* algorithms. Tseng et al. [174] develop *Batch-parallel Euler-tour trees*, which can maintain an Euler-tour tree subject to batch updates. A comparison of existing dynamic tree algorithms is given in Table 1.1.

Previous implementations of RC-Trees are all based on applying self-adjusting computation to randomized parallel tree contraction [4, 5], i.e., a static tree contraction algorithm is implemented in a framework that automatically tracks changes to the input values, and selectively recomputes procedures that depend on changed data. This results in the RC-Tree that would have been obtained if running the static algorithm from scratch on the updated data. Our work makes two contributions to bring RC-Trees into the parallel setting.

| | Parallel Operations | | Queries Supported | | |
|---|---|---|---|---|---|
| | Updates | Queries | Path | Subtree | Nonlocal |
| Link/cut trees | | | ✓ | | |
| (Parallel) Euler-tour trees | ✓✓ | ✓✓ | | ✓✓ | |
| Top trees | | ✓ | ✓ | ✓ | ✓ |
| Rake-Compress trees | | ✓ | ✓ | ✓ | ✓ |
| Parallel Rake-Compress trees (our work) | ✓✓ | ✓✓ | ✓✓ | ✓✓ | ✓✓ |

**Table 1.1: The known capabilities of various dynamic tree algorithms. Nonlocal queries are operations such as computing LCAs and diameters. Single ticks indicate that the operation is supported and takes $O(\log n)$ time. Double ticks indicate that the data structure supports a batch of $k$ operations in $O\left(k \log\left(1+\frac{n}{k}\right)\right)$ work and $O(\log n)$ span.**

## Randomized Parallel RC-Trees

We propose a framework for parallel self-adjusting computation that algorithmically dynamizes a certain class of static parallel algorithms to obtain efficient parallel batch-dynamic algorithms. We start by defining the notion of a *round-synchronous* algorithm. A *round-synchronous* algorithm consists of a sequence of rounds, where a round executes in parallel across a set of processes, and each process runs a sequential *round computation* reading and writing from shared memory and doing local computation.

Dynamization works by running the round-synchronous algorithm while tracking all write-read dependencies—i.e., a dependence from a write in one round to a read in a later round. Then, whenever a batch of changes are made to the input, *change propagation* propagates the changes through the original computation, only rerunning round computations if the values they read have changed. Depending on the algorithm, changes to the input could drastically change the underlying computation, introducing new dependencies, or invalidating old ones. To capture the cost of running the change propagation algorithm for a particular parallel algorithm and class of updates we define a *computational distance* between two computations, which corresponds to the total work of the round computations that differ in the two computations.

Our dynamic trees algorithm is a parallel version of the sequential RC-Tree data structure obtained by applying our change propagation algorithm to the parallel tree contraction algorithm of Miller and Reif [132]. This approach generalizes the sequential RC-Tree algorithm to allow for batches of edge insertions or deletions, work efficiently in parallel. A challenge, and one of our main contributions, is in analyzing the computation distance incurred by batch updates in the parallel batch-dynamic setting.

**Prior state of the art** One can maintain a batch-dynamic Euler-tour tree of a dynamic tree on $n$ vertices subject to $k$ edge insertions or $k$ edge deletions in $O\left(k \log\left(1+\frac{n}{k}\right)\right)$ work in expectation and $O(\log n)$ span w.h.p. [174].

**Theorem 1** (Randomized RC-Trees)**.** There is a randomized parallel batch-dynamic algorithm that maintains a balanced RC-Tree of a bounded-degree forest subject to batches of $k$ edge updates (insertions, deletions, or both) in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work in expectation and $O(\log n)$ span w.h.p.

The update bounds of our data structure match those of Euler-tour trees, but we will see below that RC-Trees can support substantially more queries in the same work and span.

## Deterministic Parallel RC-Trees

We describe a deterministic direct update algorithm for parallel RC-Trees that is not based on self-adjusting computation. To do so, we describe a variant of tree contraction that deterministically contracts a maximal independent set of degree one and two vertices each round. When an update is made to the forest, we identify the set of *affected vertices* (this part is similar to change propagation) and then greedily update the tree contraction by computing a maximal independent set of affect vertices and updating the contraction accordingly. The key insight is in carefully establishing the criteria for vertices being affected such that the update is correct, and bounding the number of such vertices so that it is efficient. This results in the first deterministic implementation of RC-Trees, either sequential or parallel. This result is parallelisable and achieves the same work-efficient update bounds as the randomized algorithm, at the cost of a slightly higher span.

**Prior state of the art** All efficient parallel algorithms for the dynamic trees problem were randomized, and all implementations of RC-Trees were randomized. Sequentially, one can support dynamic trees deterministically using Sleator and Tarjan's *link/cut tree* [163, 164], Henzinger and King's *Euler-tour trees* [96], Frederickson's *topology trees* [63, 64, 65], and Holm and de Lichtenberg's *top trees* [16, 98, 167] in $O(\log n)$ time per operation.

**Theorem 2** (Deterministic RC-Trees)**.** There is a deterministic parallel batch-dynamic algorithm that maintains a balanced RC-Tree of a bounded-degree forest subject to batches of $k$ edge updates (insertions, deletions, or both) in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work and $O\left(\log(n)\log^{(c)} k\right)$ span for any constant $c$.

## Batch queries on RC-Trees

RC-Trees have been shown to support a large range of queries including connectivity, path queries, subtree queries, lowest common ancestors (LCAs), tree diameter, tree center, and nearest marked vertex, all in $O(\log n)$ time [5]. A subtree query sums the weights of every vertex/edge in a given subtree with respect to some associative and commutative operation. Similarly, a path query sums the weights of every vertex/edge on a given path with respect to some associative and commutative operation.

With batch-dynamic RC-Trees comes the opportunity to also support *batch queries* which solve a batch of queries in parallel in less work than solving each one individually.

**Prior state of the art** Batch-parallel Euler-tour trees [174] support batches of $k$ connectivity queries or $k$ subtree queries in $O\left(k \log\left(1+\frac{n}{k}\right)\right)$ work in expectation and $O(\log n)$ span w.h.p. Sequentially, RC-Trees can evaluate single connectivity queries, subtree queries, path queries, diameters, centers, medians, LCAs, and nearest marked vertices in $O(\log n)$ time [5].

**Theorem 3** (Batch queries)**.** A balanced RC-Tree can be augmented to support the following operations in $O\left(k + k \log\left(1+\frac{n}{k}\right)\right)$ work (in expectation if the RC-Tree is randomized) and $O(\log n)$ span (w.h.p. if the RC-Tree is randomized) for a batch of $k$ queries

- Batch connectivity queries
- Batch subtree queries over vertex/edge weights from any commutative semigroup
- Batch path queries over vertex/edge weights from any commutative group
- Batch path queries for the lightest or heaviest edge on a path
- Batch LCA queries with respect to any possible root of the underlying tree
- Batch nearest marked vertex queries

Note that we do not prove results for batch diameters, centers, or medians, because those queries do not benefit from batching in any way since they are properties of their entire tree, and hence there is only one possible query and answer for any tree. Our results therefore provide batched versions of all known RC-Tree applications that are amenable to batching.

### 1.2.2 Parallel Batch-Dynamic Graphs

Countless papers have been written on dynamic algorithms, i.e. algorithms that can efficiently update their output when given an update to their input. Far less explored is the intersection between dynamic algorithms and parallel algorithms. In principle, the two should be highly compatible; algorithms suitable for parallelization tend to be those that can be broken into composable independent pieces, as this facilitates parallelism. Similarly, algorithms that can be decomposed into many independent pieces are often prime candidates for dynamization, since, if a small change is made to the input, it is likely that only a small number of the indpendent pieces are affected, removing the need to re-run the entire computation. There remains, however, one problem: if a small change is made to the input to an efficient dynamic algorithm, there is unlikely to be enough work to distribute among multiple processors to take full advantage of parallelism.

An elegant and highly practical approach to overcome this limitation is to consider updates in *batches*. Instead of responding to individual updates one at a time like classical dynamic algorithms, *batch-dynamic* algorithms respond to a batch of updates all at once. The concept of parallel batch-dynamic algorithms itself is not quite new, and a few works exist, particularly on graph problems such as minimum spanning trees [61]. These algorithms, however, typically targeted the classic PRAM model of computation, with the goal of minimizing the parallel time of the computation, paying no attention to the work, or equivalently, the number of processors required to perform the work. Such algorithms tend to be less practical than those that consider both the *work and span* of the algorithm [25].

*Work-efficient* parallel algorithms are those that asymptotically perform no more work than their sequential counterparts, and serve as a good design guideline for algorithms that should perform well in practice [25] and use less energy. The focus of this part of the thesis is thus on developing *work-efficient parallel batch-dynamic algorithms*. That is, given a sequential dynamic algorithm on a problem of size $n$ that processes a single update in time $O(f(n))$, we aim to show that a parallel batch-dynamic algorithm can process a batch of updates in at most $O(k\ f(n))$ work and $O(\text{polylog}\, n)$ span. We show several elegant results in which the total work of a batch-dynamic algorithm can even be asymptotically less than $O(k\ f(n))$ by eliminating redundancies that would be present if simply performing the single-update algorithm $k$ times.

In this thesis, we will design parallel batch-dynamic algorithms for two of the most fundamental and well studied graph problems: connectivity and minimum spanning trees.

## Parallel batch-dynamic connectivity

Computing the connected components of a graph is a fundamental problem that has been studied in many different models of computation [14, 17, 100, 155, 160, 168]. The *connectivity problem* takes as input an undirected graph $G$ and requires an assignment of labels to vertices such that two vertices have the same label if and only if they are in the same connected component. The dynamic version of the problem requires maintaining a data structure over an $n$ vertex undirected graph that supports insertions and deletions of edges, and queries of whether two vertices are in the same connected component. Despite the large body of work on the dynamic connectivity problem over the past two decades [55, 95, 96, 100, 103, 111, 120, 139, 172, 173, 185, 186], little is known about batch-dynamic connectivity algorithms that process *batches of queries and updates*, either sequentially or in parallel.

The starting point of our algorithm is the classic Holm, de Lichtenberg and Thorup (HDT) dynamic connectivity algorithm [100]. Like nearly all existing dynamic connectivity algorithms, the HDT algorithm maintains a spanning forest certifying the connectivity of the graph. The algorithm maintains a set of $\log n$ nested forests under two carefully designed invariants. The forests are represented an Euler tour (ET) tree [96, 135].

The main challenge in a dynamic connectivity algorithm is to efficiently find a *replacement edge*, or a non-tree edge going between the two disconnected components after deleting a tree edge. The key idea of the HDT algorithm is to organize the spanning-forest of the graph into $\log n$ levels of trees. The top-most level of the structure stores a spanning forest of the entire graph, and each level contains all tree-edges stored in levels below it. The algorithm ensures that the largest size of a component at level $i$ is $2^i$. Using these invariants, the algorithm is able to cleverly search the tree edges so that each non-tree edge is examined at most $\log n$ times as a candidate replacement edge. The main idea is to store each non-tree edge at a single level (initially the top-most level), and push the edge to a lower level each time it is unsuccessfully considered as a replacement edge. Since there are $\log n$ levels, and the cost of discovering, processing, and removing an edge from each level using ET-trees is $O(\log n)$, the amortized cost of the HDT algorithm is $O(\log^2 n)$ per edge operation.

A challenge, and sequential bottleneck in the HDT algorithm is the fact that it processes each non-tree edge one at a time—a property which is crucial for achieving good amortized bounds. Aside from hindering parallelism, processing the edges one at a time eliminates any potential for improved batch bounds, since finding the representative of the endpoints of an edge costs $O(\log n)$ time per query. Therefore, to obtain an efficient batch or parallel algorithm we must examine batches of *multiple* non-tree edges at a time, while also ensuring that we do not perform extra work that cannot be charged to level-decreases on an edge. Our approach is to use a *doubling technique*, where we examine sets of non-tree edges with geometrically increasing sizes.

Another challenge is that processing a batch of deletions can shatter a component into multiple disconnected pieces. Since the HDT algorithm deletes at most a single tree edge per deletion operation, it handles exactly two disconnected pieces per level. In contrast, since we delete batches of edges in our batch-dynamic algorithm, we may have many disconnected pieces at a given level, and must search for replacement edges reconnecting these pieces. Our algorithm searches for a replacement edge from each piece that is small enough to be pushed down to the next lower level.

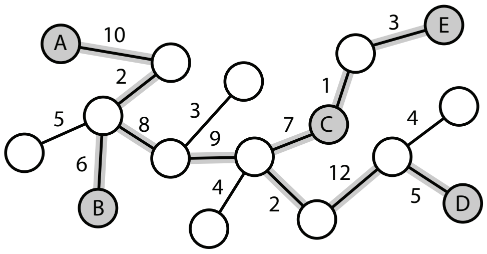
**Prior state of the art** One can maintain the connected components of a dynamic graph on $n$ vertices subject to $k$ edge insertions in $O(n\log(m/n))$ work and $O(\log n \log^{(3)} n \log^{(2)}(m/n))$ span, and $k$ edge updates in $O(n\log(m/n)\min\{k,f\})$ work and $O(\log n \log^{(3)} n \log(m/n))$ span [61]. Sequentially, single edge updates can be processed in $O(\log n(\log\log n)^2)$ expected amortized time [103], $O(\log^2 n/\log\log n)$ deterministic amortized time [185], deterministically in $O\left(\sqrt{n(\log\log n)^2/\log n}\right)$ worst-case time [120], or $O(\log^4 n)$ time for worst-case randomized updates with Monte-Carlo queries (queries correct w.h.p.) [75, 182].

**Theorem 4** (Parallel batch-dynamic connectivity)**.** There is a connectivity algorithm which processes batches of $k$ edge insertions and deletions in $O\left(k\log n\log\left(1+\frac{n}{\Delta}\right)\right)$ expected amortized work where $\Delta$ is the average batch size of all deletion operations. The cost of connectivity queries is $O\left(k+k\log\left(1+\frac{n}{k}\right)\right)$ expected work and $O(\log n)$ span w.h.p. for a batch of $k$ queries. The span to process a batch of edge insertions and deletions is $O(\log n)$ and $O(\log^3 n)$ w.h.p. respectively.

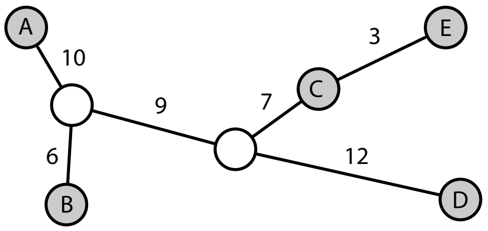
## Parallel batch-incremental minimum spanning trees

Computing the minimum spanning tree (MST) of a weighted undirected graph is a classic and fundamental problem that has been studied for nearly a century, going back to early algorithms of Borůvka [31], and Jarník [108] (later rediscovered by Prim [148] and Dijkstra [54]), and later, the perhaps more well-known algorithm of Kruskal [125]. The MST problem is, given a connected weighted undirected graph, to find a set of edges of minimum total weight that connect every vertex in the graph. More generally, the minimum spanning forest (MSF) problem is to compute an MST for every connected component of the graph. The *dynamic* MSF problem is to do so while responding to edges being inserted into and deleted from the graph. The *incremental* MSF problem is a special case of the dynamic problem in which edges are only inserted.

The key ingredient in our batch-incremental MSF data structure is a data structure for dynamically producing a *compressed path tree*. Given a weighted tree with some marked vertices, the compressed path tree with respect to the marked vertices is a minimal tree on the marked vertices and some additional "Steiner vertices" such that for every pair of marked vertices, the heaviest edge on the path between them is the same in the compressed tree as in the original tree. That is, the compressed path tree represents a summary of all possible pairwise heaviest edge queries on the marked vertices. An example of a compressed path tree is shown in Figure 1.1. More formally, consider the subgraph consisting of the union of the paths between every pair of marked vertices. The compressed path tree is precisely this subgraph but with all of the non-marked vertices of degree at most two spliced out. To produce the compressed path tree, we leverage our parallel RC-Trees from Chapter 3.



**(a) A weighted tree, with some important vertices marked (in gray). The paths between the marked vertices are highlighted.**



**(b) The corresponding compressed path tree. The edges are weighted to represent the heaviest edge on the corresponding path.**

**Figure 1.1: A weighted tree and its corresponding compressed path tree with respect to some marked vertices.**

Given a compressed path tree for each component of the graph, our algorithm follows from a generalization of the classic "cycle rule" (or "red rule") for MSTs, which states that given a heaviest edge on a cycle in a graph, there exists an MST that doesn't contain it. This fact is used to produce the efficient $O(\log(n))$ time solution to the sequential incremental MSF problem [171]. To handle a batch of edge insertions, our algorithm computes the compressed path tree with respect to their endpoints which, in a sense, generalizes the red rule, because it represents the heaviest edges on all pairwise paths, and hence all possible cycles between the newly inserted edges. More specifically, our algorithm takes the compressed path trees and inserts the new batch of edges into them, and computes the MSF of the resulting graph. For the MSF, we can use the algorithm of Cole et al. [42], which is linear work in expectation and logarithmic span w.h.p., which in turn is based on the linear time sequential algorithm [117]. Since the compressed path tree has size $O(k)$, this can be done efficiently. We then show that the edges selected by this MSF can be added to the MSF of the main graph, and those that were not selected can be removed, correctly updating the MSF.

**Prior state of the art** One can maintain a dynamic minimum spanning forest on $n$ vertices subject to $k$ edge insertions in $O(n^{2/3}(k+\log(m/n)))$ work and $O(k+\log(m/n)\log(n))$ span [45]. Sequentially, single edge insertions can be processed in $O(\log n)$ time [171].

**Theorem 5** (Batch-incremental MSF)**.** There exists a data structure that maintains the MSF of a weighted undirected graph that can insert a batch of $k$ edges into a graph with $n$ vertices in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work in expectation and $O(\log^2 n)$ span w.h.p.

## 1.2.3 Parallel Minimum Cut

The minimum cut problem is one of the most classic problems in graph theory and algorithms. The problem is to find, given an undirected weighted graph $G = (V, E)$, a nonempty subset of vertices $S \subset V$ such that the total weight of the edges crossing from $S$ to $V \setminus S$ is minimized. Early approaches to the problem were based on reductions to maximum $s$-$t$ flows [82, 92]. Several algorithms followed which were based on edge contraction [113, 116, 137, 138]. Karger was the first to observe that tree packings [140] can be used to find minimum cuts [114]. In particular, for a graph with $n$ vertices and $m$ edges, Karger showed how to use random sampling and a tree packing algorithm of Gabow [67] to generate a set of $O(\log n)$ spanning trees such that, w.h.p., the minimum cut crosses at most two edges of one of them. A cut that crosses at most $k$ edges of a given tree is called a $k$*-respecting* cut. To complete the algorithm, Karger gives an $O(m\log^2 n)$-time algorithm for finding minimum 2-respecting cuts, yielding an $O(m\log^3 n)$-time algorithm for minimum cut. Karger also gives a parallel algorithm for minimum 2-respecting cuts in $O(n^2)$ work and $O(\log^3 n)$ depth.

Until very recently, these were the state-of-the-art sequential and parallel algorithms for the weighted minimum cut problem. A new wave of interest in the problem has recently pushed these frontiers. Geissmann and Gianinazzi [71] design a parallel algorithm for minimum 2-respecting cuts that performs $O(m\log^3 n)$ work in $O(\log^2 n)$ depth. Their algorithm is based on parallelizing Karger's algorithm by replacing a sequential data structure for the so-called *minimum path* problem, based on dynamic trees, with a data structure that can evaluate a *batch* of updates and queries in parallel. Their algorithm performs just a factor of $O(\log n)$ more work than Karger's sequential algorithm, but substantially improves on the work of Karger's parallel algorithm.

Soon after, a breakthrough from Gawrychowski, Mozes, and Weimann [68] gave a randomized $O(m\log^2 n)$ algorithm for minimum cut. Their algorithm achieves the $O(\log n)$ speedup by designing an $O(m\log n)$ algorithm for finding the minimum 2-respecting cuts, which was the bottleneck of Karger's algorithm. This is the first result to beat Karger's seminal algorithm in over 20 years.

In this work, we will combine ideas from Gawrychowski et al. and Geissmann and Gianinazzi with several new techniques to close the gap between the parallel and sequential algorithms. We achieve this using a combination of results that may be of independent interest. Firstly, we design a framework for evaluating mixed batches of interleaved updates and queries on trees work efficiently in low depth. This algorithm is based on our parallel RC-Trees (Chapter 3). Roughly, we say that a set of update and query operations implemented on an RC-Tree is *simple* if the updates maintain values at the leaves that are modified by an associative operation and combined at the internal nodes, and the queries read only the nodes on a root-to-leaf path and their children. Simple operation sets include updates and queries on path and subtree weights. This result improves on Geissmann and Gianinazzi [71] who give an algorithm for evaluating $k$ path-weight updates and queries in $\Omega(k \log^2 n)$ work.

Next, we design a faster parallel algorithm for approximating minimum cuts, which is used as an ingredient in producing the tree packing used in Karger's approach. To achieve this, we design a faster sampling scheme for producing graph skeletons, leveraging recent results on sampling binomial random variables, and a transformation that reduces the maximum edge weight of the graph to $O(m \log n)$ while approximately preserving cuts.

Lastly, we show how to solve the minimum 2-respecting cut problem in parallel, using a combination of our new mixed batch tree operations algorithm and the use of RC-Trees to efficiently perform a divide-and-conquer search over the edges of the 2-constraining trees.

**Prior state of the art** The minimum cut of a graph can be computed w.h.p. in parallel with $O(m \log^4 n)$ work and $O(\log^3 n)$ span [71], or sequentially in $O(m \log^2 n)$ work [68].

**Theorem 6.** The minimum cut of a weighted undirected graph can be computed w.h.p. in $O(m \log^2 n)$ work and $O(\log^3 n)$ span.

### 1.2.4 Parallel Self-Adjusting Computation

Self-adjusting computation is an approach to automatically, or semi automatically, convert a (suitable) static algorithm to a dynamic one [1, 2, 3, 4, 36, 50, 90, 150]. Most often, self-adjusting computation is implemented in the form of a *change propagation* algorithm. The idea, roughly, is to run a static algorithm while keeping track of data dependencies. Then when an input changes (e.g. adding an edge to a graph), the change can be propagated through the computation, updating intermediate values, creating new dependencies, and updating the final output. Not all algorithms are suitable for the approach—for some, updating a single input value could propagate changes through most of the computation. To account for how much computation needs to be rerun, researchers have studied the notion of "stability" [1, 4] over classes of changes. The goal is to bound the "distance" between executions of a program on different inputs based on the distance between the inputs. For example, for an appropriate sorting algorithm adding an element to the unsorted input list ideally would cause at most $O(\log n)$ recomputation, and that recomputation could be propagated with constant overhead. This would lead to the performance of a binary search tree.

Our result on parallelising RC-Trees was achieved by designing a framework for parallel self-adjusting computation for a limited class of so-called round-synchronous algorithms. This was applied to generate efficient algorithms for batch-dynamic trees, supporting batches of links and cuts among other operations. However, the round-synchronous nature limits the applicability to algorithms that fit the model.

In this work, we develop a more general framework for supporting self-adjusting computation for arbitrary nested-parallel algorithms. We prove bounds on the cost of change propagation in the framework based on an appropriately defined distance metric. We have also implemented the framework and run experiments on a variety of benchmarks. A nested parallel program is one that is built from arbitrary sequential and parallel composition. A computation is defined recursively as either two computations that are composed in parallel (a fork), two that are composed sequentially, or the base case which is a sequential *strand*.

The crux of our technique is to represent a computation by a dependency graph that is anchored on a *Series-Parallel tree* [58], or *SP tree* for short. An SP tree corresponds to the sequential and parallel composition of binary nested parallel programs—i.e., parallel composition consists of a $P$ node with two children (the left-right order does not matter), and sequential composition consists of an $S$ node with two children (here the order does matter). The leaves are sequential strands of computation, and can just be modeled as leaf $S$ nodes. The SP tree represents the control dependencies in the program—i.e., that a particular strand needs to executed before another strand. We introduce R nodes to indicate data reads, which are used to track data dependencies between writes and reads—i.e., that a particular read depends on the value of a particular write. Together we refer to the trees as *RSP trees*. The RSP tree of a computation allows propagating a change in a way that respects sequential control dependencies while allowing parallelism where there is no dependence. We prove that a parallel change propagation algorithm can propagate changes through the computation in a manner that is both efficient and scalable.

Programs written in our framework write their inputs and any nonlocal values that depend on them into “modifiable references”, or *modifiables* for short, which track all reads to them and facilitate change propagation. Like previous work on sequential change propagation [4], we achieve our efficiency by restricting input programs to those which write to each modifiable exactly once. All race-free functional programs satisfy this restriction. Since local variables do not need to be tracked, they are not bound by this restriction, so the scope of programs amenable to our framework is not just those which are purely functional.

Roughly speaking, given two executions of the same algorithm on different inputs, we define the *computation distance* to be the work that is performed by one but not the other (see Chapter 10 for the full definition). We then show Theorem 7 which bounds the runtime of the change propagation algorithm as a function of computation distance.

**Prior state of the art** Burckhartd et al. [32] develop a general-purpose system for parallel self-adjusting computation. Their work is evaluated on a set of five benchmark problems, where it is demonstrated experimentally that the combination of parallelism and self-adjusting computation can produce both work savings and parallel time speedups. This work is purely experiential and does not provide theoretical guarantees on the runtime of updates. Hammer et al. [88] present Parallel Adaptive Language (PAL), a proposed (though not fully implemented) language for parallel self-adjusting computation. They do not provide a complete implementation of their system, or theoretical guarantees on the runtime.

**Theorem 7.** Consider an algorithm $A$, two input states $I$ and $I'$, and their corresponding RSP trees $T$ and $T'$. Let $W_\Delta = \delta(T, T')$ denote the computation distance, $R_\Delta$ denote the number of affected reads, $s$ denote the span of $A$, and $h$ denote the maximum heights of $T$ and $T'$. Then, change propagation on $T$ with the dynamic update $(I, I')$ runs in $O(W_\Delta + R_\Delta \cdot h)$ work in expectation and $O(s \cdot h)$ span w.h.p.

We have implemented the proposed techniques in a library for C++, which we call PSAC++[1] (Parallel Self-Adjusting Computation in C++). The library allows writing self-adjusting programs by using several small annotations in a style similar to writing conventional parallel programs. Self-adjusting programs can respond to changes to their data by updating their output via the built-in change propagation. Our experiments with several applications show that parallel change propagation can handle a broad variety of batch changes to input data efficiently and in a scalable fashion. For small changes, parallel change propagation can yield very significant savings in work; such savings can amount to orders of magnitude of improvement. For larger changes, parallel change propagation may save some work, and still exploit parallelism, yielding improvements due to both reduction in work and an increase in scalability.

## 1.3 Publications and Attributions

The work presented in this thesis are the fruits of much collaborative work with several collaborators. The joint publications arising from this work are listed below:

- **Parallel Batch-Dynamic Graph Connectivity** (Chapter 6)
  Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala
  The 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 19), 2019
- **Work-efficient Batch-incremental Minimum Spanning Trees with Applications to the Sliding Window Model** (Chapter 7)
  Daniel Anderson, Guy E. Blelloch, Kanat Tangwongsan
  The 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 20), 2020
- **Parallel Batch-dynamic Trees via Change Propagation** (Chapter 3–4)
  Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Sam Westrick
  The 28th Annual European Symposium on Algorithms (ESA 2020), 2020
- **Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth** (Chapter 8–9)
  Daniel Anderson, Guy E. Blelloch
  The 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 21), 2021
- **Efficient Parallel Self-Adjusting Computation** (Chapter 10)
  Daniel Anderson, Guy E. Blelloch, Anubhav Baweja, Umut A. Acar
  The 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 21), 2021

Lastly, our results on parallel batch-dynamic graph connectivity are joint work with Laxman Dhulipala, and also appear in his PhD thesis:

- **Provably Efficient and Scalable Shared-Memory Graph Processing**
  Laxman Dhulipala
  PhD Thesis, Carnegie Mellon University, 2020

[1]Our code is publicly available at `https://github.com/cmuparlay/psac`

## 1.4 Broader Outlook and Thesis Statement

Today, you would have a hard time finding a computer that is *not* parallel. Consumer-grade phones, desktops, laptops, and even your new smart fridge come equipped with multi-core CPUs. Even more importantly, large-scale computer systems, such as those that run your favorite social network or search engine could not exist without parallel computing due to the unfathomable amount of data that they process each day.

Parallel algorithms and dynamic algorithms have been studied for decades, but classic algorithms are ill-equipped to the kind of modern data sets that we see today which are both massive in scale *and* frequently changing. The study of parallel batch-dynamic algorithms is a relatively under-explored area, and the few results that existed prior to the last few years were highly work inefficient, making them completely impractical for real-world implementation. By focusing on the study of *work-efficient* parallel batch-dynamic algorithms, we aim to equip researchers with the theoretical tools that they need to design and analyse parallel systems suitable for processing the datasets of today and beyond.

Many of today's large datasets are graphs, collections of vertices connected by edges. The Microsoft Academic Graph, for example, consists of 179 million vertices and 2 billion edges. The hyperlink web graph, the largest known publicly available graph at the time of writing has over 3.5 billion vertices and 128 billion edges. Proprietary graph datasets, such as the Facebook graph, possibly the largest known graph dataset, have over one billion vertices and one trillion edges. Our goal was therefore to contribute tools in the space of work-efficient parallel batch-dynamic graph algorithms that would be broadly applicable to future researchers and practitioners, and to inspire additional work in this emerging field.

Since the appearance of the results of Chapters 3, 6, and 7, researchers have contributed additional independent results on parallel batch-dynamic graphs including $k$-Clique counting [52], and $k$-core decomposition [126]. Results on parallel batch-dynamic algorithms have also begun to appear in other models of computation such as the MPC model [51]. Furthermore, our results have been utilized by researchers as ingredients in new algorithms. Tseng et al. [175] extend our parallel batch-dynamic connectivity algorithm to obtain a parallel batch-dynamic algorithm for MSF, Ghaffari et al. [74] use Parallel RC-Trees as an ingredient to implement a nearly work-efficient highly parallel DFS, and of course we use our own results on Parallel RC-Trees to implement the first work-efficient parallel algorithm for minimum cuts (Chapters 8–9).

Based on these results, we conclude with the following as our thesis statement:

### Thesis Statement:

*Work-efficient parallel batch-dynamic algorithms can serve as a foundation for designing algorithms and systems that are well suited for processing today's large scale and rapidly evolving datasets, particularly for fundamental areas such as graph processing.*

# Chapter 2
# Preliminaries

## 2.1 Models of Computation

### 2.1.1 Models of Parallelism

#### Nested parallel programs

Our model for designing and analyzing parallel algorithms will be the shared-memory *nested-parallelism* model. We endow the model with the features of the standard word random-access machine (word RAM) model, i.e., it executes a program that has random-access to a set of registers, each of which stores a $\Theta(\log n)$-bit sized word, where $n$ is the problem size. The model assumes that bitwise and arithmetic operations on $\Theta(\log n)$-bit words takes $O(1)$ time. To express parallelism, the model provides a *fork* instruction. A *fork* spawns a set of child procedures that are eligible to run in parallel while the forking procedure suspends and waits until all of the child procedures are complete to resume. There is no restriction on the number of children that can be forked simultaneously by a fork instruction. The model supports *nested parallelism* in which a child procedure is allowed to fork its own additional child procedures. This model is an abstract model with no concept of "processors" or "threads". This means that algorithms are designed agnostic to the amount of available parallelism that may be used to execute them, and indicate *opportunities* for parallelism by using forks to spawn procedures that are eligible to be run in parallel. A scheduling algorithm is used to map a nested parallel program onto a concrete parallel machine model.

**Work and span** The efficiency of a nested parallel algorithm is described by a pair of measurements: its *work* and its *span* (also commonly called *depth*, *critical path length*, or *parallel time*). The work of the algorithm is the total number of instructions performed, and the span is the length of the longest chain of sequentially dependent instructions. The work can be viewed as the traditional running time of the algorithm in the sequential setting where no parallelism is available and all forked procedures are executed synchronously. The span can be viewed as the running time of the algorithm on a hypothetical machine with infinite available parallelism that is capable of executing all of the child procedures concurrently.

**The parallel for loop** When specifying nested-parallel algorithms, we will often use the higher-level *parallel for loop* abstraction. A parallel for loop specifies a set of elements for the loop variable and a loop body which is a function of the loop variable such that each iteration of the loop is eligible to be run in parallel.

### Related models

**The fork-join model** Nested parallel programs can also be endowed with a *join* operation. When a program spawns a set of child procedures, instead of immediately suspending and awaiting the completion of its children, the program continues executing and explicitly specifies when it wants to suspend and wait on the completion of a spawned procedure with the join instruction. In a *fully-strict* computation, child procedures can only be joined by the parent that spawned it. Some models allow joins of arbitrary procedures but this complicates scheduling so many scheduling algorithms do not permit it [30]. Our algorithms will not make use of the join operation.

**The binary-forking model** A variant of the nested parallel model is the *binary-forking* model [29], where a procedure may only fork two procedures per fork operation. The binary-forking model can simulate the general arbitrary-way model by having the fork instruction create a divide-and-conquer tree of binary forks, which incurs only a constant-factor work overhead but costs $\Theta(\log p)$ span to fork $p$ child procedures. Our algorithms are specified in the arbitrary-way model.

**The PRAM model** An classic model which we will reference frequently is the Parallel Random-Access Machine (PRAM) model, which consists of $p$ word RAM processors that execute in lock step with access to a shared memory. A specification of an algorithm in the PRAM model usually specifies $p$, the number of processors, as a function of the problem size $n$. We will describe all of our new algorithms as nested parallel programs, but we will use many classic PRAM algorithms as subroutines. To translate runtime bounds for classic PRAM algorithms into the nested parallel model, we use the fact that a PRAM algorithm that runs in $t$ time on $p$ processors performs at most $pt$ work with $t$ span.

## 2.1.2 Concurrency

A parallel computation may be endowed with the ability to read/write the shared memory concurrently. The classic PRAM model defines three kinds of concurrent machines, the EREW, CREW, and CRCW PRAMs.

**EREW (Exclusive-Read Exclusive-Write)** In the *EREW* model, no concurrency is allowed for either reads or writes. This is the most restrictive model.

**CREW (Concurrent-Read Exclusive-Write)** In the *CREW* model, concurrent reads are allowed but concurrent writes are prohibited. Writes may not be concurrent with reads or other writes, but reads may freely execute concurrently with each other.

**CRCW (Concurrent-Read Concurrent-Write)** The *CRCW* model permits concurrent reads and concurrent writes. The CRCW model may be further subdivided by the behaviour of concurrent writes.

- In the *Tolerant* CRCW PRAM, if multiple processors attempt to write to the same memory location concurrently, the memory location keeps its original value (the concurrent writes “fail”, but do not invalidate the computation).
- 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.
- The *Priority* CRCW PRAM assigns fixed unique priorities to each processor, and specifies that the highest priority processor’s write succeeds.
- The *Maximum* CRCW PRAM permits concurrent writes and assumes that processor attempting to write the largest value always succeeds.

### Atomic operations

Computations that rely on concurrent writes may require the model to be further endowed with additional more powerful primitives. Atomic *read-modify-write* primitives are a family of operations that allow a computation to simultaneously read a memory location and write a new value into it, which may even depend on the value that was read. A (non-exhaustive) list of common read-modify-write operations include the following.

- **TestAndSet**(): An atomic *test-and-set* writes the value 1 (or true) to the boolean variable and returns the original value.
- **Exchange**(desired): An atomic *exchange* simultaneously writes the value *desired* to the variable and returns the original value.
- **FetchAndAdd**($x$): An atomic *fetch-and-add* increments the value of the variable by the given amount $x$ and returns the original value.
- **CompareAndSwap**(expected, desired): An atomic *compare-and-swap* operation compares the value of a variable to the value of *expected*, and if they are equal, writes the new value *desired* to the variable, returning true. Otherwise it returns false.

## 2.2 Parallel Primitives

We make use of the following operations which are fundamental for many parallel algorithms.

**Reduce** A reduce operation over a range of elements takes an associative operator ($+$, $\times$, or any custom associative function $\oplus$) and returns the sum of the elements with respect to the operator. Assuming that the operator takes $O(1)$ time to evaluate, a reduce takes $O(n)$ work and $O(\log n)$ span.

**Scan** The scan operation generalizes the reduce operation. It takes a range of elements and an associative operator over those elements and returns a range consisting of the partial sums with respect to the operator and the total sum. Scan can be implemented in $O(n)$ work and $O(\log n)$ span assuming that the operator takes $O(1)$ time to evaluate.

**Filtering** A filter over a range takes a predicate $f$ and returns a new range consisting of the elements of the input for which $f$ returns true, in the same relative order. Filtering can be performed in $O(n)$ work and $O(\log n)$ span, provided that $f$ can be evaluated in $O(1)$ time.

**Approximate compaction** Approximate compaction takes the same input as a filter, a range of elements and a predicate $f$. The output is a range of elements from the input such that if there are $m$ elements that satisfy the predicate, the output is an array of size $O(m)$ containing the elements that satisfy the predicate (not necessarily in their original relative order), and some blank elements. Approximate compaction takes $O(n)$ work and $O(\log^* n)$ span w.h.p. in the Arbitrary CRCW model. Alternatively, it can be solved deterministically in $O(n)$ work and $O(\log\log n)$ span in the Common CRCW model [81].

**Pack** The *pack* operation takes a range of elements $A$ and a corresponding boolean range $B$ of the same length. The output is a sequence of all the elements $a \in A$ such that the corresponding entry in $B$ is true. The elements appear in the same relative order that they appeared in the input. Packing can be implemented in $O(n)$ work and $O(\log n)$ span.

**Comparison-based sorting** A comparison-based sorting algorithm takes as input a range of elements with associated keys from some totally ordered set and outputs a copy of that range such that the keys appear in sorted order. A range of $n$ elements can be sorted in $O(n \log n)$ work and $O(\log n)$ span [38].

**Integer sorting** An integer sorting algorithm takes as input a range of $n$ elements with associated integers keys in the range $[1, n]$ and and outputs a copy of that range such that the keys appear in sorted order. Integer sorting can be performed in $O(n)$ work and $O(\log n)$ span w.h.p. [153].

**Semisorting** The *semisort* operation takes as input a range of elements with associated keys, and outputs a copy of that range such that elements with equal keys are contiguous, but the keys do not necessarily appear in sorted order. A range of $n$ elements can be semisorted in $O(n)$ expected work and $O(\log n)$ span w.h.p. provided that the keys can be hashed uniformly into the range $\left[1, n^{O(1)}\right]$ [84].

**Parallel dictionaries** A parallel dictionary data structure supports batch insertion, batch deletion, and batch lookups of elements from some universe with hashing. A batch of $k$ operations can be performed in $O(k)$ work and $O(\log^* k)$ span w.h.p. in the Arbitrary CRCW model [76].

## 2.3 Randomness

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 $O(n^{-c})$. In line with existing algorithms research that uses random sampling [112], we assume that we can generate $O(1)$ random bits in $O(1)$ time. Sometimes we may invoke subroutines from other work that requires random $\Theta(\log n)$-bit words (such as [57]) and assumes that these can be generated in $O(1)$ time, so we will have to adjust bounds accordingly by adding an extra $O(\log n)$ factor to the work of generating the random bits.

An algorithm is called *Monte Carlo* if it is correct w.h.p. but runs in a deterministic amount of time. Similarly, an algorithm is called *Las Vegas* if it is fast w.h.p. but always correct. When designing Monte Carlo algorithms, we can use Las Vegas algorithms as subroutines because any Las Vegas algorithm can be converted into a Monte Carlo algorithm by halting and returning an arbitrary answer after the desired time bound. It is however not always possible to convert a Monte Carlo algorithm into a Las Vegas one unless a fast algorithm for verifying a solution is available.

# Part I

# Parallel Rake-Compress Trees

# Chapter 3
# Parallel Rake-Compress Trees

## 3.1 Introduction

Dynamic trees are a fundamental building block of a number of graph algorithms. The end goal of a dynamic tree data structure is to map a possibly imbalanced tree onto an efficient data structure that allows for altering the structure of the tree (such as inserting or removing edges, or changing the weights of edges) and querying for its properties (such as the total weight in a subtree, the heaviest edge on a path, or the lowest common ancestor of a pair of vertices) efficiently. Formally, the goal is support at least the following interface:

- **LINK**$(u, v, w)$: Add an edge $(u, v)$ to the forest, with (optional) weight $w$
- **CUT**$(e)$: Remove the edge $e$ from the forest
- **CONNECTED**$(u, v)$: Returns true if $u$ and $v$ are connected, i.e., in the same tree

Many additional operations may be supported, depending on the data structure. An illustrative but non-exhaustive list of possible operations include:

- **SUBTREEWEIGHT**$(u, p)$: Return the minimum/maximum/total weight of the vertices in the subtree rooted at $u$ relative to its parent $p$,
- **PATHWEIGHT**$(u, v)$: Return the minimum/maximum/total weight of all edges on the path between $u$ and $v$,
- **LCA**$(u, v, r)$: Returns the lowest common ancestor of $u$ and $v$, assuming that the tree containing $u$ and $v$ is rooted at $r$,
- **DIAMETER**$(u)$: Returns the diameter of the tree containing the vertex $u$. The diameter is the length of the longest simple path between any pair of vertices in the tree,
- **CENTER**$(u)$: Returns the center vertex of the tree containing $u$. The center is the vertex that minimizes the maximum distance to any other vertex in the same tree,
- **NEARESTMARKED**$(u)$: Return the nearest marked vertex to the vertex $u$.

Variations of these operations also exist. For example, **SUBTREEWEIGHT** could operate on edge weights and **PATHWEIGHT** could operate on vertex weights, or both at the same time. The Dynamic Median problem is similar to the Center problem, except that the median vertex $m$ is defined as the vertex that minimizes the weighted distance $\sum \text{weight}(v)\text{dist}(m, v)$ to all other vertices in the tree.

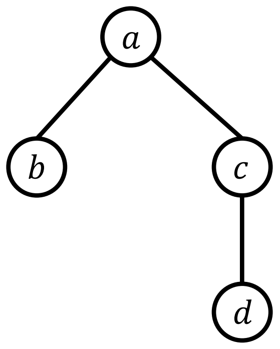
### 3.1.1 Related Work

Many dynamic tree data structures have been designed, with various trade-offs and abilities to support different subsets of the above operations. The most prominent are Euler-tour trees [94], link/cut trees [163, 164], topology trees [63, 64, 65], top trees [16, 98, 167], and Rake-Compress Trees (RC-Trees) [4, 5]. In this thesis, we extend RC-Trees, which were previously designed for sequential operations, to handle parallel batch-dynamic operations.

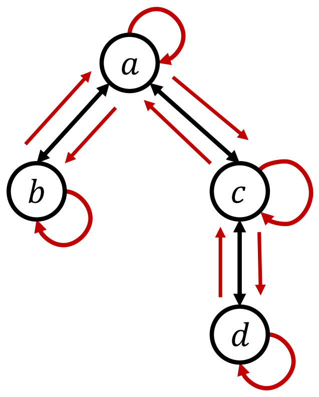
#### Euler-tour trees

Arguably the simplest dynamic tree data structure are the Euler-tour trees of Henzinger and King [94], which encode an arbitrary tree by an Eulerian traversal of its edges, which is then stored inside an efficient ordered data structure, usually a balanced binary search tree, but alternatives such as a skip list can be used too. The insight that makes this powerful is that every subtree of the input tree corresponds to some contiguous interval of the Eulerian traversal. This means that (1) cutting a subtree from its parent corresponds to simply splicing out a contiguous subsequence of the Eulerian traversal, (2) to link a tree into another tree by making it the child of a vertex corresponds to splicing the Eulerian traversal of the first tree into the middle of the Eulerian traversal of the second, and (3) to query for a property of a subtree, such as its maximum or total edge weight, we just need to compute the maximum or total edge weight in the corresponding contiguous subsequence of the tour, which is a standard operation on binary search trees and skip lists. The downside of Euler-tour trees is that they are only capable of supporting subtree-based queries, and not capable of supporting path queries, such as determining the heaviest or lightest edge on a path.

Earlier work has shown how to implement parallel batch-dynamic Euler-tour trees by using parallel batch-dynamic skip lists to store the traversals [174]. This batch-dynamic data structure can handle batches of links and batches of cut operations, as well as batch queries for the total weight in a subtree.



**(a) A tree**



**(b) An Eulerian traversal of the tree.**

**Figure 3.1: An Euler-tour tree computes an Eulerian traversal of the tree and stores the resulting edge sequence in an efficient data structure such as a binary search tree.**

## Link/cut trees

Link/cut trees are the earliest efficient solution to the dynamic trees problem, proposed by Sleator and Tarjan [163]. Unlike Euler-tour trees which are based on subtrees, they represent the underlying tree by breaking it up into vertex-disjoint paths, which are then stored inside a dynamic search tree. The key idea is to designate each edge of the forest as either solid or broken, such that each vertex has a solid edge to at most one of its children. The vertex-disjoint paths are then induced by the solid edges (some paths may consist of an isolated vertex). These paths are kept structured in such a way that $m$ dynamic tree operations translates into $O(m \log n)$ path operations. Using standard balanced binary search trees, this would lead to a runtime of $O(\log^2 n)$ per operation, so Sleator and Tarjan show that one can instead used *biased* search trees [163] or Splay Trees [164] to obtain just $O(\log n)$ time (amortized if using Splay Trees).

Being based on path decomposition, link/cut trees are, unsurprisingly, ideal for handling path-based operations. They can implement the **PathWeight** query, in addition to supporting more powerful *path update* operations. For example, a link/cut tree can support adding a given amount $x$ to the weight of every edge on the path between two given vertices $u$ and $v$. Goldberg, Grigoriadis, and Tarjan [80] show how to extend link/cut trees to also support subtree queries, though it complicates the internal implementation of the data structure, and only supports bounded-degree input trees.

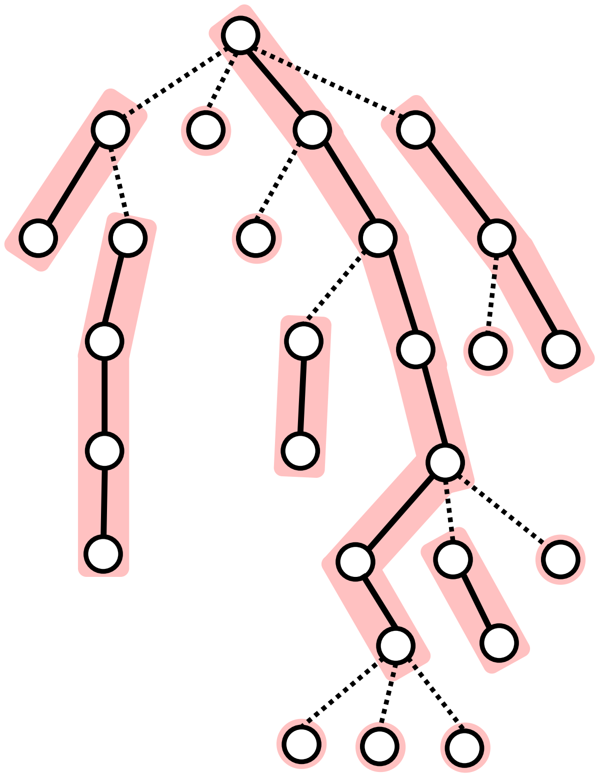


**Figure 3.2: A link/cut tree [163]. Dashed edges are broken and solid edges are solid. The resulting path decomposition is depicted by the colored background.**

## Topology trees

Topology trees [63, 64, 65] introduce a different kind of decomposition strategy compared to Euler-tour trees and link/cut trees. Instead of representing the underlying tree by decomposing it into subtrees or paths, they represent the tree using a *hierarchical clustering*.

Topology trees use a clustering on the vertices of the tree, which they call a *restricted multi-level partition*. At the bottom level, each vertex is a singleton cluster, then at each subsequent level, a maximal set of adjacent clusters are paired together subject to some constraints on the structure and maximum degree. The maximum degree requirement limits the data structure to bounded-degree trees only. The topology tree represents the clustering such that each topology tree node represents one of the clusters in the multi-level partition, where an internal node represents the cluster formed by the union of the clusters represented by its children. By maintaining aggregate information on the clusters, such as weight totals or maximums, various kinds of queries can be answered while the tree is updated dynamically.

Frederickson [65] shows how to implement all of the operations of a link/cut tree, such as querying for the maximum weight edge on a path, using topology trees, and also uses them to implement dynamic expression trees. Topology trees were originally invented to help solve the dynamic MST problem by finding replacement edges for the MST if a tree edge had its cost increased.

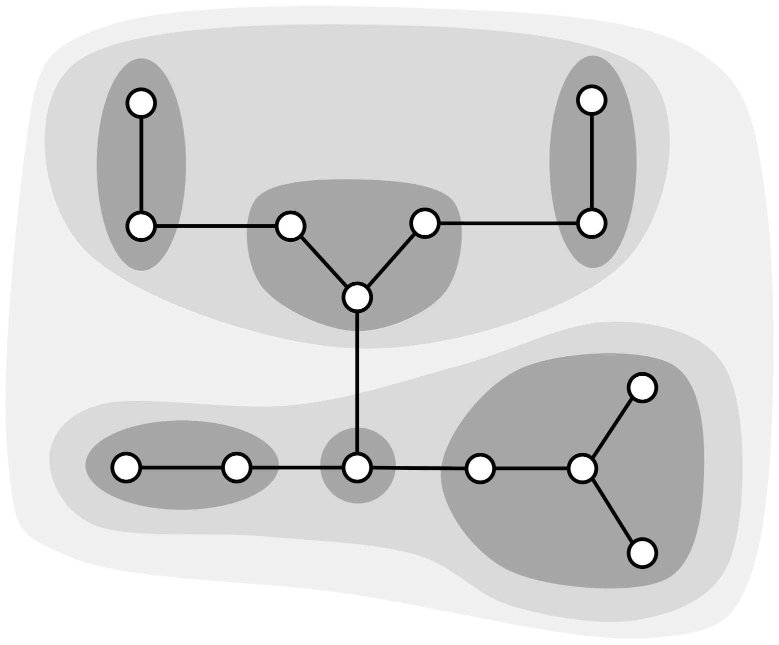


**Figure 3.3: A restricted multi-level partition that is represented by a topology tree [63]. Darker backgrounds represents clusters formed at lower levels.**

## Top trees

Top trees [16, 98, 167] are similar to topology trees in that they represent a hierarchical clustering of the underlying tree. The key difference is that instead of using a clustering of the vertices, they represent a *clustering of the edges* of the forest. This allows them to elegantly overcome the bounded-degree restriction of topology trees. Top trees are the most general of the sequential dynamic tree data structures. They are able to implement all of the styles of queries, from both path and subtree queries, to complex queries such as lowest common ancestors, nearest marked nodes, centers, medians, and diameters, all in $O(\log n)$ work.

There are several different implementations of top trees. They were originally designed as an interface on top of top trees or link/cut trees [16, 98], which support either a worst-case or amortized runtime. They have also been implemented directly using techniques similar to Splay Trees to yield *self-adjusting top trees* [167] and *splay top trees* [102], which both exhibit amortized running times.

## 3.2 Rake-Compress Trees

In this section we will introduce and define Rake-Compress Trees (RC-Trees). RC-Trees were initially proposed by Acar et al. [5], but our definitions will differ slightly from theirs to clarify some corner cases that we believe theirs do not cover. We will begin by describing parallel tree contraction, the underlying process that RC-Trees are based on. We will describe how to build an RC-Tree, how to maintain one subject to dynamic updates, and how to solve various common queries using the data structure. Our main contribution is then to describe how each of these processes can be generalized to handle parallel batch-dynamic operations.

### 3.2.1 Parallel Tree Contraction

Tree contraction is a procedure for computing functions over trees in parallel in low span [132]. It involves repeatedly applying *rake* and *compress* operations to the tree while aggregating data specific to the problem. The rake operation removes a leaf (a vertex of degree one) from the tree and aggregates its data with its neighbor. If the leaf is vertex $u$ and its neighbor is vertex $v$, we say that $u$ *rakes onto* $v$. 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 [132] describe a linear work and $O(\log n)$ span randomized algorithm that performs a set of rounds, each round raking every leaf (unless there are two adjacent leaves, then one of them rakes onto the other) 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 [70] improve it to obtain a work-efficient deterministic algorithm with $O(\log n)$ span.

Parallel tree contraction is defined for constant-degree trees, so non-constant-degree trees are handled by converting them into bounded-degree equivalents, e.g., by ternerization [63]. An example of parallel tree contraction applied to a tree is depicted in Figure 3.4.

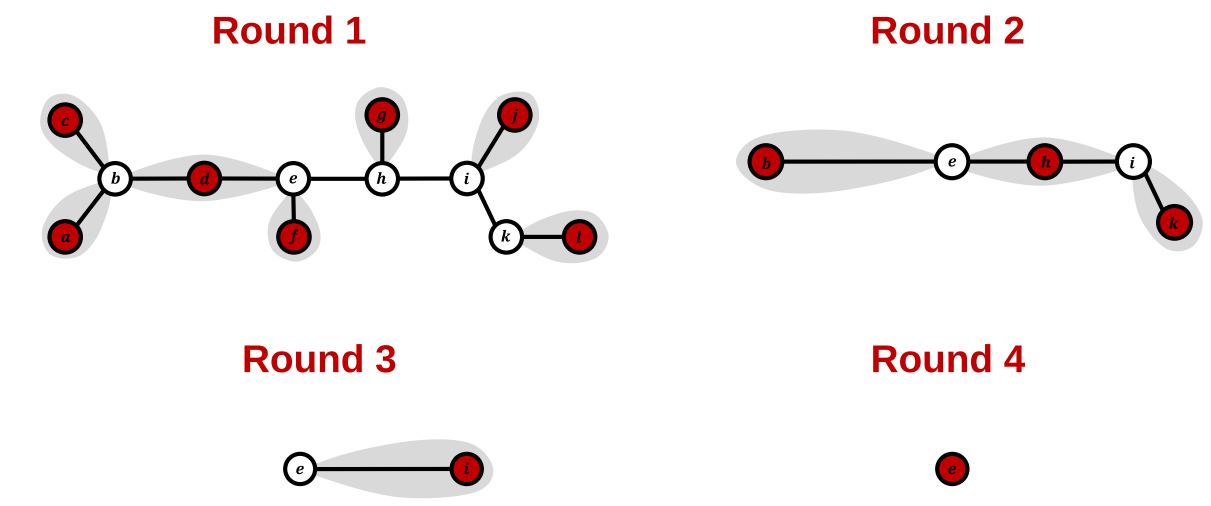


**Figure 3.4: Parallel tree contraction. Each round, an independent set of vertices is selected to contract, until the final round in which there is only a single remaining vertex.**

## 3.2.2 From Tree Contraction to RC-Trees

Like topology trees and top trees, RC-Trees are based on a hierarchical clustering of the underlying tree. The key idea is to view the process of parallel tree contraction as inducing a clustering. Each rake and each compress operation induces a cluster, hence the name Rake-Compress Tree. For RC-Trees, a cluster is a connected subset of edges and vertices of the tree. The 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 and top trees, which cluster just the vertices or just the edges respectively. Importantly but somewhat unintuitively, a RC cluster may contain an edge without containing the 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. Every cluster has at most two boundary vertices. Figure 3.5 shows an example of a cluster and its boundary vertices

**Definition 1** (Boundary Vertex)**.** A boundary vertex of an RC Cluster is a vertex that is **not** in the cluster, but is an endpoint of an edge that is in the cluster.

The base clusters are singletons containing the individual edges and vertices of the tree, so there are $n + m$ base clusters. An internal cluster contains the union of the contents of its children. 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).

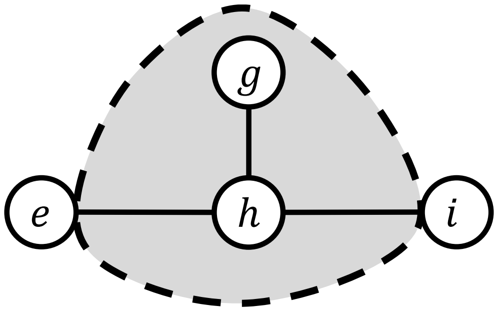


**Figure 3.5: An example of a hypothetical cluster containing the vertices $g$ and $h$, and the edges $(g,h),(e,h),(h,i)$. Since vertices $e$ and $i$ are *not* contained in the cluster but are endpoints of edges inside the cluster, they are the cluster's *boundary vertices.***

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. Since each vertex contracts exactly once, there is a one-to-one mapping between representative vertices of the original tree and internal clusters. The boundary vertices of the resulting cluster will always be the union of the boundary vertices of the constituents, minus $v$.

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 [16]).

## Unary clusters

Clusters that arise from rake operations always have exactly one boundary vertex, and are hence called *unary clusters*. A rake operation that deletes a leaf $v$ and its adjacent edge $(u,v)$ will create a unary cluster containing:

1. A single binary cluster with boundaries $u$ and $v$ corresponding to the edge that was deleted
2. A base cluster corresponding to the representative vertex $v$ that was raked
3. Zero or more unary clusters corresponding to vertices that raked onto $v$ in earlier rounds

## Binary clusters

Clusters that arise from compress operations always have exactly two boundary vertices, so we call them *binary clusters*. A compress operation deletes some vertex $v$ with degree two, whose neighbor's we will call $u$ and $w$, and replaces the edges $(u,v)$ and $(v,w)$ with a single edge $(u,w)$. This creates an internal cluster that contains:

1. A pair of binary clusters corresponding to the edges $(u,v)$ and $(v,w)$ that were deleted
2. A base cluster corresponding to the representative vertex $v$ that was compressed
3. Zero or more unary clusters corresponding to vertices that raked onto $v$ in earlier rounds

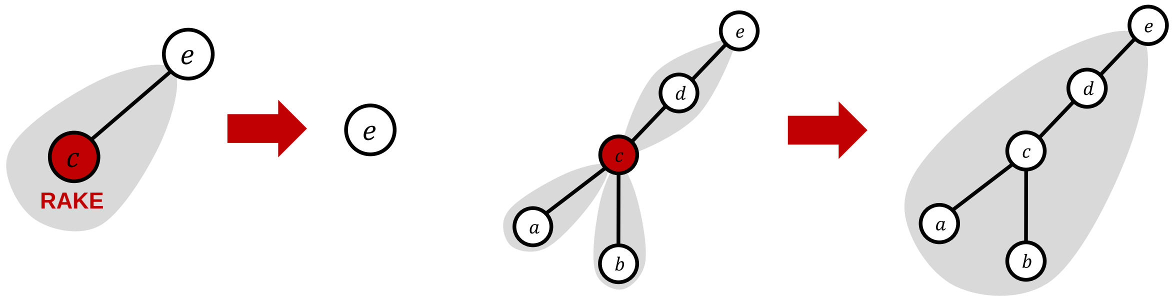


**Figure 3.6: A unary cluster arising from a rake operation. The left pair of images is the contracted tree, where the vertex $c$ rakes onto the vertex $e$. The right pair of images is the corresponding induced clustering. The rake causes all of the RC clusters that share $c$ as a boundary vertex to merge into a single cluster represented by $c$. This consists of the unary clusters represented by $a$ and $b$, which must have raked onto $c$ in an earlier round, and the binary cluster represented by $d$, which must have compressed in an earlier round. Since $e$ is the only boundary vertex not shared by all of the clusters, it remains as the resulting cluster's single boundary vertex.**

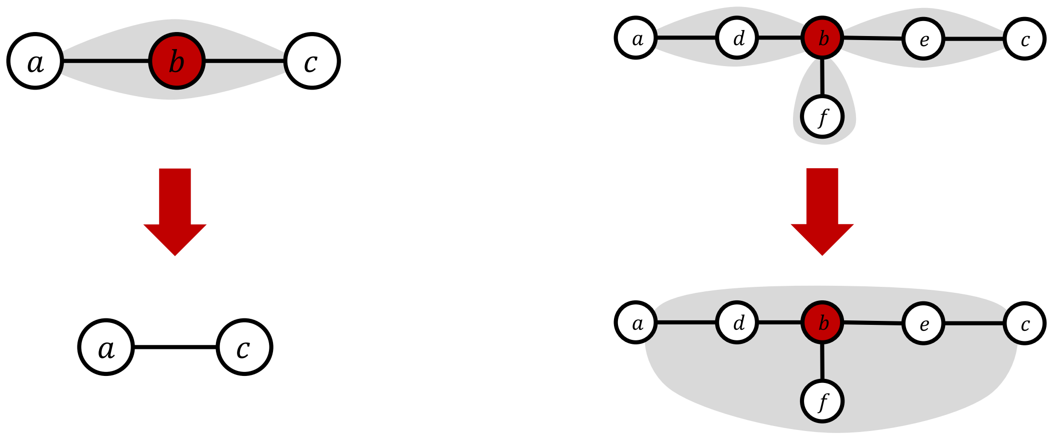


**Figure 3.7: A binary cluster arising from a compress operation. On the left, the vertex $b$ compresses in between $a$ and $c$. This causes all of the RC clusters that share $c$ as a boundary vertex to merge into a single cluster represented by $c$. This consists of the unary cluster represented by $f$, which must have raked onto $b$ in an earlier round, and the two binary clusters represented by $d$ and $e$, which must have compressed in an earlier round. $a$ and $c$ are the surviving vertices and end up as the boundary vertices of the resulting cluster.**

### Nullary clusters and root clusters

A cluster with no boundary vertices is called a *nullary cluster*. There are two ways in which nullary clusters arise. First, the base vertex clusters do not contain any boundary vertices since they contain no edges, so they are naturally nullary clusters. The second kind of nullary cluster is a root cluster, i.e., a cluster containing an entire tree. This arises from contracting the singleton vertex that remains once tree contraction has reduced a tree down to a single vertex. Contracting a singleton vertex is often referred to as a *finalize* operation. Note that a finalize cluster only has unary children and can not have binary children.

## 3.2.3 Representing RC-Trees

An RC-Tree is a tree of nodes[1] each representing an RC Cluster. The leaves of the tree are the base clusters representing singleton vertices and edges, and each internal node is a cluster arising from a rake, compress, or a root cluster. If the input forest is not connected, the RC-Tree will actually be a forest of RC-Trees, one for each connected component in the input.

Since there is a one-to-one correspondence between vertices of the input tree and internal nodes of the RC-Tree (their representative vertex), and every cluster corresponding to an internal node necessarily has the base cluster corresponding to its representative vertex as a child, we can optionally omit the base vertex clusters from the representation for simplicity, since we can always identify where they would be. Therefore, an internal binary (compress) cluster always has two binary children and zero or more unary children, an internal unary (rake) cluster always has one binary child and zero or more unary children, and the root (finalize) cluster has no binary children and zero or more unary children. A complete example of a clustering into RC Clusters and the corresponding RC-Tree is depicted in Figure 3.8.

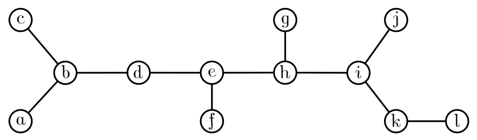
## 3.2.4 Balanced RC-Trees

Given an input tree, there is no guarantee that there is a unique RC-Tree that encodes it. Indeed there are likely to be many. For example, one can always construct the "rake tree" of an input tree by performing only rake operations. The resulting RC-Tree unfortunately will be just as imbalanced as the input tree, so this is not very useful. To make sure that our RC-Trees are useful, we define the notion of a "balanced RC-Tree".

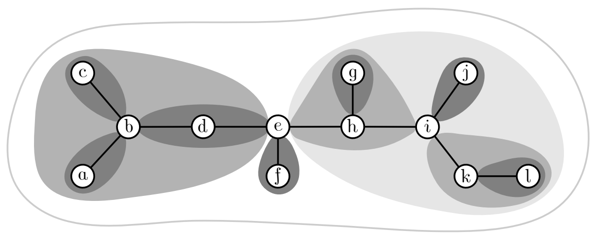
**Definition 2** (Balanced RC-Tree)**.** An RC-Tree is $\beta$-balanced if it corresponds to a tree contraction such that in each round, at least a fraction of $1-\beta$ of the vertices contract. If the contraction processed is randomized, it is $\beta$-balanced in expectation if at least an expected fraction of $1-\beta$ of the vertices contract. We call an RC-Tree balanced (resp. in expectation) if it is $\beta$-balanced (resp. in expectation) for some constant $0 < \beta < 1$ that is independent of $n$.

The randomized tree contraction algorithm that we analyze in Chapter 4 has $\beta = 7/8$, and our deterministic algorithm in Chapter 5 has $\beta = 5/6$.

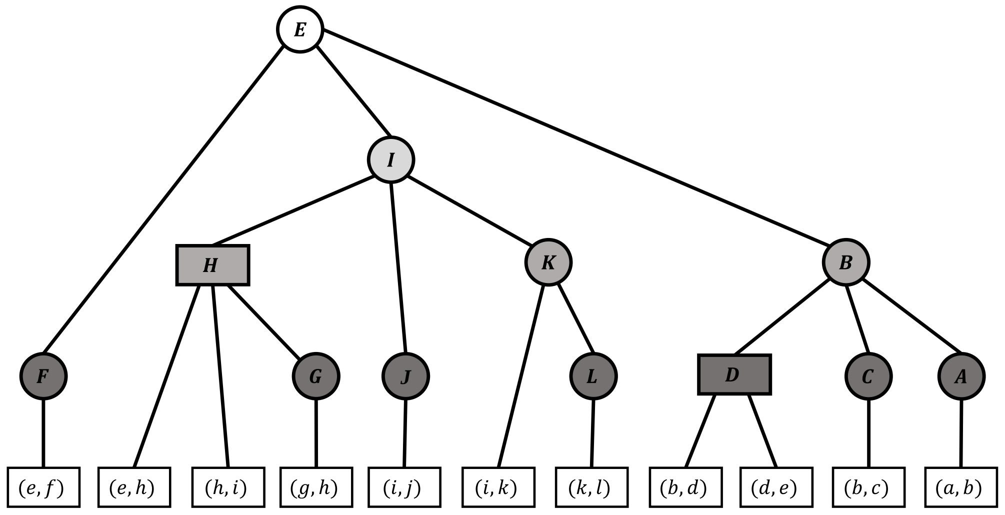
[1]For clarity, we will refer to the vertices of the RC-Tree as "nodes", and the vertices of the input as "vertices".



**(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 RC-Tree. Clusters produced from rakes (and the root cluster) are shown as circles, and clusters produced from compress as rectangles. The base clusters (edges) are labeled in lowercase, and the composite clusters are labeled with the uppercase of their representative vertex. The shade of a composite cluster corresponds to its height in the clustering. Lower heights (i.e., contracted earlier) are darker. The order of the children of a cluster is not important.**

**Figure 3.8: A tree, a clustering, and the corresponding RC-Tree.**

It is easy to show that a balanced RC-Tree will have height $O(\log n)$, but we note that simply having $O(\log n)$ height alone is not sufficient for us to consider the tree balanced. Although this would result in single queries being efficient, our goal is to ensure that *batch queries* executed on an RC-Tree will be efficient. It will be typical of batch RC-Tree queries to start at a set of nodes related to the queries, then aggregate some information along all of the paths to the root from those nodes (see Figure 3.9). The total work of such queries is typically proportional to the total number of unique nodes in this set of paths. On any balanced RC-Tree, the number of such nodes is $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ as shown by the following.

**Theorem 8** (Balanced RC-Trees)**.** Given any $k$ nodes in a balanced (resp. in expectation) RC-Tree, the total number of nodes in the union of the paths from those nodes to the root is at most $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ (resp. in expectation).

*Proof.* We consider separately the nodes at height at least $h = \log_{1/\beta}\left(1+\frac{n}{k}\right)$ and those lower. First consider the nodes with height less than $h$. In the worst case, the root paths of the $k$ nodes have no intersection at these heights, and given any starting node, it can contribute at most $h$ nodes to the paths before those nodes have height at least $h$. Therefore, there are at most $kh$ total nodes at height less than $h$.

Now consider the nodes of the RC-Tree at height at least $h$. If a node is at height at least $h$, then it has at least $h$ descendants in the RC-Tree, which means that it must correspond to a contraction that occurs after at least $h$ rounds have already occurred. Since the RC-Tree is balanced, there exists a $\beta$ such that at least a $1-\beta$ fraction of the vertices contract each round (in expectation if randomized). So after $\log_{1/\beta}\left(1+\frac{n}{k}\right)$ rounds, there are at most

$$n\beta^{\log_{1/\beta}\left(1+\frac{n}{k}\right)} = \frac{n}{1+\frac{n}{k}} = \frac{nk}{n+k} = k\left(\frac{n}{n+k}\right) \le k$$

remaining vertices (in expectation if randomized). Therefore there are at most $k$ nodes in the RC-Tree that can correspond to contractions that occur after $h$ rounds. Therefore, the total number of nodes in the union of the $k$ paths is at most

$$kh + k = k\log_{1/\beta}\left(1+\frac{n}{k}\right) + k = O\left(k\log\left(1+\frac{n}{k}\right)\right),$$

(in expectation if randomized). □

To see why the RC-Tree being height $O(\log n)$ is insufficient, consider a hypothetical RC-Tree that consists of a perfect binary tree with $n/\log n$ leaves, with chains of length $\log n$ attached to each of those leaves, as per Figure 3.10. From here onward, even when not explicitly specified, we will always assume that we are working with balanced RC-Trees.

## 3.2.5 Rooted RC-Trees

The RC-Tree framework is defined for undirected/unrooted trees. Typically, queries that involve rooted trees are supported by passing the would-be root of the tree, or the parent vertex of a query parameter as an additional argument to clarify how the tree is oriented. For example, the **SubtreeWeight**$(v, p)$ operation typically takes two arguments: the root of the subtree $v$, and the parent vertex $p$. In some applications however, it may be desirable or convenient for the RC-Tree to permanently consider the underlying tree as rooted with a fixed, unchanging root vertex. This can be achieved simply by requiring that the underlying tree-contraction algorithm always contracts the root vertex last (indeed, tree contraction was originally defined exactly this way for rooted trees). The tree contraction algorithms that we consider in Chapters 4 and 5 can easily handle this requirement. The result is an RC-Tree with some additional useful properties for processing rooted trees:

1. The representative of the root cluster will always be the root vertex of the underlying tree,
2. (Non-base) binary clusters can distinguish their two binary children as the "top" and "bottom" child, since one will always be the ancestor of the other in the underlying tree.

Unless otherwise specified, we will always assume by default that we are working with undirected/unrooted trees, and will specifically mention if we require rooted RC-Trees.

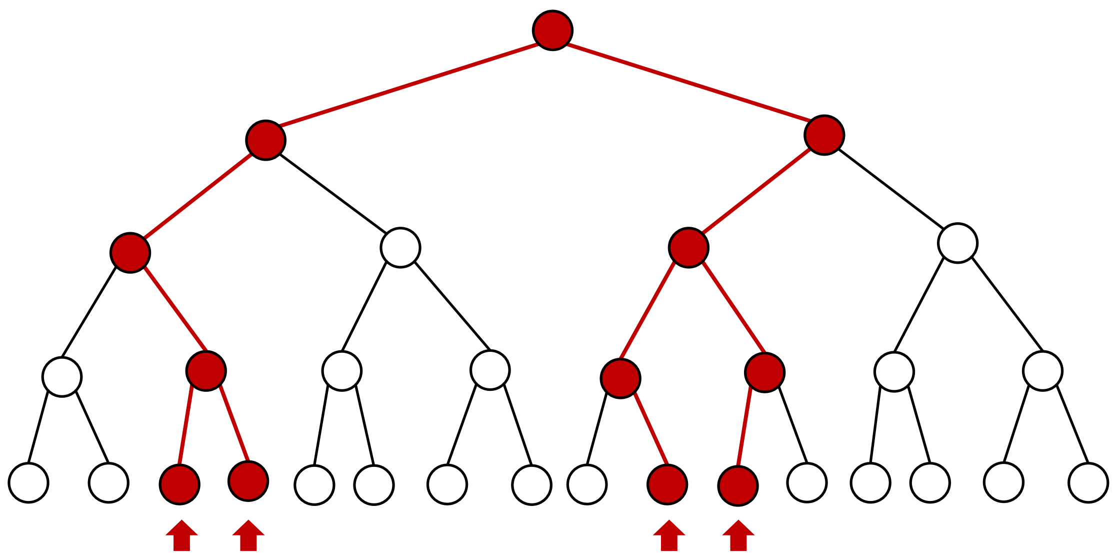


**Figure 3.9: A typical batch query on an RC-Tree will select a set of RC nodes then aggregate some information along the paths from those nodes to the root. The work performed by such a query is usually proportional to the number of unique nodes on these paths.**

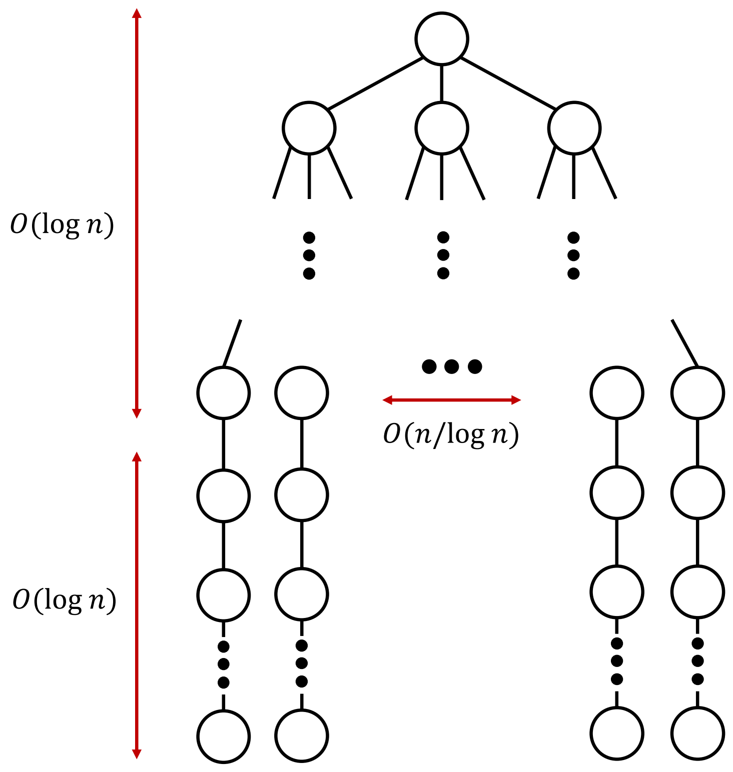


**Figure 3.10: A hypothetical "imbalanced" tree despite it having height $O(\log n)$. If we select $k$ nodes at the bottom of the chains, then the union of their root-to-leaf paths contains $\Theta(k \log n)$ nodes, but we want to guarantee that there are only $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$.**

## 3.3 Parallel Batch-Dynamic RC-Trees

Acar et al. [5] describe sequential RC-Trees based on applying self-adjusting computation to parallel tree contraction. The application of self-adjusting computation to parallel tree contraction yielded a *sequential* algorithm for efficiently updating a tree contraction. When the tree contraction is updated by change propagation, the corresponding parts of the RC-Tree are rebuilt. Our goal is to generalize this to the parallel batch-dynamic setting. Specifically, our goal is to support at least the following interface for a dynamic forest $F$.

- **BatchLink**($\{(u_1, v_1), \ldots, (u_k, v_k)\}$) takes a batch of edges and adds them to $F$. The edges must not create a cycle.
- **BatchCut**($\{(u_1, v_1), \ldots, (u_k, v_k)\}$) takes a batch of edges and removes them from $F$.
- **BatchConnected**($\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\}$) takes an array of tuples representing queries. The output is an array where the $i$-th entry returns whether vertices $u_i$ and $v_i$ are connected by a path in $F$.

Additional batch queries may be supported, such as

- **BatchSubtreeWeight**($\{(u_1, p_1), \ldots, (u_k, p_k)\}$) takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry contains the minimum/maximum/total weight of the contents of the subtree rooted at $u_i$ relative to the parent $p_i$.
- **BatchPathWeight**($\{(u_1, v_1), \ldots, (u_k, v_k)\}$) takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry contains the minimum/maximum/total of the weights on the path between $u_i$ and $v_i$.
- **BatchLCA**($\{(u_1, v_1, r_1), \ldots, (u_k, v_k, r_k)\}$) takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry is the LCA of $u_i$ and $v_i$ with respect to the root $r_i$.
- **BatchNearestMarked**($\{v_1, \ldots, v_k\}$) takes an array of vertices representing queries. The output is an array where the $i^{\text{th}}$ entry is the nearest marked vertex to $v_i$.

Note that there is no reason to consider batched versions of **Diameter**, **Center**, etc, since those queries return a single value for an entire tree. A batch of such queries would either consist of duplicates of the same query, or queries on disjoint trees which can not benefit from batching. Since these queries can already be solved sequentially in $O(\log n)$ time, by convexity, solving a set of $k$ such queries on disjoint trees would take just

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

work anyway, where the $n_i$ are the sizes of the trees, and $\sum n_i = n$, the total number of vertices in the forest. Therefore for such queries, efficient batch bounds are obtained trivially by simply running the existing sequential algorithm concurrently for each tree.

In Section 3.5, we will show how to solve all of the above batch queries efficiently in $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work and $O(\log n)$ span for batches of size $k$.

### 3.3.1 Performing Structural Updates to RC-Trees

Like sequential RC-Trees, we will implement Parallel RC-Trees by building them on top of dynamic parallel tree contraction. In Chapters 4 and 5, we will derive two algorithms for maintaining a parallel tree contraction under batch updates, one randomized and one deterministic. To maintain the Parallel RC-Tree, when the user executes an update operation (**BatchLink** or **BatchCut**), we will execute the corresponding operation on the tree contraction. Updates to the tree contraction performed by the dynamic tree contraction algorithm are then reflected in the RC-Tree by rebuilding any cluster whose representative vertex was affected during the update.

Since there is a one-to-one correspondence between clusters of the RC-Tree and representative vertices, a straightforward and convenient way to maintain the RC-Tree is to store an array of clusters of length $n+m$, where the first $n$ clusters correspond to the non-base clusters with the corresponding representative vertex, and the final $m$ clusters are the base edge clusters (including their weights if applicable). When the tree contraction algorithm performs a rake, compress, or finalize of a vertex $v$, it creates or updates the cluster stored at the position of $v$. Optionally, another $n$ slots could be used to store a weight or value for each vertex, depending the application.

Creating or updating a cluster for vertex $v$ consists in recording the children of the cluster, which are the unary clusters corresponding to vertices that raked onto $v$, and the one or two binary clusters that form the contraction; and computing any required *augmented values*, which are additional information required by the desired query algorithms.

### 3.3.2 Handling Trees of Arbitrary Degree

Since RC-Trees are based on parallel tree contraction, which works only on trees of bounded degree, we must take additional care to handle trees of arbitrary degree. The most common technique is called *ternerization* [109]: we can split a vertex $u$ of degree $d$ into a path consisting of $u$ and $d$ “fake vertices” $u_1, \ldots, u_d$ of degree at most three, connected by “fake edges”. Each of the fake vertices is adjacent to one of the original neighbors of $u$ via a “real edge”. See Figure 3.11 for an example. Given a static tree, performing this transformation to every high-degree vertex results in a tree with maximum degree three with at most $O(n+m)$ vertices and $O(m)$ edges for an initial tree of $n$ vertices and $m$ edges. Since any forest has $m < n$, the total size of the forest is increased by no more than a constant factor.

**Linking new edges** Handling dynamic trees is straightforward if we allow adding new vertices to the underlying forest. Although most dynamic tree data structures do not explicitly mention handling vertex additions in their API, supporting them is usually trivial. For instance, adding an isolated vertex to an RC-Tree consists in adding the vertex to the underlying forest and then creating a single RC-node with no children represented by that vertex. The addition of a new edge $(u, v)$ to the underlying forest then consists in adding two new fake vertices to the ternarized forest, one for $u$ and one for $v$, then adding the real edge between them. The modifications are depicted in Figure 3.12.

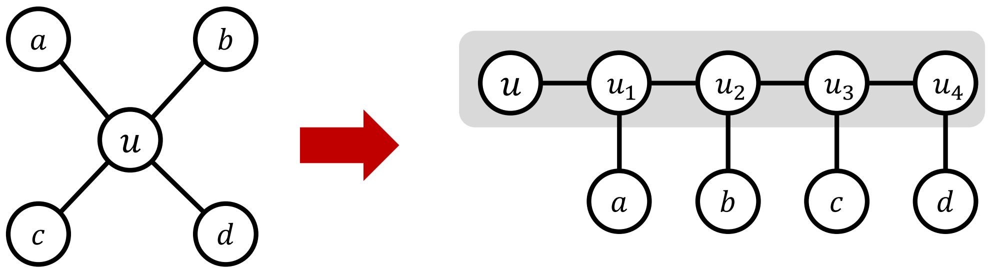


**Figure 3.11: Ternarization of a vertex in a tree. The original vertex $u$ of degree $d$ is connected to a path $u_1, ..., u_d$ of "fake" vertices. The resulting vertices have degree at most three.**

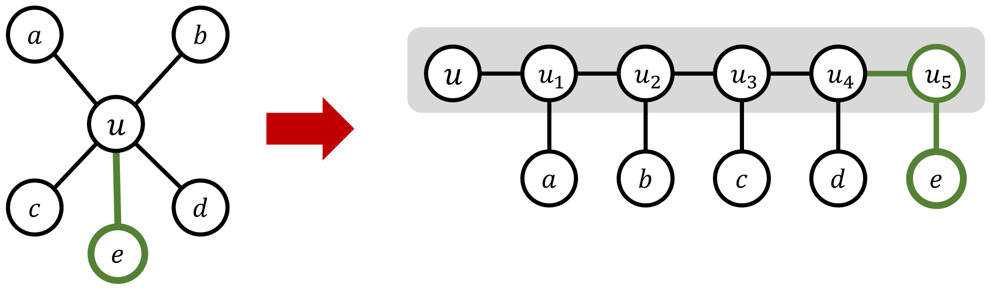


**Figure 3.12: Adding an edge to a ternarized forest. Only the modification to the path representing $u$ is depicted. The path representing $e$ would be modified symmetrically.**

**Cutting an edge** Cutting an edge from the underlying forest removes the corresponding edge and its neighboring fake vertices from the ternarized forest. If a fake vertex adjacent to the edge has degree three, then the two neighboring fake vertices must be linked to keep the path connected. Figure 3.13 depicts the process of cutting an edge and the resulting modifications to the path representing the vertex $u$.

**Batch updates** To support batch-dynamic algorithms over high-degree trees we may need to perform ternarization during batch links and batch cuts as well. To perform a batch link, the endpoints of the edges can be collected using a semisort [84] so that each real vertex gets a list of new edges to add. Each real vertex then adds the corresponding number of fake vertices to their path. The position of each endpoint in the result of the semisort can be

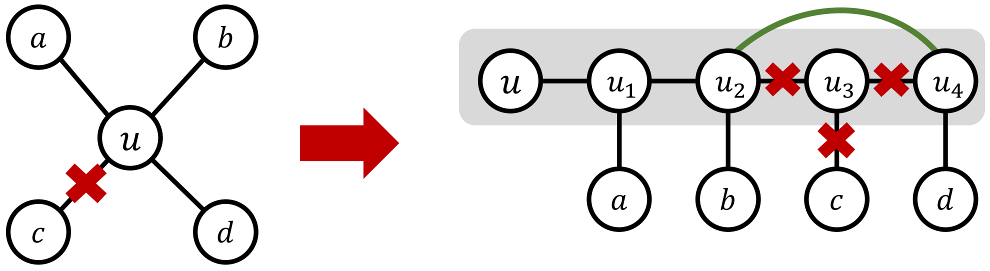


**Figure 3.13: Cutting an edge from a ternarized forest. Only the modification to the path representing $u$ is depicted. The path representing $c$ would be modified symmetrically.**

used to uniquely assign each real edge to a fake vertex. A batch link can then be performed on the ternarized forest to join all of the new fake vertices by fake edges, and to add the corresponding real edges for each edge in the input. This transforms a batch of $k$ links into $2k$ new fake vertices and $3k$ links in the ternarized forest.

Batch cuts are similar but slightly more involved. For each edge in the input batch, the corresponding real edge in the ternarized forest is removed in addition to their neighboring fake vertices. Note that multiple adjacent fake vertices on a path might be removed. This means that the algorithm needs to batch cut all of the fake edges along each contiguous run of removed fake vertices and then reconnect each such run with a single fake edge. To determine these replacement edges, the algorithm can take the batch of removed fake edges and run list contraction [39], keeping track of the endpoints of each component. Each component with two endpoints generates a replacement fake edge between those endpoints, which are inserted with a batch link into the ternarized forest. This transforms a batch of $k$ cuts into $2k$ removed vertices and $O(k)$ links and cuts in the ternarized forest.

**Handling queries on the ternarized forest** When ternarizing the input forest, additional care might be necessary depending on the problem being solved. In general, there is no single transformation that works for every problem so these transformations need to be designed specifically for the problem at hand. For simple problems such as connectivity, no additional modifications are necessary, as the result of a connectivity query in the underlying forest is the same as in the ternarized forest. For weighted problems such as computing the sum of the edge weights in a particular subtree or the maximum weight edge in a subtree, suitable identity weights must be used for the fake edges in the ternarized forest.

For example, if computing the sum of the weights, the fake edges would have a weight of zero to preserve the answers to the queries. If computing the maximum weight edge in a subtree, the fake edges should have a weight of $-\infty$. Lastly, if solving the minimum cut problem, giving the fake edges a weight of $\infty$ would preserve the value of all minimum cuts. If an algorithm uses vertex weights, similar strategies should be used to provide an identity weight that preserves the answers to queries.

### 3.3.3 Maintaining Augmented Values in RC-Trees

To support non-trivial queries, RC-Trees maintain *augmented values* on the clusters, which are functions of the child clusters and their augmented values. Each kind of query that we wish to support will therefore need to specify what augmented values it maintains, and how they are aggregated when clusters are created. The type of data stored may be different depending on the type of the cluster (unary, binary, finalize). The weights of the edges, if present, can act as "base cases" for the augmented values.

For example, to support queries for the maximum edge weight in a subtree, we may wish to maintain, for each cluster, the maximum edge weight inside that cluster as the augmented value. This would be computed by taking the maximum of the augmented values of the child clusters, including any base-edge clusters if present.

It is also straightforward to support weight updates. If one wishes to modify the weight of an edge, it suffices to recompute the augmented values for all of the ancestor clusters of that edge. No modification to the tree contraction needs to be made, and hence no structural updates to the RC-Tree are necessary when only updating a weight.

It is similarly possible to maintain weights or additional augmented data on vertices if desired. Augmented values on clusters could then depend on the weight of the representative vertex and take it into account when creating or updating a cluster. Updating the weight of a vertex would similarly just require updating the augmented values of the ancestors of the cluster that it represents.

## 3.4 Decomposition Properties and Queries

One factor that contrasts RC-Trees and top trees against the more classic dynamic tree data structure such as link/cut trees and Euler-tour trees is that they represent the underlying tree by hierarchically decomposing it into clusters. link/cut trees decompose the tree into paths and Euler-tour trees decompose it into rooted subtrees, which can be viewed as very restricted kinds of clusters that are suitable for specific purposes (i.e., path queries and subtree queries). The more general and hierarchical clustering used by RC-Trees and top trees is what makes them so powerful and able to solve a much wider range of queries.

One way to see this is as follows. We have two main kinds of clusters: unary clusters, which have one boundary vertex; and binary clusters, which have two boundary vertices. Unary clusters represent rooted subtrees that are rooted at the boundary vertex, while binary clusters represent the path between its two boundary vertices, as well as any branches that hang off that path. The unary/binary clustering therefore simultaneously represents information about both paths and subtrees.

### 3.4.1 The Cluster Path

Binary clusters are particularly useful because they can simultaneously be used to represent information about the entire subtree that they contain, or to represent information purely about the path between the two boundary vertices. The later is used quite extensively so it gets a name, the *cluster path.*

**Definition 3** (Cluster path)**.** Given an RC-Tree for an input forest $F$ and a binary cluster of the RC-Tree with boundary vertices $u$ and $v$, we define the *cluster path* of the binary cluster as the path connecting $u$ and $v$ in $F$.

Since the cluster path of a composite binary cluster is just the union of the cluster paths of its binary children, it is easy to aggregate and store information about the cluster paths.

### 3.4.2 The Common Boundary

When describing the algorithms and properties for path and subtree queries, we will often encounter a special vertex, so we also give it a name. Given any two vertices $u$ and $v$ that are connected (in the same tree) in $F$, we define the common boundary of $u$ and $v$.

**Definition 4** (Common boundary)**.** Given an RC-Tree for an input forest $F$ and a pair of vertices $u$ and $v$ that are connected in $F$, the common boundary is the representative vertex of the lowest common ancestor of the clusters $U$ and $V$, where $U$ and $V$ are the clusters represented by $u$ and $v$ respectively.

What does this vertex look like? Imagine two distinct connected vertices $u$ and $v$ and the clusters that contain them. Initially, $u$ and $v$ are in different clusters (their respective base clusters), and at some point in the contraction process they must be contained in the same cluster since they are connected. Imagine with the help of Figure 3.14 the last point in time before $u$ and $v$ are contained in the same cluster (i.e., the largest clusters that contain $u$ and $v$, but not both). By construction at this point, $u$ and $v$ are contained within a pair of clusters that share a single common boundary vertex. This common boundary vertex is the vertex that contracts to become the representative of the common cluster. Another way to define the common boundary would therefore be as the representative vertex of the smallest cluster containing both $u$ and $v$.

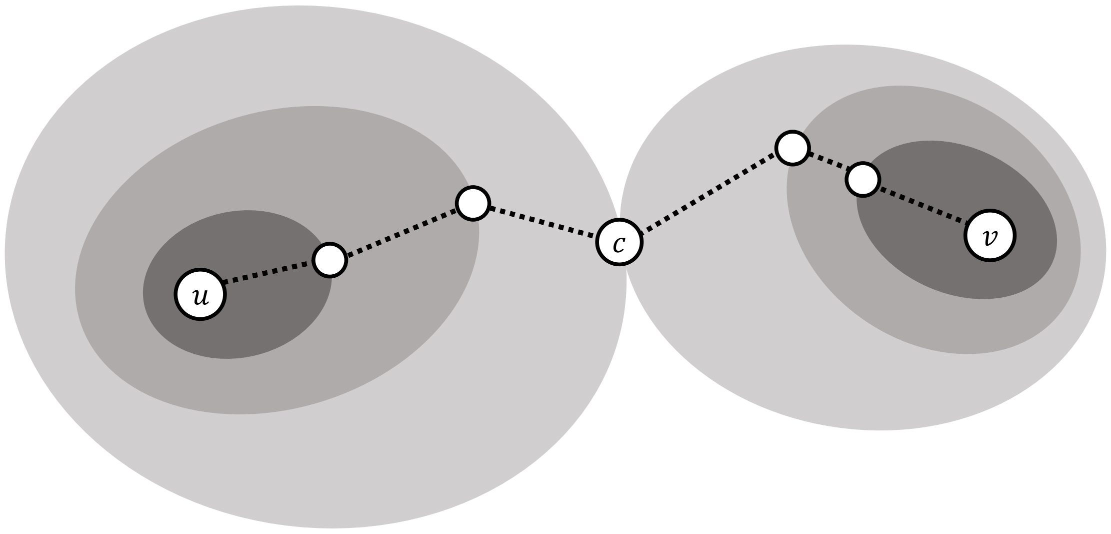


**Figure 3.14: The common boundary of two vertices $u$ and $v$, the representative vertex of the smallest cluster that contains both $u$ and $v$.**

The common boundary $c$ can be found by simply walking up the tree from $u$ and $v$ until they meet at their lowest common ancestor. Note that an edge case is possible where the lowest common ancestor of $U$ and $V$ is one of $U$ or $V$. This means that $u$ or $v$ is a boundary vertex of one of the clusters containing the other one.

### 3.4.3 Path Decompositions and Queries

One of the first applications of dynamic trees was solving path queries as a subroutine for maximum network flow algorithms [163]. Path queries involve endowing the data structure with an associative and commutative binary operation (e.g., sum, minimum, maximum, or anything more complicated) and labeling the vertices and/or edges of the graph with a weight. A path query then asks for the sum with respect to the binary operation of all of the edge weights on the path in $F$ between a given pair of vertices $u$ and $v$.

The key fact that allows RC-Trees to handle path operations is the following *path decomposition* property. Every path in the underlying forest $F$ can be represented by a small set of cluster paths that are all located adjacent to a corresponding path in the RC-Tree.

**Theorem 9** (Path decomposition property)**.** Given an RC-Tree for an input forest $F$ on $n$ vertices and a pair of vertices $u$ and $v$ that are connected in $F$. Let $U$ be the cluster represented by $u$ and $V$ be the cluster represented by $v$, and let $P$ be the path connecting $u$ and $v$ in $F$. There exists a set of disjoint binary RC Clusters such that:

1. the union of their cluster paths is exactly $P$,
2. each cluster is a direct child of the path in the RC-Tree between $U$ and $V$.

**Corollary 1** (Path queries)**.** Consider an RC-Tree for a weighted input forest $F$ on $n$ vertices and an associative and commutative operator $f$ over the weights. Then, we can maintain augmented values on the clusters of the RC-Tree such that given any pair of connected vertices $u$ and $v$, we can compute the sum of the weights along the path between $u$ and $v$ with respect to $f$ in time proportional to the height of the RC-Tree.

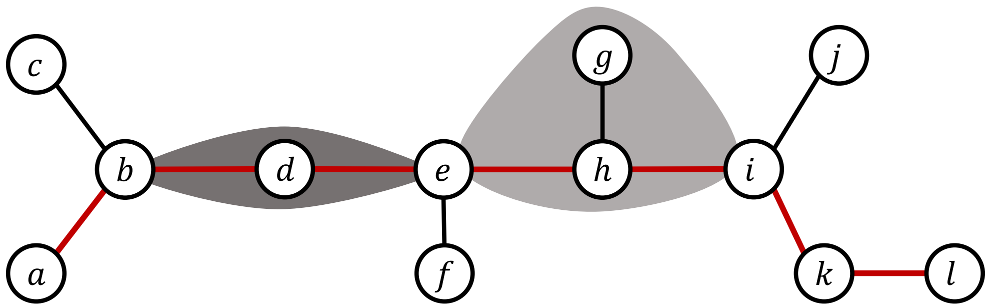
An example of a path decomposition in the RC-Tree is shown in Figure 3.15.

*Proofs.* We prove Theorem 9 constructively by describing how to find such a set of clusters. The algorithm for path queries (Corollary 1) follows similarly. To simplify, we start by considering the common boundary $c$ of $u$ and $v$. Let $U, V, C$ be the clusters represented by $u, v, c$ respectively. We aim to build a path decomposition of the path from $u$ to $c$ and $v$ to $c$, then take their union to obtain a path decomposition of $u$ to $v$. A path decomposition of $u$ to $c$ must consist of binary clusters that are children of the RC-Tree path from $U$ to $C$.

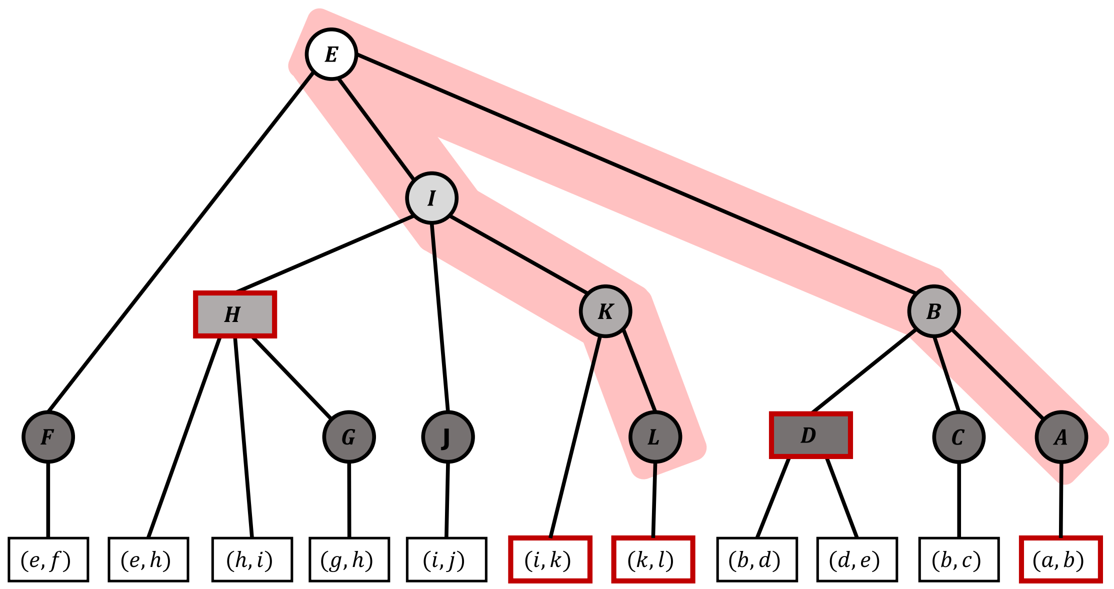
We can show inductively that for any vertex $u$ and any cluster $B$ containing $u$ with boundary vertex $b$, there is a set of binary clusters that are children of $U$ to $B$ in the RC-Tree whose cluster paths form the path $u$ to $b$ in $F$. Figure 3.16 shows an example of the resulting paths.

Consider the first cluster to contain $u$. If $u$ contracts via rake then it forms some unary cluster $B$ with boundary vertex $b$. This rake operation must rake the edge $(u, b)$ which is represented by a binary cluster with boundary vertices $u$ and $b$ and whose cluster path is therefore the path between $u$ and $b$ in $F$. This cluster is a child of $B$ and hence this is a valid path decomposition as described. If instead $u$ contracts via compression and is contained in a binary cluster $B$, we consider either of its boundary vertices $b$. One of the two edges that compressed is $(u, b)$, which is represented by a binary cluster with boundaries $u$ and $b$, and hence a cluster path that covers $u$ to $b$ in $F$.

Now suppose for the purpose of induction that for a cluster $B$ containing $u$ we know a path decomposition from $u$ to the boundary vertices of $B$. We want to show that for the parent cluster $B'$ and any of its boundary vertices $b'$ that there is a path decomposition from $u$ to $b'$. We will show that the new path decomposition consists of the old path decomposition



**(a) The path from $a$ to $l$ in the forest can be decomposed into the cluster paths of the RC clusters $(a,b), D, H, (i.k), (k,l)$.**



**(b) The clusters in the decomposition are children of the path between $A$ and $L$. The path in the RC-Tree is highlighted in red and the chosen clusters have a bold red outline.**

**Figure 3.15: An example of a path decomposition in an RC-Tree.**

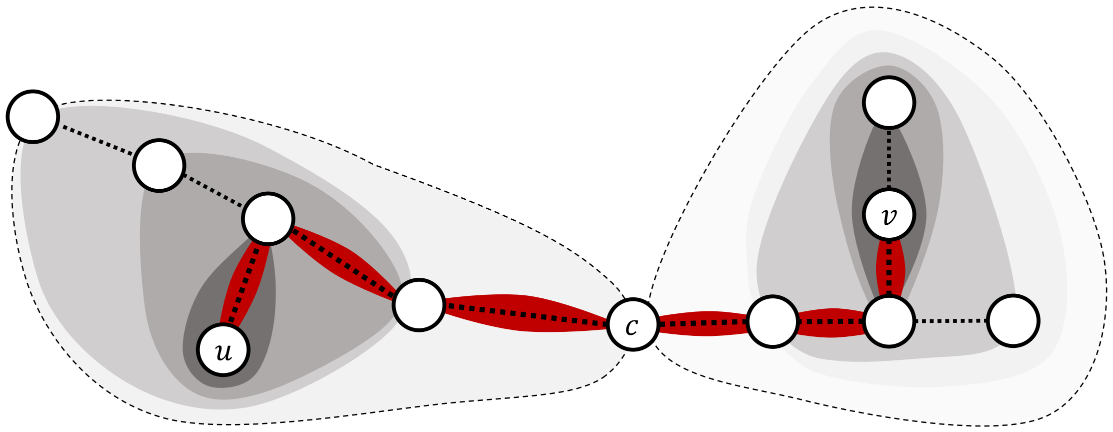


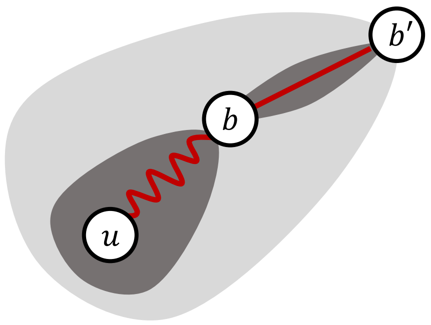
**Figure 3.16: A path decomposition obtained by finding a set of binary clusters connecting $u$ and $v$ to their ancestors' boundary vertices until they meet at the common boundary $c$**

with at most one additional cluster path added. Consider four cases depending on whether $B$ and $B'$ are unary or binary clusters (or a combination of both).

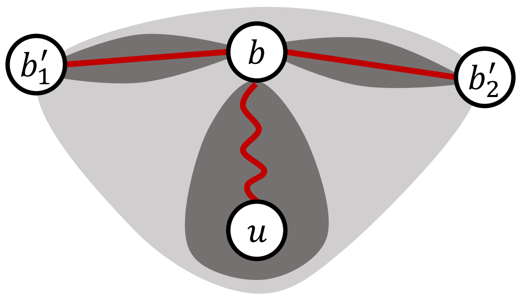
1. **Unary and unary**: $b$ contracts and rakes onto $b'$ via an edge $(b, b')$ which corresponds to a binary cluster with boundaries $b$ and $b'$, and hence there is a path decomposition consisting of the previous path decomposition plus this cluster, which is a child of $B'$.
2. **Unary and binary**: $b$ contracts and compresses the two edges $(b'_1, b)$ and $(b'_2, b)$, corresponding to binary clusters with boundaries $b$ and $b'_1$, and $b$ and $b'_2$. The resulting boundaries of $B'$ are $b'_1$ and $b'_2$, for which there are path decompositions consisting of the previous path decompositions plus the cluster corresponding to $(b'_1, b)$ and $(b'_2, b)$ respectively, each of which are children of $B'$.
3. **Binary and unary**: If $B$ has boundaries $b_1$ and $b_2$, then one of them contracts and rakes the edge $(b_1, b_2)$ onto the other one, leaving it as the sole boundary vertex of $B'$. We therefore already know the path decomposition for this boundary.
4. **Binary and binary**: If $B$ has boundaries $b_1$ and $b_2$, assume without loss of generality that $b_1$ contracts and compresses the two edges $(b_1, b_2)$ and $(b'_1, b_1)$ resulting in an edge $(b'_1, b_2)$, which corresponds to a binary cluster with boundaries $b'_1$ and $b_2$. A path decomposition for $b'_1$ consists of the previous path decomposition and the cluster corresponding to $(b'_1, b_2)$. We already know a path decomposition of $b_2$ since it did not change.

The cases are illustrated in Figure 3.17. By induction, we can find a path decomposition from $u$ to any of its ancestors' boundaries, and hence there is a path decomposition for $u$ to $c$, the common boundary, and similarly for $v$ to $c$.

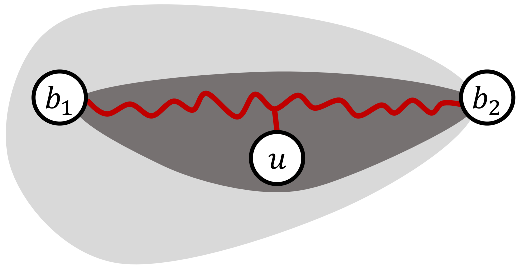
Corollary 1 follows by storing the sums of the weights on the cluster paths on each binary cluster. The algorithm then walks up the RC-Tree from $U$ to $C$ and maintains the sum from $u$ to the current boundary vertices by summing the augmented values of the clusters in the path decomposition. Combining the sums from $u$ to $c$ and $v$ to $c$ gives the answer. □



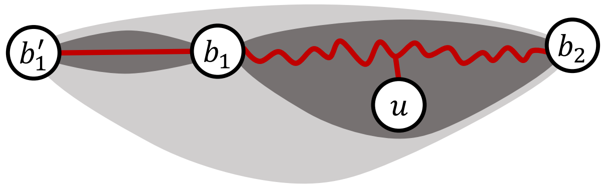
**(a) Case 1: A unary cluster with a unary parent.**



**(b) Case 2: A unary cluster with a binary parent.**



**(c) Case 3: A binary cluster with a unary parent.**



**(d) Case 4: A binary cluster with a binary parent.**

**Figure 3.17: Building a path decomposition inductively.**

### 3.4.4 Subtree Decompositions and Queries

Subtree queries supported by dynamic tree data structures typically come in one of two flavors. Either the underlying tree has a particular fixed root (which can sometimes be changed by an explicit re-root operation), or it is a free/unrooted tree. In the rooted case, subtree queries take a single vertex $u$ and ask for the sum of the weights on the vertices or edges in the subtree rooted at $u$. For unrooted trees, a single vertex is not enough information to define a subtree, so a query will typically take two arguments: a root vertex for the subtree $u$, and the parent vertex $p$. This kind of query is more general but consequently more tricky to implement since the orientation of a subtree could change between queries without being able to pre-process the tree for a particular root in advance.

Similar to the path decomposition property, RC-Trees are robust enough to handle arbitrarily rooted subtree queries because of a powerful *subtree decomposition* property that says any rooted subtree in $F$ can be represented by a small set of clusters that are all adjacent to a corresponding path in the RC-Tree.

**Theorem 10** (Subtree decomposition property)**.** Consider an RC-Tree for an input forest $F$ on $n$ vertices and a rooted subtree $S$ of $F$ defined by a pair of vertices $u$ and $p$, such that $S$ is the subtree rooted at $u$ if $F$ were rooted at $r$. Let $U$ be the RC Node represented by $u$. There exists a set of RC Clusters such that

1. the union of their contents is exactly $S$,
2. each cluster is a direct child of the path in the RC-Tree from $U$ to its root cluster.

**Corollary 2** (Subtree queries)**.** Consider an RC-Tree for a weighted input forest $F$ on $n$ vertices and an associative and commutative operator $f$ over the weights. Then, we can maintain augmented values on the clusters of the RC-Tree such that given any rooted subtree $S$ of $F$, we can compute the sum of the weights of the contents in $S$ with respect to $f$ in time proportional to the height of the RC-Tree.

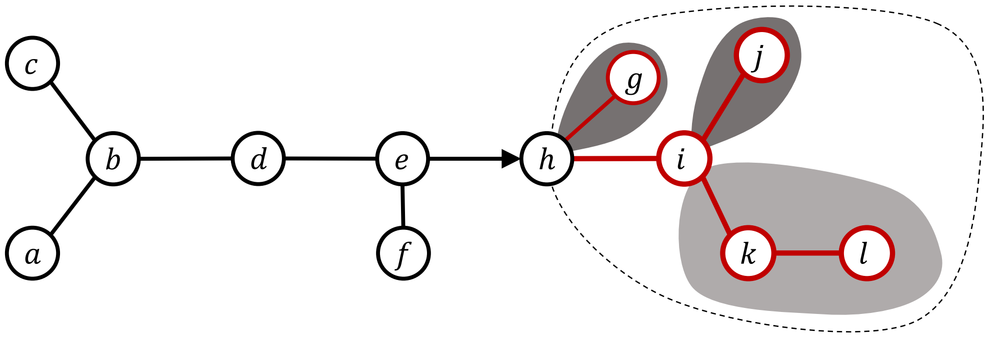
Acar et al [5] sketch an algorithm for subtree queries, though it appears to be incomplete and does not cover all possible cases. An example of a subtree decomposition and the corresponding clusters in the RC-Tree is shown in Figure 3.18. We now prove a lemma that will come in handy during the proof. Consider some cluster $U$ represented by $u$ and one of its boundary vertices $b$. We define the "subtree growing out of $b$" with respect to $u$ to be the subtree rooted at $b$ oriented such that the tree was rooted at $u$.

**Lemma 1.** Consider a cluster $U$ represented by $u$, and a boundary vertex $b$. The subtree growing out of $b$ with respect to $u$ as the tree root can be decomposed into a set of disjoint RC clusters which are all direct children of the path from $U$ to the root cluster.

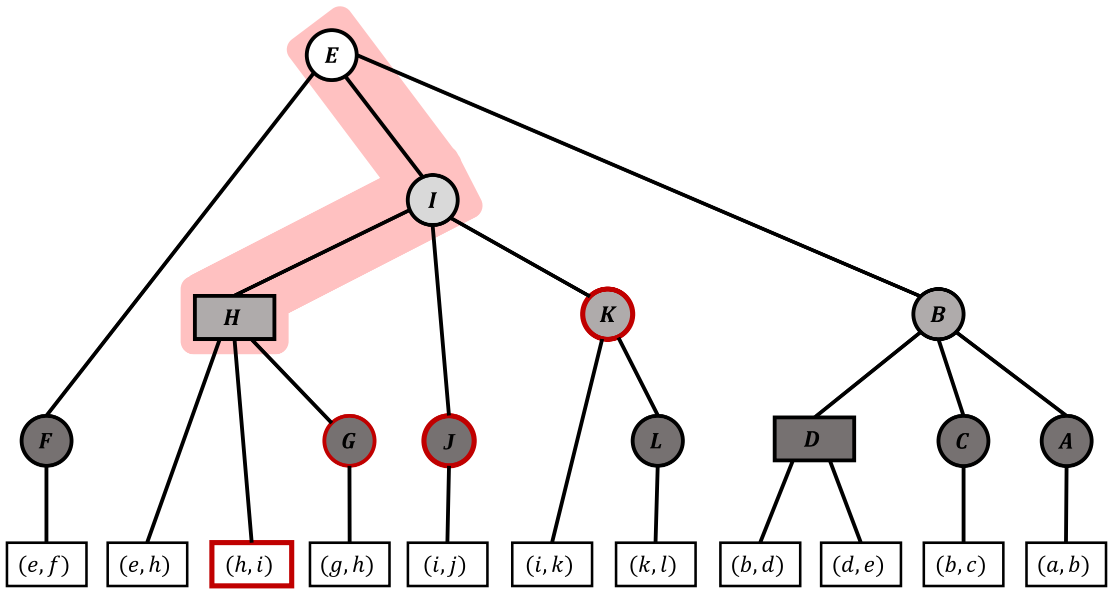
*Proof.* The situation is depicted in Figure 3.19. The goal is to collect a set of disjoint clusters that covers the subtree rooted at $b$, all of which are on the RC-Tree path from $U$ to the root cluster. To do so, observe that when $b$ contracts, it forms a larger cluster $B$. This cluster either contains $U$ as a direct child, or possible as a descendent further down the tree if the other boundary vertex of $U$ contracted earlier. $B$ is therefore an ancestor of $U$, and furthermore, all of the other children of $B$ must be contained inside the subtree of interest.

We now consider the boundary vertices of $B$. If $B$ shares a boundary vertex with its child that contains $u$, we ignore that, since it is on the opposite side of $u$ to $b$ on the left side of $u$ in Figure 3.19) and hence is not contained in the subtree of interest. If $B$ has no other boundary vertex, then $B$ therefore contains the entirety of the subtree of interest and we are done. Otherwise, if $B$ has any other boundary vertex $b'$ (it may have up to two if $B$ is a binary cluster and the child containing $U$ is a unary cluster), it is on the opposite side of $b$ to $u$ (on the right side of $b$ in Figure 3.19), and hence is contained in the subtree of interest. We therefore simply recursively repeat this process with $b'$ (possibly two of them), identifying the ancestor $B'$ of $B$ where $b'$ contracts, and collecting all of its children except the one containing $b$. Once we run out of boundary vertices on the side of $b$, we have completed the subtree, and since every cluster collected was a child of an ancestor of $U$, it is a valid decomposition. □

*Proofs of Theorem 10 and Corollary 2.* We prove Theorem 10 constructively, similar to Theorem 9. The key idea is depicted in Figure 3.20. Consider the vertex $u$ and the cluster $U$ that it represents. $U$ consists of a constant number of child clusters, with at least one and at most two binary children, and some number of unary children. The subtree rooted at $u$ with respect to $p$ essentially consists of the entire tree except for anything in the direction of $p$. Furthermore, since $p$ is adjacent to $u$, we know that $p$ is either contained within one of the child clusters, or it is one of the boundary vertices. The second case happens if a binary child of $U$ is a single-edge base cluster with $p$ as the other endpoint.



**(a) The subtree rooted at $h$ oriented with respect to its parent $e$ can be decomposed into the clusters $\{h\}, G, (h,i), J, K, \{i\}$. The base clusters $\{h\}$, $\{i\}$ would be children of $H$ and $I$ if represented explicitly.**



**(b) The clusters in the decomposition are children of the path from $H$ to its root cluster. The path in the RC-Tree is highlighted in red and the chosen clusters have a bold red outline.**

**Figure 3.18: An example of a subtree decomposition in an RC-Tree.**

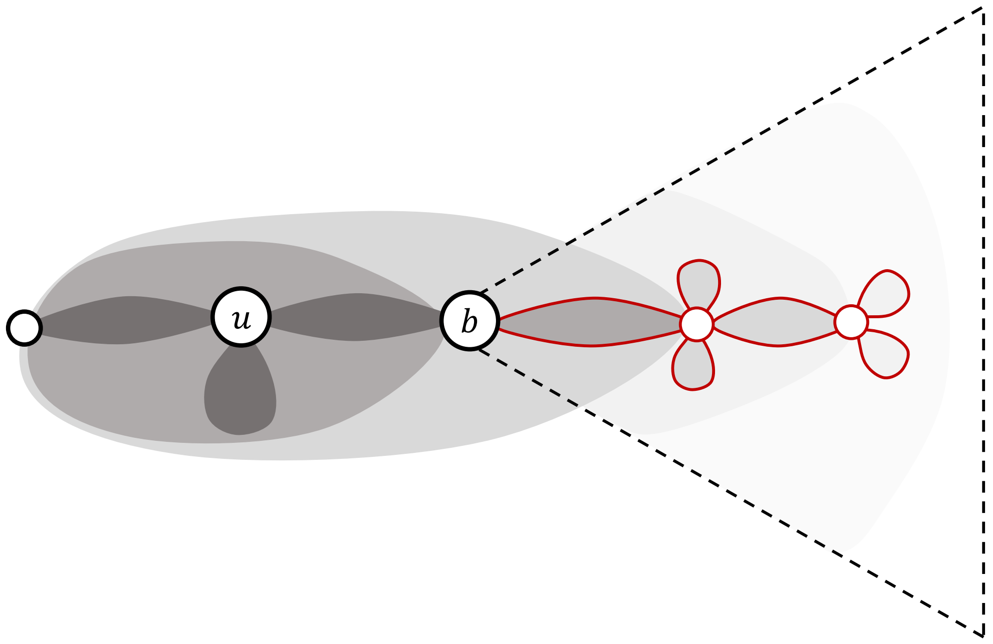


**Figure 3.19: Constructing a decomposition of the subtree growing out of $b$ with respect to $u$ as the tree root. The process essentially walks along the boundary vertices to the right by following their contractions in the RC-Tree and collecting all of the adjacent clusters along the way.**

So, to construct a subtree decomposition, we start by taking all of the children of $U$, except for the one that contains/is adjacent to $p$. Then, for each binary child with boundary $b$, unless it is the one that contains/is adjacent to $p$, we add to the decomposition the contents of the subtree growing out of $b$. By Lemma 1, the subtree growing out of $b$ is decomposable into children of the path from $U$ to the root cluster, so this results in a valid subtree decomposition.

Lastly, to support subtree queries and conclude Corollary 2, we augment each cluster with its total weight. The algorithm then walks up the RC-Tree from $U$ and aggregates the total weight of the clusters in the decomposition. This consists in taking the total weight on all of the children of $U$ except the one containing/adjacent to $p$, then summing the weights on clusters that make up the subtrees growing out of the boundary vertices as per Lemma 1. □

## 3.5 Batch Queries on RC-Trees

Sequential/single queries on RC-Trees typically consist in aggregating some information along a path or set of paths within the RC-Tree. Most commonly, a query will begin with a set of RC nodes and then sweep upwards towards either their common boundary, or all the way to the root cluster, then sometimes propagate some information back down the tree along the same paths. Parallel batch-dynamic RC-Trees afford us the opportunity to implement batch-queries that answer multiple queries in work that is at most or better than the work of answering each query individually, and in low span.

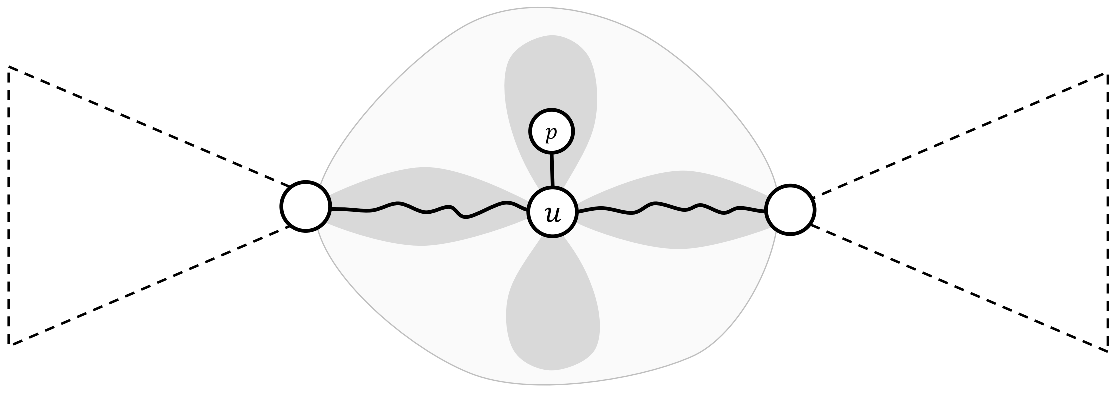


**Figure 3.20: Constructing a subtree decomposition of the subtree rooted at $u$ with respect to $p$ as the parent. The subtree consists of (1) the adjacent clusters except the one containing/adjacent to $p$ and (2) the subtrees growing out of the boundary vertices unless adjacent to the cluster containing/adjacent to $p$.**

The simplest way to implement a batch-query would be to simulate the sequential query concurrently for each one in the batch, provided that queries are read-only and do not modify the RC-Tree. This however doesn't take advantage of batching to reduce work, which is a key desired feature of batch-dynamic algorithms. To save work, when multiple starting nodes in a traversal share some common ancestor, it is important to save work by *not* processing that ancestor separately for each of them. The key ingredient in obtaining efficient *batch queries* will be eliminating this potential redundant work. Figure 3.9 shows the kind of traversals that are typical of batch queries on an RC-Tree. When several node-to-root paths intersect, those ancestors should be processed at most once each. If this can be successfully achieved, then Theorem 8 implies that the work of $k$ queries will be bounded by $O\left(k + k\log\left(1+\frac{n}{k}\right)\right)$, and by parallelizing by levels, the span will be $O(\log n)$. Note that the extra additive $k$ term comes from the fact that some queries might have $k = \Omega(n^2)$.

Acar et al. [5] sketch how to perform single-query versions of path queries, subtree queries, diameters, LCAs, centers and medians, and nearest marked vertices on sequential RC-Trees. Alstrup et al. [16] categorize these operations as *local* and *nonlocal*. A local property of a tree is one such that if an edge or vertex has a property in a tree, then it has that property in all subtrees that it appears in. For instance, being the heaviest edge is a local property.

## Fundamental techniques: Bottom-up and top-down computations

In this section, we will demonstrate the fundamental techniques for implementing batch queries by describing a simple algorithm for batch connectivity queries, followed by more complex examples of batch subtree queries, batch LCA queries, and invertible batch path queries. A batch query typically consists of *two kinds* of computations, which we will distinguish as *bottom-up* and *top-down* computations. Some queries use only one or the other, while others perform both, typically a bottom-up followed by a top-down.

A *bottom-up* computation is one in which every cluster wants to compute some data which is a function of its children. For example, computing the sum of the edge weights on the cluster path of a binary cluster is a bottom-up computation, because it can be implemented by summing the edge weights of the cluster paths of its two binary children (unless it is a leaf cluster, in which the weight is simply the weight of the edge). Bottom-up computations are not performed at query time, but rather they are stored as *augmented values* on the clusters. This means that they are computed at build time and then maintained during update operations. They will then be available precomputed for queries to utilize.

A *top-down* computation is one in which every cluster wants to compute some data which is a function of its *ancestors*, most commonly its boundary vertices. Note that the boundary vertices of a cluster *always* represent ancestors of that cluster, and furthermore that one of them is guaranteed to represent the parent cluster. Top-down computations can not be efficiently stored as augmented values since updating a cluster high in the tree could require updating all the descendants of that cluster which would cost linear work. Instead, top-down computations are always performed at query time. The query algorithm is responsible for identifying the set of clusters for which the data is needed, which for $k$ queries typically consists of some $O(k)$ clusters and all of their ancestors, and hence by Theorem 8 is at most $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ clusters in total. The algorithm then traverses the RC-Tree from the root cluster (or multiple root clusters in parallel in the case of a disconnected forest), visits those relevant clusters, and computes the desired data from the data on the ancestors.

For algorithms that utilize both, the bottom-up and top-down computations are typically not independent. Our batch-query algorithms will often maintain a bottom-up computation using augmented values and then proceed at query time by performing a top-down computation that makes use of these values. This combination strategy is what allows us to derive some of the more complicated batch-query algorithms.

### 3.5.1 Batch Connectivity Queries

A batch-connectivity query over a forest $F$ takes as input a sequence of pairs of vertices $u, v$ and must answer for each of them whether they are connected, i.e., whether they are contained in the same tree. That is, we want to support the following.

- **BatchConnected**$(\{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\})$ takes an array of tuples representing queries. The output is an array where the $i^{\text{th}}$ entry is a boolean denoting whether vertices $u_i$ and $v_i$ are connected by a path in $F$.

Connectivity queries are arguably the simplest kind of query since they require no augmented data to be stored on the clusters and no additional auxiliary data structures; only the structure of the RC-Tree is needed to determine the answer. The main ideas however are valuable and will be re-used in the later more complicated algorithms

In the sequential setting, the query is quite simple due to the fact that a pair of vertices $u$ and $v$ are connected if and only if they are in the same RC-Tree component. To determine which component vertex $v$ is in, it suffices to simply walk up the RC-Tree starting at the cluster represented by $v$ until reaching the root cluster and noting its representative. $u$ and $v$ are in the same component if and only if they have the same representative root cluster.

To generalize this for the batch setting we will use the same technique. We reduce the problem of solving a batch of connectivity queries to given a batch of vertices, find the

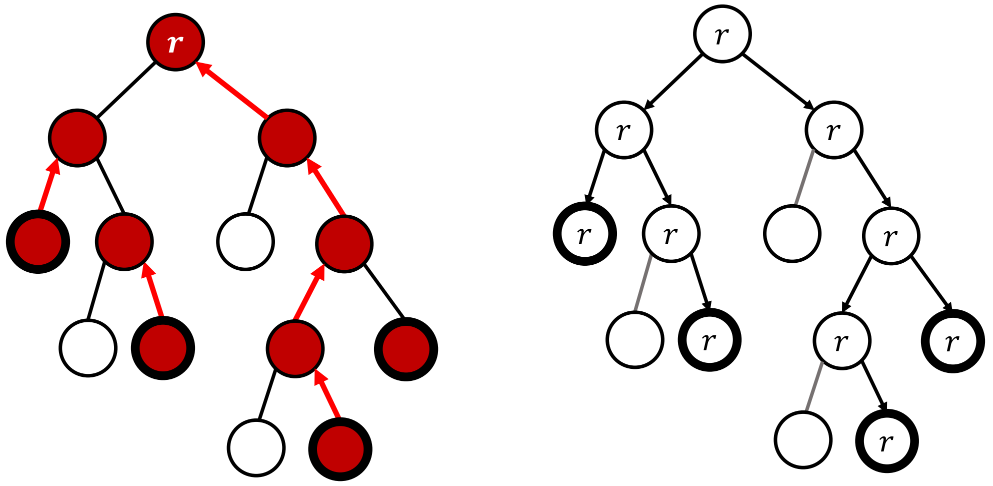


**Figure 3.21: An example execution of the batch find-representative operation for one connected component. The initial query nodes (shown as circles with thicker outlines) traverse to the root (red arrows), but only the first to arrive at each ancestor proceeds. After arriving at the root node, the winning node propagates the answer back down the tree in parallel.**

representatives of their root clusters. This information can then solve the batch connectivity problem by checking for each pair whether they have the same representative.

The simplest solution would be for each query vertex to concurrently start at the corresponding node and walk up the RC-Tree until we find the root representative. However, this would perform $\Theta(k \log n)$ work since it makes no attempt to reduce redundant work. To optimize this, we want paths in the RC-Tree that intersect to only be evaluated once.

## Algorithm: Batch find-representative

Our approach to performing the representative finding algorithm is to utilize concurrency. We assume that each node of the RC-Tree is augmented with two additional fields: a boolean flag and a field that will store the answer to the query for that component. Initially, in parallel for each starting node of the query, the algorithm walks up the RC-Tree setting the boolean flag with an atomic TESTANDSET. If a task fails the test-and-set because it was beaten by another node, it stops walking up the tree.

After every node has completed its walk and marked the relevant ancestors, the algorithm performs a top-down computation on all of the flagged nodes, copying the identity of the root node (the answer to the query) into the answer field on each node and unsetting the flag for the next query. The query nodes can then each read the answer out of their respective fields. An illustration of this algorithm is depicted in Figure 3.21.

**Theorem 11** (Batch connectivity queries)**.** Given a balanced (resp. in expectation) RC-Tree representing a forest on $n$ vertices, a batch of $k$ connectivity queries can be answered in $O\left(k + k\log\left(1+\frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log n)$ span (resp. w.h.p.).

*Proof.* The batch find-representative algorithm performs work proportional to the number of unique nodes visited, which by Theorem 8 is $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ (in expectation if the RC-Tree is randomized) for $k$ queries. The span is the height of the RC-Tree, which is $O(\log n)$ for a balanced RC-Tree (w.h.p. if the RC-Tree is randomized). The algorithm answers a batch of find-representative queries, which is sufficient to answer a batch of connectivity queries by checking whether the representatives of each pair are the same. Therefore the final work to answer the connectivity queries is $O\left(k+k\log\left(1+\frac{n}{k}\right)\right)$. □

## 3.5.2 Batch Subtree Queries

In a weighted unrooted tree, a subtree query takes a subtree root $u$ and a parent $p$ and asks for the sum of the weights of the vertices/edges in the subtree rooted at $u$, assuming the tree is rooted such that $p$ is the parent of $u$. The weights can be aggregated using any predefined associative and commutative operator (i.e., the weights are from a commutative semigroup), such as minimum, maximum, or sum. Weights can be present on vertices or edges or both.

- **BatchSubtreeWeight**$(\{(u_1,p_1),\ldots,(u_k,p_k)\})$ takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry contains the sum over the commutative semigroup operation of the contents of the subtree rooted at $u_i$ relative to the parent $p_i$.

Like connectivity, our algorithm for batch subtree queries is similar to the sequential single-query algorithm but with the added care to avoid performing redundant work and redundant computations. Naively running $k$ subtree queries in parallel would result in a lot of redundant work since many subtrees have substantial overlap, so our goal is to avoid that redundancy. Unlike connectivity, it is much less obvious how to do so, since different subtree queries, although they may traverse the same root-to-leaf path in the RC-Tree, may accumulate different chunks of information along the way. Batch connectivity queries on the other hand were simpler since the answer for every RC node on the same path was guaranteed to be the same. Figuring out precisely how to break up the computation to avoid redundancy will be the key insight of this algorithm.

Recall the sequential single-query algorithm of Section 3.4.4. The key idea was that a subtree rooted at $u$ can be decomposed into (1) all but one of the clusters adjacent to $u$ when $u$ contracts (i.e., all but one of the children of the cluster $U$ represented by $u$), and (2) one or two subtrees growing out of the boundary vertices of $U$. Refer to Figure 3.20 for a reminder. The contributions of Part (1) can be computed in constant time since those clusters are children of $U$ in the RC-Tree. The complexity therefore lies in batching the computation of Part (2), the contributions of the subtrees growing out of the boundary vertices.

### Algorithm: Batch subtrees

The algorithm begins with a bottom-up computation that stores on each cluster the total aggregate weight of the contents of that cluster. This is stored as an augmented value and hence already available at query time. The key step of the batched algorithm is the subsequent top-down computation. Specifically, our algorithm computes for every relevant cluster, the contributions of the subtrees growing out of its boundary vertices. The first step is the same as the batch find-representative algorithm. Starting from every query vertex $u_i$, it walks up the tree concurrently and marks every ancestor of every $u_i$. This marks out a subtree of the RC-Tree consisting of $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ relevant clusters, specifically, every boundary vertex of every ancestor of every query vertex $u_i$.

The algorithm computes these values starting at the top of the RC-Tree with a top-down computation. Given a node $U$ with representative $u$, we want to compute for each of its boundary vertices $b$, the total weight in the subtree growing out of $b$. Consider the node $B$ represented by $b$. $B$ is an ancestor of $U$ in the RC-Tree and hence the total contributions of the subtrees growing out of its boundaries have already been computed. $B$ has one child $U'$ which is either $U$ itself or an ancestor of $U$. The algorithm collects the contributions of every other child of $B$, since these are contained in the subtree growing out of $b$. Then, it considers each boundary of $B$, and for each boundary $b'$ of $B$ *not shared with* $U'$, it recursively adds the contribution of the subtree growing out of $b'$. Since the value of the subtree growing out of $b'$ was computed earlier, looking this up takes constant time. Therefore, the contribution of the subtree growing out of $b$ with respect to $u$ can be computed in constant time.

After completing this top-down computation, the algorithm can then answer every query $(u,p)$ in parallel by summing the contributions of the children of $U$ except the one containing/adjacent to $p$ and then adding the contributions from the top-down computation of the subtrees growing out of the boundary vertices of $U$ except for at most one that is adjacent to the cluster containing/adjacent to $p$. Thus we have the following theorem.

**Theorem 12** (Batch subtree queries)**.** A balanced (resp. in expectation) RC-Tree for a forest on $n$ vertices with weights from a commutative semigroup can be augmented to solve a batch of $k$ subtree sum queries in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log n)$ span (resp. w.h.p.).

*Proof.* The preprocessing step visits every ancestor of every query root $u$, of which there are at most $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ (in expectation if randomized) according to Theorem 8. The top-down computation then examines the children of each node, of which there are a constant number, and looks up the contributions of at most four previously computed subtrees (there are at most two boundary vertices of the current node, and for each they consider the at most two boundary vertices of the corresponding ancestor node). Therefore a constant amount of work is required per node, resulting in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ preprocessing work (in expectation if randomized), and by parallelising the top-down traversal, a span of $O(\log n)$ (w.h.p. if randomized). After preprocessing, each query $(u,p)$ is then answered in constant time by examining the constant number of children of $U$ and looking up at most two precomputed contributions for the boundary vertices. Since there are at most $O(n)$ possible subtrees, the final work bound is $O\left(k+k\log\left(1+\frac{n}{k}\right)\right)=O\left(k\log\left(1+\frac{n}{k}\right)\right)$. □

### 3.5.3 Batch LCA Queries

Lowest common ancestors (LCAs) are a useful subroutine for several tree and graph algorithms, and there exists a wide variety of algorithms for the problem including parallel algorithms [157] and algorithms that work on dynamic trees [163]. Link/cut trees [163, 164], Euler-tour trees [96], and sequential RC-Trees [5] are all able to solve LCA queries. However, we know of no existing algorithm that can efficiently solve *batches* of LCA queries on a dynamic tree. We will describe how to do just this using parallel RC-Trees.

- **BatchLCA**($\{(u_1, v_1, r_1), \ldots, (u_k, v_k, r_k)\}$) takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry is the LCA of $u_i$ and $v_i$ with respect to the root $r_i$.

The following well known fact will be useful.

**Lemma 2.** Given an algorithm that can compute LCA($u, v$) with respect to an arbitrary fixed root vertex, the value of LCA($u, v, r$) for any given root $r$ can be computed in constant time from LCA($u, v$), LCA($u, r$), and LCA($v, r$).

Given this fact, we can simplify our batch algorithm by describing an algorithm that computes LCA($u, v$) with respect to the root vertex of the RC-Tree (the representative vertex of the root cluster). A batch of general LCA queries with arbitrary roots can then be reduced to a batch of LCA queries with respect to the RC-Tree root with constant overhead. We will also use the following two tools.

**Lemma 3** (Parallel LCA [157])**.** Given a rooted tree $T$ on $n$ vertices, with $O(n)$ work and $O(\log n)$ span preprocessing, LCA queries can be answered in $O(1)$ time.

We will also make use of *level ancestors*. In the level ancestor problem we are given a static rooted tree $T$ and wish to preprocess it such that we can answer the following question: given a vertex $u$ and an integer $i$, what is the $i^{\text{th}}$ vertex on the path from $u$ to the root? The following result will be very useful.

**Lemma 4** (Level Ancestors [21])**.** Given a rooted tree $T$ on $n$ vertices, with $O(n)$ work and $O(\log n)$ span preprocessing, level ancestor queries can be answered in $O(1)$ time.

We will begin by describing a useful property that helps us find the LCA of a pair of vertices $u$ and $v$. Let $r$ be the root of the RC-Tree, and let $U$ and $V$ be the clusters represented by them respectively. Let $c$ be their common boundary, which represents the cluster $C$. There will be a few cases depending on $c$ and the location of $r$. Assume that $c$ is equal to one of $u$ and $v$. Without loss of generality, say that $c = v$. Then $v$ is a boundary vertex of one of the ancestors of $c$. If $C$ is a unary cluster, then since $v$ is a boundary vertex and $u$ is contained inside the cluster, $v$ must be the LCA of $u$ and $v$. Note that this is true because $r$ can not also be inside the cluster, since as the root of the RC-Tree, it contracted last. If $C$ is a binary cluster, then there are two possibilities. Either $r$ is on the opposite side of $v$ (or is equal to $v$) in which case $v$ is the LCA, or $r$ is on the opposite side of the other boundary vertex.

In the second case, observe that there is a path from $r$ to $v$ that goes along the cluster path of $C$. Therefore the LCA must be the vertex on the cluster path of $C$ that is closest to $u$. This means that there is a unary cluster containing $u$ that rakes onto the cluster path. Specifically, the target of that rake operation must be the LCA. To locate this vertex, observe that after the cluster containing $u$ rakes onto the cluster path, all subsequent ancestor clusters until $C$ must be binary clusters since they lie on the cluster path. The LCA can therefore be determined by identifying *the first (highest) unary cluster* on the path in the RC-Tree from $C$ to $U$. The boundary vertex of this cluster, i.e., the representative of its parent, is the LCA.

Now suppose that $c$ is neither of $u$ or $v$, so there are two clusters containing $u$ and $v$ respectively which merge with the vertex $c$. If $c = r$, then $c$ is the LCA. Otherwise, $r$ must be on the opposite side of one of the boundary vertex of $C$. The location of the LCA depends on which boundary vertex this is. If this boundary vertex belonged to a child of $C$ that does not contain either $u$ or $v$, then $c$ must be the LCA. Otherwise, assume without loss of generality that $r$ is on the other side of the boundary vertex belonging to the cluster that contains $u$. Call this cluster $U'$. This implies once again that there is a path from $r$ to $v$ that goes through the entire cluster path of $U'$, which contains $u$. Therefore the LCA is again the vertex on the cluster path closest to $u$, which can be identified by locating the first (highest) unary cluster on the path in the RC-Tree from $U'$ to $U$ and returning its boundary vertex.

## Algorithm: Batch LCA

Our algorithm for batch LCA does not require any augmented data to be stored on the clusters, so it is entirely a top-down computation. The idea is to determine for every binary cluster, which boundary vertex is closer to the root $r$, i.e., in which direction the root exists. Then, the case analysis above can be used to determine whether the LCA is $c$, the common boundary, or a vertex on the cluster path on some binary cluster. In the later case, we utilize bit tricks and the parallel level ancestor data structure (Lemma 4) to find the first unary cluster on the path in the RC-Tree in constant time. Implementing this step in constant time is important to eliminate the redundancy that would occur if each query had to search for that cluster separately, which would lead to a runtime of $O(k \log n)$.

The first step of the algorithm is the same as every other. It begins by marking all of the clusters that contain any of the vertices in the input, i.e., all of the ancestors in the RC-Tree of every cluster represented by any $u_i$ or $v_i$. This marks out a subset of the RC-Tree consisting of $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ nodes (Theorem 8). The algorithm takes this subset of the RC-Tree and computes a static LCA and level ancestor data structure on it using Lemmas 3 and 4 [21, 157].

Next, the algorithm computes two properties via a top-down computation. First, it computes for each binary cluster in the marked subset, which boundary vertex is closest to the root vertex. This is a simple top down computation to perform by considering the parent of the current cluster. If the parent of the current cluster is a unary cluster, then the closest boundary vertex is the one shared with the parent. If the parent is a binary cluster and the boundary shared with the current cluster is closest to the root, then the shared boundary is the answer, otherwise the other boundary is.

The second top-down property that the algorithm needs to know is for each marked cluster, which of its ancestors are binary and which are unary. Of course, each cluster has up to $O(\log n)$ ancestors, so we can not store an explicit set or list since this would take more than constant space per cluster and therefore violate our desired work bound of $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$. Instead, we can store this information using a *bitset*. That is, each cluster will store a bit sequence denoting the type of each of its ancestors in level order. For each unary cluster, write 1, otherwise write 0. Since there are at most $O(\log n)$ ancestors (w.h.p. if randomized), this can be stored inside a *constant* number of words (w.h.p. if randomized). Computing the bitset top-down can be implemented by simply reading the bitset of the parent then adding one additional bit corresponding to the type of the parent. This takes constant time and space and hence this can be implemented as an efficient top-down computation.

Finally we have all of the data needed to answer the queries. For each query $(u, v)$, the algorithm computes the common boundary $c$ using the static LCA data structure. If $c = r$ then $c$ is the LCA of $u$ and $v$. Otherwise, let $C$ be the cluster representing $c$. If $c$ is one of $u$ or $v$, without loss of generality suppose that $c = v$. If $C$ is a unary cluster, then the answer is $v$. If $C$ is a binary cluster and the boundary closest to the root is $v$, then $v$ is the LCA. If $C$ is a binary cluster and $v$ is not the boundary closest to $r$, then the LCA is the vertex on the cluster path of $C$ that is closest to $u$.

Otherwise, $c$ is neither of $u$ or $v$. Consider the boundary vertex of $C$ that is closest to the root (there may just be one if $C$ is a unary cluster), and consider the binary child $B$ of $C$ that shares this boundary vertex. If $B$ does not contain $u$ or $v$, then $c$ is the LCA. In the final case, the closest boundary to $r$ is adjacent to a binary cluster $B$ containing either $u$ or $v$. Without loss of generality say that this cluster contains $u$. The LCA is the vertex on the cluster path of $B$ that is closest to $u$.

In the non-trivial cases where the LCA is not equal to $c$, all that remains is to compute the closest vertex on the cluster path of a binary cluster to the vertex $u$. Call the binary cluster $B$. The answer is the boundary vertex of the first (highest) unary cluster on the path in the RC-Tree from $B$ to $U$. To locate this, the algorithm looks at the ancestor bitset of $U$. It masks out (set to zero) all of the ancestors that are higher than $B$, then locates the most significant bit (first 1 bit) in the resulting bitset. This identifies the level of the first unary cluster on the path. Finally, the identity of this cluster can be found by looking up the corresponding level ancestor using the level ancestor data structure. The boundary vertex of it is the LCA.

**Theorem 13** (Batch LCA queries)**.** Given a balanced (resp. in expectation) RC-Tree for a forest on $n$ vertices, a batch of $k$ LCA queries can be answered in $O\left(k + k\log\left(1+\frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log n)$ span (resp. w.h.p.).

*Proof.* The preprocessing of the marked nodes in the RC-Tree visits every ancestor of every query node, of which there are $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ (in expectation if randomized) according to Theorem 8. By Lemmas 3 and 4, it takes linear work and $O(\log n)$ span to build the LCA and level ancestor data structures. Since the input tree to these algorithms is just the marked nodes, this takes $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work (in expectation if randomized).

The top-down computations both only need to examine the parent of each marked node, so it takes constant time per node, for a total of $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work (in expectation if randomized) and $O(\log n)$ span (w.h.p. if randomized). Following the preprocessing and top-down computations, each query is answered in constant time since it takes constant time to query the LCA and level ancestor data structures, and constant time to perform the required bit operations on the ancestor bitsets. Therefore the algorithm answers $k$ LCA queries in $O\left(k + k\log\left(1+\frac{n}{k}\right)\right)$ work (in expectation if randomized) and $O(\log n)$ span (w.h.p. if randomized). By Lemma 2, LCA queries with respect to arbitrary root vertices can be reduced to at most three times as many LCA queries with respect to the RC-Tree root, which at most adds a constant factor overhead. $\square$

### 3.5.4 Batch Path Queries with Inverses

In the sequential single-query setting, path queries on RC-Trees were defined to operate on weighted vertices or edges where the weights are combined by any specified associative and commutative operation (formally, the weights form a commutative *semigroup*). This meant that a single algorithm could handle queries for the sum of the edge weights, the maximum/minimum edge weight, or the total weight with respect to any arbitrarily complicated associative and commutative operation. Unfortunately, this turns out to be *impossible* to do efficiently in the batch setting (we can not hope to achieve $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work.)

Tarjan [169] shows that the problem of verifying the edges of an MST with edge weights from a semigroup requires $\Omega((m+n)\alpha(m+n, n))$ time, where $\alpha$ is the inverse ackerman function. MST verification consists in querying for the minimum weight edge in a given spanning tree between the endpoints of the $\Theta(m)$ non-tree edges of the graph, and checking that the non-tree edge is no lighter than the MST edge. This can be reduced to a batch path query, and hence this implies a superlinear lower bound on batch path queries.

Fortunately, not all hope is lost. The MST verification problem, or more generally, the offline path query problem admits several efficient solutions both sequentially and parallel when additional assumptions are made about the weights. For example, if the weights have an inverse (i.e., we can perform subtraction), linear-work algorithms are plentiful. In this section, we will give an algorithm for batch path queries on RC-Trees when the weights admit an inverse (formally, the weights form a commutative *group*). This covers the case where the weights are real-valued numbers and the goal is to compute the sum of the weights along a path. It does not work for computing the minimum- or maximum-weight edge on a path since there is no inverse operation for the minimum or maximum operation.

- **BatchPathSum**$(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry contains the sum over the commutative group operation of the weights on the path between $u_i$ and $v_i$.

The main idea is a simple and classic technique involving the use of LCAs. Suppose we are able to compute the total weight of the path from the RC-Tree root to any other vertex in the tree. Then the total weight on the path between two arbitrary vertices $u$ and $v$ is the sum of the paths from the root to $u$, plus the root to $v$, less twice the weight of the path from root to LCA of $u$ and $v$. Using our batch LCA algorithm from Section 3.5.3, all that remains is to be able to compute the total weight from the root to any other vertex.

#### Algorithm: Batch path queries over a commutative group

We will reduce a batch of path queries over a commutative group to an invocation of our batch-LCA algorithm from Section 3.5.3 and a simpler batch query. For each path query $(u, v)$, we compute the LCA of $u$ and $v$ with respect to the RC-Tree root $r$. Then, we solve a batch of path queries of the form PathSum$(r, x)$. The solution to the query $(u, v)$ is then calculated as PathSum$(r, u)$ + PathSum$(r, v)$ − 2PathSum$(r, \text{LCA}(u, v))$.

The algorithm maintains augmented values on each binary cluster corresponding to the sum of the weights on the cluster path. Since this can be computed by summing the weights of the binary children, this is a simple bottom-up computation that uses constant time and space per cluster. No augmented values are stored on the unary clusters. At query time, the algorithm begins with the usual process of marking all of the ancestor clusters of every vertex $u$ and $v$ considered in the query. This marks out a set of $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ nodes in the RC-Tree (Theorem 8). It then performs a top-down computation on those nodes which computes the total weight on the path from $r$ to the representative vertex of the cluster.

This top-down computation can be computed in constant time per cluster by considering the values of the boundary vertices. If the current cluster is a unary cluster, then the total weight from the root is just the total weight to the boundary vertex plus the total weight of the cluster path of the binary child. If the current cluster is a binary cluster, then the total weight to the root is the total weight to the boundary vertex that is closer to the root, plus the weight of the cluster path of the adjacent binary child. Determining which boundary vertex is closer to the root can be done using the same top-down computation as the batch-LCA algorithm. Once each marked cluster has the total weight from the root to its representative, the input queries can be solved.

**Theorem 14** (Batch path queries over a commutative group)**.** A balanced (resp. in expectation) RC-Tree for a forest on $n$ vertices with weights from a commutative group can be augmented to solve a batch of $k$ path sum queries in $O\left(k+k\log\left(1+\frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log n)$ span (resp. w.h.p.).

*Proof.* The augmented values need only examine the values of child clusters, so they take constant time. The computation of the batch-LCAs takes $O\left(k+k\log\left(1+\frac{n}{k}\right)\right)$ work (in expectation if randomized) and $O(\log n)$ span (w.h.p. if randomized) by Theorem 13. Marking all the ancestors of the input vertices also takes $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work and $O(\log n)$ span. The top-down computation looks at the at most two boundary vertices of the cluster and one of the children of the cluster and hence a constant amount of work is required per node. This results in $O\left(k+k\log\left(1+\frac{n}{k}\right)\right)$ preprocessing work (in expectation if randomized), and by parallelising the top-down traversal, a span of $O(\log n)$ (w.h.p. if randomized). After preprocessing, each query $(u,v)$ is answered in constant time by looking up the values on the clusters represented by $u$, $v$, and LCA$(u,v)$, which takes constant time. □

## 3.5.5 Batch Path-Minimum/Maximum Queries

In Section 3.5.4, we discussed the impossibility of a general batch path query algorithm that works for any commutative semigroup (i.e., any desired associative and commutative operation over the weights with no additional assumptions), and gave an algorithm that works for any commutative group, i.e., the case where the weights admit an inverse, such as computing the sum of the weights on a path. Notably, this doesn't cover perhaps the most well studied kinds of path queries, which are path minima and maxima queries. These are often referred to as *bottleneck* queries, and are important because they form the basis of many algorithms for MSTs and MST verification [117, 121, 170], as well as network flow algorithms [171]. In this section we show that the special case of path minima (and maxima by trivial modifications) queries also circumvents the lower bound and is efficiently solvable.

- **BatchPathMin**$(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes an array of tuples representing queries. The output is array where the $i^{\text{th}}$ entry is the lightest edge on the path between $u_i$ and $v_i$.

Unlike our previous batch-query algorithms, the algorithm for path minima will make less use of the RC-Tree and instead defer much of the heavy lifting to existing algorithms. A classic technique for solving bottleneck problems on weighted trees is to first shrink the tree to a smaller tree such that the minimum weight edge on the paths between the query vertices is unaffected. In Chapter 7, we give a parallel algorithm that achieves this. Specifically, given an RC-Tree over a weighted input tree and $k$ query vertices, our algorithm produces a so-called *compressed path tree* containing the query vertices and at most $O(k)$ additional vertices such that the path maxima[2] between every pair of query vertices is preserved. Using this tool, the idea is then to reduce the problem to a small static offline path minima problem and then use the existing algorithm of King et al. [122] to solve it.

## Algorithm: Batch path minima

In Chapter 7, Section 7.2 we show that given an RC-Tree for an edge-weighted tree on $n$ vertices and a set of $k$ marked vertices, we can produce a *compressed path tree* with respect to the $k$ marked vertices in $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work and $O(\log n)$ span. Our algorithm for batch path minimas first produces the compressed path tree with respect to the union of the endpoints of the query vertices $u_i$ and $v_i$. By design of the compressed path tree, the answers to the path queries are the same if they are evaluated on the compressed path tree as if they were evaluated on the original tree. At this point, since the compressed path tree has just $O(k)$ vertices, we can run a static offline algorithm that evaluates the queries and no longer need the RC-Tree. Our algorithm uses the subroutine of King et al.'s parallel MST verification algorithm [122] which evaluates a static offline batch of path minima queries in $O(n + k)$ work and $O(\log n)$ span, where $n$ is the size of the tree and $k$ is the number of query edges. The results of these queries is the solution. Given the algorithm for path minima queries, the algorithm can handle path maxima queries by reversing the result of each comparison, or equivalently, negating the cost of each edge.

**Theorem 15** (Batch path minima/maxima queries)**.** A balanced (resp. in expectation) RC-Tree for a forest on $n$ vertices with comparable edge weights can be augmented to solve a batch of $k$ path minima/maxima queries in $O\left(k + k \log\left(1 + \frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log n)$ span (resp. w.h.p.).

*Proof.* Building the compressed path tree takes $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work (in expectation if randomized) and $O(\log n)$ span (w.h.p. if randomized) by Theorem 31. The static offline algorithm of King et al. [122] runs in linear work and logarithmic span in the size of the tree and number of queries. Since the compressed path tree has $O(k)$ vertices and we evaluate $k$ queries, it takes $O(k)$ work and $O(\log k)$ span. Therefore the total work is $O\left(k + k \log\left(1 + \frac{n}{k}\right)\right)$ (in expectation if randomized) with $O(\log n)$ span (w.h.p. if randomized). □

[2]The algorithm is described for maximums, but can trivially be made to preserve minimums instead

### 3.5.6 Batch Nearest Marked Vertex Queries

In the sequential setting, RC-Trees have been used to solve the *nearest marked vertex* problem in a non-negative edge-weighted tree [5]. Vertices may be marked or unmarked by update operations, and a query must then return the nearest marked vertex to a given vertex. Optionally, edge weight updates can also be supported.

- **BatchNearestMarked**($\{v_1, \ldots, v_k\}$) takes an array of vertices representing queries. The output is an array where the $i^{\text{th}}$ entry is the nearest marked vertex to $v_i$.

The algorithm consists in maintaining augmented values via a bottom-up computation which locate the nearest marked vertices to the representatives inside each cluster, then using a top-down computation to find the global nearest marked vertices by considering those both inside and outside each cluster of interest.

#### Algorithm: Batch nearest marked

Our algorithm maintains several augmented values via a bottom-up computation, similarly to the sequential single-query algorithm [5]. For each cluster, it maintains (1) the nearest marked vertex in the cluster to the representative, (2) the nearest marked vertex in the cluster to each boundary vertex, and (3), if a binary cluster, the length (total weight) of the cluster path. If a cluster contains no marked vertices, it stores a value of **null**/$\infty$ for (1) and (2). Each of these can be maintained in constant time given the values of the children.

(1) If the representative vertex is marked, then it is the nearest marked vertex to itself. Otherwise, since the representative is the boundary vertex shared by the children, the nearest marked vertex to the representative in the cluster is just the nearest out of all the nearest marked vertices to that boundary vertex in the children.

(2) Each boundary vertex is shared with one of the children. The nearest marked vertex to it in the cluster is either the nearest marked vertex to it in that child cluster or it is in one of the other children and hence is the same as the nearest marked vertex to the representative. The nearest of the two can be compared by adding the weight of the cluster path to the distance to the nearest to the representative.

(3) The total weight on the cluster path is just the sum of the weights of the two binary children's cluster paths, or the edge weight if the cluster is a base cluster.

To perform updates such as **BatchMark** and **BatchUnmark**, the algorithm just sets the marks on the corresponding vertices and then propagates the augmented values up the RC-Tree to every ancestor cluster containing those vertices. This takes $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work and $O(\log n)$ span. The time to propagate an edge-weight update would be the same.

The augmented values store what we will call the *locally* nearest marked vertices, i.e., the nearest marked vertices inside the same cluster. At query time, the algorithm performs a top-down computation to compute the *globally* nearest marked vertices, i.e., the actual nearest marked vertices in the entire tree. Specifically, after performing the standard preprocessing step of marking every ancestor cluster of the input vertices, it will perform a top-down computation that will compute for each marked cluster, the (globally) nearest marked vertex to the representative vertex.

For a root cluster, since it contains its entire subtree, the (globally) nearest marked vertex to the representative is the same as the augmented value. Otherwise, given a non-root cluster there are two cases. Either the nearest marked vertex is inside the cluster or it is outside the cluster. The nearest marked vertex inside the cluster is read from the augmented value. If the nearest marked vertex is outside the cluster, then the path from the representative to the nearest marked vertex must pass through one of the boundary vertices, and hence the nearest marked vertex is just the closer of the nearest marked vertex to each of the boundary vertices, which have already been computed earlier in the top-down computation. The algorithm therefore computes the distances from the representative vertex to the nearest marked vertices of the boundaries by adding the weights of the cluster paths that connect them, then takes the closest of the options.

After the top-down computation, the answers can be read off the values of the represented clusters of the query vertices in constant time. We therefore have the following result.

**Theorem 16** (Batch nearest marked vertex queries)**.** A balanced (resp. in expectation) RC-Tree for a forest on $n$ vertices with non-negative edge weights can be augmented to solve a batch of $k$ nearest marked vertex queries in $O\left(k + k\log\left(1+\frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log n)$ span (resp. w.h.p.).

*Proof.* The augmented values need only examine the values of child clusters, so they take constant time. Marking all the ancestors of the input vertices also takes $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work and $O(\log n)$ span (Theorem 8). The top-down computation looks at the augmented values and at the augmented values of the at most two boundary vertices of the cluster and hence a constant amount of work is required per node. This results in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ preprocessing work (in expectation if randomized), and by parallelising the top-down traversal, a span of $O(\log n)$ (w.h.p. if randomized). After preprocessing, each query $v_i$ is answered in constant time by looking up the value on the cluster represented by $v_i$. $\square$

## 3.6 Discussion

RC-Trees are a powerful tool for implementing dynamic algorithms on trees. Their ability to represent both path and subtree decompositions and solve nonlocal queries makes them more generally applicable than classic link/cut trees and Euler-tour trees. Top trees also support these more general queries but appear more difficult to parallelize, and to the best of our knowledge, batch operations on top trees have not been considered. RC-Trees are parallelizable, amenable to batching, and general, making them the perfect framework for implementing parallel batch-dynamic algorithms on trees. To implement parallel batch-dynamic RC-Trees, we need an efficient parallel batch-dynamic tree contraction algorithm. We will give two algorithms in Chapters 4 and 5, one randomized and one deterministic.

We will then see a range of applications of batch-dynamic trees in the subsequent chapters. In Chapter 7 we will use parallel RC-Trees to implement a parallel batch-incremental minimum spanning tree algorithm. In Chapter 8 we will extend parallel RC-Trees to be able to solve batches of mixed queries and weight updates on static trees. This is then used as an ingredient in the first ever work-efficient parallel minimum cut algorithm in Chapter 9. Lastly, in Chapter 10 we will implement our batch-dynamic tree contraction algorithm using parallel self-adjusting computation and experimentally evaluate its performance, demonstrating that parallel RC-Trees could possibly be efficient in practice too.

Since the publication of our work, parallel RC-Trees have also been in used as key ingredients in algorithms by other researchers. Recently, Ghaffari et al. [74] gave the first nearly work-efficient parallel depth-first search algorithm using RC-Trees.

In this chapter, we gave efficient batch query algorithms for all known operations supported by RC-Trees that are amenable to batching. Our algorithms are the first to implement batch queries other than subtree queries on a parallel dynamic tree data structure. They support batch subtree queries in the same work and span bound as batch-parallel Euler-tour trees in addition to supporting batch path queries. To the best of our knowledge, this is the first data structure to support parallel batch path queries. Additionally, our data structure is also the first to consider batch algorithms for *nonlocal* properties. Lastly, although batch subtree queries were applicable to weights with any associative and commutative operation, we showed that path queries are inhibited by a lower bound that makes a similarly general algorithm unattainable. Instead, we showed that batch path queries can be solved efficiently for sums with invertible operations and path minimum/maximum queries, which covers most typical applications.

# Chapter 4
# Randomized Batch-Dynamic Parallel Tree Contraction

## 4.1 Introduction

*Tree contraction* is the process of shrinking a tree down to a single vertex by repeatedly performing local contractions. Each local contraction deletes a vertex of degree at most two and merges its adjacent edges if it had degree two. Tree contraction has a number of useful applications, studied extensively in [5, 133, 134]. It can be used to perform various computations by associating data with edges and vertices and defining how data is accumulated during local contractions.

We are interested in the problem of maintaining a tree contraction dynamically as the underlying tree undergoes batches of links and cuts, i.e., insertions and deletions of edges. Reif and Tate [154] were the first to study batch-dynamic tree contraction and gave an algorithm that can insert or delete a batch of $k$ leaves in $\Theta(k \log n)$ work. The algorithm is not fully general since it only supports modifications at the leaves.

Acar et al. [5] show that dynamic parallel tree contraction is an important tool since it can be used as the basis of an extremely general and powerful dynamic trees framework, RC-Trees, which we discussed extensively in Chapter 3. Our goal is to bring RC-Trees from the sequential, single-update world into the realm of parallel batch-dynamic algorithms. In Chapter 3, we discussed how RC-Trees arise as a byproduct of parallel tree contraction, and therefore, to obtain a parallel batch-dynamic algorithm for maintaining an RC-Tree, we must design an efficient parallel batch-dynamic algorithm for tree contraction.

The existing parallel batch-dynamic Euler-tour trees of Tseng et al. [174] perform batches of $k$ links, cuts, or queries in $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work. In Chapter 3, we showed that RC-Trees can support batches of $k$ queries in $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work, so we naturally wish to aim for a matching bound of $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work for links and cuts in a dynamic parallel tree contraction algorithm. This makes the algorithm of Reif and Tate insufficient for our goals, since it performs $\Theta(k \log n)$ work.

Instead, we will follow the strategy of Acar et al. [4, 5], who designed a *change-propagation* algorithm for static functional computations and used it to automatically dynamize the static tree contraction algorithm of Miller and Reif [133]. In their work, they say that an algorithm is $O(f(n))$-stable under a class of input changes if the number of differences in operations executed by the static algorithm on the old and new input after a change is at most $O(f(n))$. That is, the algorithm would perform roughly $O(f(n))$ different operations on the newly updated input compared to the original input. Then, they give a change propagation algorithm that maintains a *trace* of the algorithm as it runs on the original input, and can then take a change to the input and propagate it through the computation to obtain the updated output, in just $O(f(n))$ time, as long as the algorithm satisfies certain restrictions. They then obtain their result on dynamic trees by proving that Miller and Reif's tree contraction algorithm satisfies these restrictions and is $O(\log n)$-stable under single edge insertions or deletions, hence showing that their change propagation algorithm can efficiently update the contraction in $O(\log n)$ time.

Our goal henceforth is to replicate this result in the parallel setting. We will do so by defining a restricted class of algorithms to which our change propagation framework will apply, which we call *round synchronous* algorithms. A round-synchronous algorithm is essentially one that performs sequential rounds, in each of which a set of parallel processes is allowed to read from shared memory, perform some computation, then write back to shared memory. Parallel tree contraction neatly falls into this class of algorithms and hence can be dynamized by our framework.

## 4.2 Round-Synchronous Algorithms

In this framework, we consider dynamizing algorithms that are *round synchronous*. The round synchronous framework encompasses a range of classic BSP [177] and PRAM algorithms. A round-synchronous algorithm consists of $M$ *processes*, with process IDs bounded by $O(M)$. The algorithm performs sequential rounds in which each active process executes, in parallel, a *round computation*. At the end of a round, any processes can decide to *retire*, in which case they will no longer execute in any future round. The algorithm terminates once there are no remaining active processes—i.e., they have all retired. Given a fixed input, round-synchronous algorithms must perform deterministically. Note that this does not preclude us from implementing randomized algorithms (indeed, our dynamic trees algorithm is randomized), it just requires that we provide the source of randomness as an input to the algorithm, so that its behavior is identical if re-executed. An algorithm in the round synchronous framework is defined in terms of a procedure ComputeRound$(r, p)$, which performs the computation of process $p$ in round $r$. The initial run of a round-synchronous algorithm must specify the set $P$ of initial process IDs.

### Memory model

Processes in a round-synchronous algorithm may read and write to local memory that is not persisted across rounds. They also have access to a *shared memory*. The input to a round-synchronous algorithm is the initial contents of the shared memory. Round computations can read and write to shared memory with the condition that writes do not become visible until the end of the round. Reads can only access shared locations that have been written to, and shared locations can only be written to once, hence concurrent writes are not permitted. The contents of the shared memory at termination is considered to be the algorithm's output. Change propagation is driven by tracking all reads and writes to shared memory.

### Pseudocode

We describe round-synchronous algorithms using the following primitives:

1. The **read** instruction reads the given shared memory locations and returns their values,
2. The **write** instruction writes the given value to the given shared memory location.
3. Processes may retire by invoking the **retire process** instruction.

### Measures

The following measures will help us to analyse the efficiency of round-synchronous algorithms. For convenience, we define the *input configuration* of a round-synchronous algorithm as the pair $(I, P)$, where $I$ is the input to the algorithm (i.e. the initial state of shared memory) and $P$ is the set of initial process IDs.

**Definition 5** (Initial work, Round complexity, and Span)**.** The *initial work* of a round-synchronous algorithm on some input configuration $(I, P)$ is the sum of the work performed by all of the computations of each processes over all rounds when given that input. Its *round complexity* is the number of rounds that it performs, and its *span* is the sum of the maximum costs per round of the computations performed by each process.

## 4.3 Algorithmic Dynamization

Given a round-synchronous algorithm, a *dynamic update* consists of a change to the input configuration, i.e. changing the contents of shared memory, and/or adding or deleting processes. The initial run and change propagation algorithms maintain the following data:

1. $R_{r,p}$, the memory locations read by process $p$ in round $r$
2. $W_{r,p}$, the memory locations written by process $p$ in round $r$
3. $S_m$, the set of round, process pairs that read memory location $m$
4. $X_{r,p}$, which is **true** if process $p$ retired in round $r$

Algorithm 1 depicts the procedure for executing the initial run of a round-synchronous algorithm before making any dynamic updates. To help formalize change propagation, we define the notion of an *affected computation*. The task of change propagation is to identify the affected computations and rerun them.

**Definition 6** (Affected computation)**.** Given a round-synchronous algorithm $A$ and two input configurations $(I, P)$ and $(I', P')$, the *affected computations* are the round and process pairs $(r, p)$ such that either:

1. process $p$ runs in round $r$ on one input configuration but not the other
2. process $p$ runs in round $r$ on both input configurations, but reads a variable from shared memory that has a different value in one configuration than the other

**Algorithm 1** Initial run

```
 1: procedure RUN($P$)
 2:     local $r \leftarrow 0$
 3:     while $P \neq \emptyset$ do
 4:         for each process $p \in P$ do in parallel
 5:             COMPUTEROUND($r, p$)
 6:             $R_{r,p} \leftarrow \{\text{memory locations read by } p \text{ in round } r\}$
 7:             $W_{r,p} \leftarrow \{\text{memory locations written to by } p \text{ in round } r\}$
 8:             $X_{r,p} \leftarrow (\textbf{true if } p \text{ retired in round } r \text{ else } \textbf{false})$
 9:         for each $m \in \cup_{p \in P} R_{r,p}$ do in parallel
10:             $S_m \leftarrow S_m \cup \{(r,p) \mid m \in R_{r,p} \wedge p \in P\}$
11:         $P \leftarrow P \setminus \{p \in P : X_{r,p} = \textbf{true}\}$
12:         $r \leftarrow r + 1$
```

The change propagation algorithm is depicted in Algorithm 2. It works by maintaining the affected computations as three disjoint sets, $P$, the set of processes that read a memory location that was rewritten, $L$, processes that outlived their previous self, i.e. that retired the last time they ran, but did not retire when re-executed, and $D$, processes that retired earlier than their previous self. First, at each round, the algorithm determines the set of computations that should become affected because of shared memory locations that were rewritten in the previous round (Lines 12–14). These are used to determine $P$, the set of affected computations to rerun this round (Line 15). To ensure correctness, the algorithm must then reset the reads that were performed by the computations that are no longer alive, or that will be reran, since the set of locations that they read may differ from last time (Lines 18–19). Lines 22–26 perform the re-execution of all processes that read a changed memory location, or that lived longer (did not retire) than in the previous configuration. The algorithm then subscribes the reads of these computations to the memory locations that they read (Lines 28–29). Finally, on Lines 32–36, the algorithm updates the set of changed memory locations ($U$), the set of computations that lived longer than their previous self ($L$) and the set of computations that retired earlier then their previous self ($D$).

## 4.3.1 Correctness

In this section, we sketch a proof of correctness of the change propagation algorithm (Algorithm 2). Intuitively, correctness is assured because of the write-once condition on global shared memory, which ensures that computations can not have their output overwritten, and hence do not need to be re-executed unless data that they depend on is modified.

**Lemma 5.** Given a dynamic update, re-executing only the affected computations for each round will result in the same output as re-executing all computations on the new input.

*Proof.* Since by definition they read the same values, computations that are not affected,

**Algorithm 2** Change propagation

```
1: // $U$ = sequence of memory locations that have been modified
2: // $P^+$ = sequence of new process IDs to create
3: // $P^-$ = sequence of process IDs to remove
4: procedure PROPAGATE($U, P^+, P^-$)
5:     local $D \leftarrow P^-$                                  // Processes that died earlier than before
6:     local $L \leftarrow P^+$                                  // Processes that lived longer than before
7:     local $A \leftarrow \emptyset$                                   // Affected computations at each round
8:     local $r \leftarrow 0$
9:     while $U \neq \emptyset \vee D \neq \emptyset \vee L \neq \emptyset \vee \exists r' \geq r : (A_{r'} \neq \emptyset)$ do
10:        // Determine the computations that become affected
11:        // due to the newly updated memory locations U
12:        local $A' \leftarrow \cup_{m \in U} S_m$
13:        for each $r' \in \cup_{(r',p) \in A'} \{r'\}$ do in parallel
14:            $A_{r'} \leftarrow A_{r'} \cup \{p \mid (r', p) \in A'\}$
15:        local $P \leftarrow A_r \setminus D$                                  // Processes to rerun
16:        // Forget the prior reads of all processes that are
17:        // now dead or will be rerun on this round
18:        for each $m \in \cup_{p \in P \cup D} R_{r,p}$ do in parallel
19:            $S_m \leftarrow S_m \setminus \{(r, p) \mid m \in R_{r,p} \wedge p \in P \cup D\}$
20:        local $X^{\text{prev}} = \{p \mapsto X_{r,p} \mid p \in P\}$
21:        // (Re)run all changed or newly live processes
22:        for each process $p$ in $P \cup L$ do in parallel
23:            COMPUTEROUND($r, p$)
24:            $R_{r,p} \leftarrow$ {memory locations read by $p$ in round $r$}
25:            $W_{r,p} \leftarrow$ {memory locations written to by $p$ in round $r$}
26:            $X_{r,p} \leftarrow$ (true if $p$ retired in round $r$ else false)
27:        // Remember the reads performed by processes on this round
28:        for each $m \in \cup_{p \in P \cup L} R_{r,p}$ do in parallel
29:            $S_m \leftarrow S_m \cup \{(r, p) \mid m \in R_{r,p} \wedge p \in P \cup L\}$
30:        // Update the sets of changed memory locations,
31:        // newly live processes, and newly dead processes
32:        $U \leftarrow \cup_{p \in (P \cup L)} W_{r,p}$
33:        $L' \leftarrow \{p \in P \mid X_p^{\text{prev}} = \textbf{true} \wedge X_{r,p} = \textbf{false}\}$
34:        $L \leftarrow L \cup L' \setminus \{p \in L \mid X_{r,p} = \textbf{true}\}$
35:        $D' \leftarrow \{p \in P \mid X_p^{\text{prev}} = \textbf{false} \wedge X_{r,p} = \textbf{true}\}$
36:        $D \leftarrow D \cup D' \setminus \{p \in D \mid X_p^{\text{prev}} = \textbf{true}\}$
37:        $r \leftarrow r + 1$
```

if re-executed, would produce the same output as they did the first time. Since all shared memory locations can only be written to once, values written by processes that are not re-executed can not have been overwritten, and hence it is safe to not re-execute them, as their output is preserved. Therefore re-executing only the affected computations will produce the same output as re-executing all computations. □

**Theorem 17** (Consistency)**.** Given a dynamic update, change propagation correctly updates the output of the algorithm.

*Proof.* Follows from Lemma 5 and the fact that all reads and writes to global shared memory are tracked in Algorithm 2, and since global shared memory is the only method by which processes communicate, all affected computations are identified. □

### 4.3.2 Cost Analysis

To analyze the work of change propagation, we need to formalize a notion of *computation distance*. Intuitively, the computation distance between two computations is the work performed by one and not the other. We then show that change propagation can efficiently re-execute the affected computations in work proportional to the computation distance.

**Definition 7** (Computation distance)**.** Given a round-synchronous algorithm $A$ and two input configurations, the *computation distance* $W_\Delta$ between them is the sum of the work performed by all of the affected computations with respect to both input configurations.

**Theorem 18.** Given a round-synchronous algorithm $A$ with input configuration $(I, P)$ that does $W$ work in $R$ rounds and $S$ span, then

1. the initial run of the algorithm with tracking takes $O(W)$ work in expectation and $O(S + R \cdot \log(W))$ span w.h.p.,
2. change propagation on a dynamic update to the input configuration $(I', P')$ takes expected $O(W_\Delta + R')$ work and $O(S' + R' \log(W'))$ span w.h.p., where $S', R', W'$ are the maximum span, rounds, and work of the algorithm on the two input configurations.

*Proof.* We begin by analyzing the initial run. By definition, all executions of the round computations, ComputeRound, take $O(W)$ work and $O(S)$ span in total, with at most an additional $O(\log(M)) = O(\log(W))$ span to perform the parallel for loop. We will show that all additional work can be charged to the round computations, and that at most an additional $O(\log(W))$ span overhead is incurred.

We observe that $R_{r,p}, W_{r,p}$ and $X_{r,p}$ are at most the size of the work performed by the corresponding computations, hence the cost of Lines 6 – 8 can be charged to the computation. The reader sets $S_m$ can be implemented as dynamic arrays with lazy deletion (this will be discussed during change propagation). To append new elements to $S_m$ (Line 10), we can use a semisort performing linear work in expectation to first bucket the shared memory locations in $\cup_{p \in P} R_{r,p}$, whose work can be charged to the corresponding computations that performed the reads. This adds an additional $O(\log(W))$ span w.h.p. since the number of reads is no more than $W$ in total. Finally, removing retired computations from $P$ (Line 11) requires a compaction operation. Since compaction takes linear work, it can be charged to the execution of the corresponding processes. The span of compaction is at most $O(\log(W))$.

Summing up, we showed that all additional work can be charged to the round computations, and the algorithm incurs at most $O(\log(W))$ additional span per round w.h.p. Hence the cost of the initial run is $O(W)$ work in expectation and $O(S + R \cdot \log(W))$ span w.h.p.

We now analyze the change propagation procedure (Algorithm 2). The core of the work is the re-execution of the affected readers on Line 23, which, by definition takes $O(W_\Delta)$ work, and $O(S')$ span, with at most $O(\log(W'))$ additional span to perform the parallel for loop. Since some rounds may have no affected computations, the algorithm could perform up to $O(R')$ additional work to process these rounds. We will show that all additional work can be charged to the affected computations, incurring at most an additional $O(\log(W'))$ span.

Lines 12 – 14 bucket the newly affected computations by round. This can be achieved with an expected linear work semisort and by maintaining the $A_r$ sets as dynamic arrays. The work is chargeable to the affected computations and the span is at most $O(\log(W'))$ w.h.p. Computing the current set of affected computations (Line 15) requires a filter/compaction operation, whose work is charged to the affected computations and span is $O(\log(W'))$.

Updating the reader sets $S_m$ (Line 19) can be done as follows. We maintain $S_m$ as dynamic arrays with lazy deletion, meaning that we delete by marking the corresponding slot as empty. When more than half of the slots have been marked empty, we perform compaction, whose work is charged to the updates and whose span is at most $O(\log(W'))$. In order to perform deletions in constant time, we augment the set $R_{r,p}$ so that it remembers, for each entry $m$, the location of $(r, p)$ in $S_m$. Therefore these updates take constant amortized work each (using a dynamic array), charged to the corresponding affected computations, and at most $O(\log(W'))$ span if a resize/compaction is triggered.

$X^{\text{prev}}$ can be implemented as an array of size $|P|$, with work charged to the affected computations in $P$. As in the initial run, the cost of updating $R_{r,p}$, $W_{r,p}$ and $X_{r,p}$ can also be charged to the work performed by the affected computations.

Updating the reader sets $S_m$ (Line 29) is a matter of appending to dynamic arrays, and, as mentioned earlier, remembering for each $m \in R_{r,p}$, the location of $(r, p)$ in $S_m$. The work can be charged to the affected computations, and the span is at most $O(\log(W'))$.

Collecting the updated locations $U$ (Line 32) can similarly be charged to the affected computations, and incurs no more than $O(\log(W'))$ span. On Lines 33 – 36, the sets $L'$ and $D'$ are computed by a compaction over $P$, whose work is charged to the affected computations in $P$. Updating $L$ and $D$ correspondingly requires a compaction operation, whose work is charged to the affected computations in $L$ and $D$ respectively. Each of these compactions costs $O(\log(W'))$ span.

We can finally conclude that all additional work performed by change propagation can be charged to the affected computations, and hence to the computation distance $W_\Delta$, while incurring at most $O(\log(W'))$ additional span per round w.h.p. Therefore the total work of change propagation is $O(W_\Delta + R')$ in expectation and the span is $O(S' + R' \cdot \log(W'))$ w.h.p. □

We now show that for a special class of round-synchronous algorithms, the span overhead can be reduced. Our dynamic trees algorithm falls into this special case.

**Definition 8.** A *restricted round-synchronous* algorithm is a round-synchronous algorithm such that each round computation performs only a constant number of reads and writes, and each shared memory location is read only by a constant number of computations, and only in the round directly after it was written.

**Theorem 19.** Given a restricted round-synchronous algorithm $A$ with input configuration $(I, P)$ that does $W$ work in $R$ rounds and $S$ span, then

1. the initial run of the algorithm takes $O(W)$ work and $O(S + R\log^*(W))$ span w.h.p.
2. change propagation on a dynamic update to the input configuration $(I', P')$ takes expected $O(W_\Delta)$ work, and $O(S' + R'\log^*(W'))$ span w.h.p., where $S', R', W'$ are the maximum span, rounds, and work of the algorithm on the two input configurations.

*Proof sketch.* Rather than recreate the entirety of the proof of Theorem 18, we simply sketch the differences. In essence, we obtain the result by removing the uses of scans, and semisorts, which were the main cause of the $O(\log(W'))$ span overhead and the randomized work. Instead, we rely only on (possibly approximate) compaction, which requires randomization to achieve the lowest possible span. We also lose the $R'$ term in the work since computations can only read from locations written in the previous round, and hence the set of rounds on which there exists an affected computation must be contiguous.

The main technique that we will make use of is the sparse array plus compaction technique. In situations where we wish to collect a set of items from each executed process, we would, in the unrestricted model, require a scan, which costs $O(\log(W'))$ span. If each executed process, however, only produces a constant number of these items, we can allocate an array that is a constant size larger than the number of processes, and each process can write its set of items to a designated offset. We can then perform (possibly approximate) compaction on this array to obtain the desired set, with at most a constant factor additional blank entries. This takes $O(\log^*(W'))$ span w.h.p.

Maintaining $S_m$ in the initial run and during change propagation is the first bottleneck, originally requiring a semisort. Since each computation performs a constant number of writes, we can collect the writes using the sparse array plus compaction technique. Since, in the restricted model, each modifiable will only be read by a constant number of readers, we can update $S_m$ in constant time.

To compute the affected computations $A_r$ also required a semisort, but in the restricted model, since all reads happen on the round directly after the write, no semisort is needed, since they will all have the same value of $r$. Collecting the affected computations from the written modifiables can also be achieved using the sparse array and compaction technique, using the fact that each computation wrote to a constant number of modifiables, and each modifiable is subsequently read by a constant number of computations. Additionally, $A_r$ will be empty at the beginning of round $r$, so computing $P$ requires only a compaction.

Lastly, collecting the updated locations $U$ can also be performed using the sparse array and compaction technique. In summary, we can replace all originally $O(\log(W'))$ span operations with (approximate) compaction in the restricted setting, and hence we obtain the given span bounds since this takes $O(\log^*(W'))$ span w.h.p. $\square$

**Remark 1** (Space usage)**.** We do not formally specify an implementation of the memory model, but one simple way to achieve good space bounds is to use hashtables to implement global shared memory. Each write to a particular global shared memory location maps to an entry in the hashtable. When a round computation is invalidated during a dynamic update, its writes can be purged from the hashtable to free up space, preventing unbounded space blow up. Since the algorithm must also track the reads of each global shared memory location, using this implementation, the space usage is proportional to the number of shared memory reads and writes. In the restricted round-synchronous model, the number of reads must be proportional to the number of writes, and hence the space usage is proportional to the number of writes.

## 4.4 Dynamizing Tree Contraction

In this section, we show how to obtain a dynamic tree contraction algorithm by applying our dynamization technique to the static tree contraction algorithm of Miller and Reif [132].

### Input forests

The algorithms described here operate on undirected forests $F = (V, E)$, where $V$ is a set of vertices, and $E$ is a set of undirected edges. If $(u, v) \in E$, we say that $u$ and $v$ are *adjacent*, or that they are *neighbors*. A vertex with no neighbors is said to be *isolated*, and a vertex with one neighbour is called a *leaf*.

We assume that the forests given as input have bounded degree. That is, there exists some constant $t$ such that each vertex has at most $t$ neighbors. Arbitrary-degree trees should be handled by ternarization, i.e., by converting high-degree vertices into paths of vertices with degree at most three, as described in Chapter 3.

### The static algorithm

The static tree contraction algorithm (Algorithm 4) works in rounds, each of which takes a forest from the previous round as input and produces a new forest for the next round. On each round, some vertices may be *deleted*, in which case they are removed from the forest and are not present in all remaining rounds. Let $F^i = (V^i, E^i)$ be the forest after $i$ rounds of contraction, and thus $F^0 = F$ is the input forest. We say that a vertex $v$ is *alive* at round $i$ if $v \in V^i$, and is *dead* at round $i$ if $v \notin V^i$. If $v \in V^i$ but $v \notin V^{i+1}$ then $v$ was deleted in round $i$. There are three ways for a vertex to be deleted: it either *finalizes* (Line 32), *rakes* (Line 21), or *compresses* (Line 26). Finalization removes isolated vertices. Rake removes all leaves from the tree, with one special exception. If two leaves are adjacent, then to break symmetry and ensure that only one of them rakes, the one with the lower identifier rakes into the other (Line 8). Finally, compression removes an independent set of degree two vertices that are not adjacent to any degree one vertices, as in Miller and Reif's algorithm. The choice of which vertices are deleted in each round is made locally for each vertex based upon its own degree, the degrees of its neighbors, and coin flips for itself and its neighbors (Line 13). For coin flips, we assume a function $\textsc{Heads}(i, v)$ which indicates whether or not vertex $v$ flipped heaps on round $i$. It is important that $\textsc{Heads}(i, v)$ is a function of both the vertex and the round number, as coin flips must be repeatable for change propagation to be correct.

The algorithm produces a *contraction data structure* which serves as a record of the contraction process. The contraction data structure is a tuple, $(A, D)$, where $A[i][u]$ is a list of pairs containing the vertices adjacent to $u$ in round $i$, and the positions of $u$ in the adjacency lists of the adjacent vertices. $D[u]$ stores the round on which vertex $u$ contracted. The algorithm also records leaf$[i][u]$, which is true if vertex $u$ is a leaf at round $i$. An implementation of the tree contraction algorithm in our framework is shown in Algorithm 4.

### Updates

We consider update operations that implement the interface of a batch-dynamic tree data structure. This requires supporting batches of links and cuts. A *link (insertion)* connects two trees in the forest by a newly inserted edge. A *cut (deletion)* deletes an edge from the forest, separating a single tree into two trees. Recall the interface that we wish to support:

- $\textbf{BatchLink}(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes a batch of edges and adds them to $F$. The edges must not create a cycle.
- $\textbf{BatchCut}(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes a batch of edges and removes them from the forest.

An implementation of the high-level interface for updates in terms of the contraction data structure is depicted in Algorithm 3.

## 4.5 Stability Analysis

We now analyse the initial work, round, complexity, span, and computation distance of the tree contraction algorithm. This section is dedicated to proving the following theorem.

**Theorem 20.** Given a forest of $n$ vertices, the initial work of tree contraction is $O(n)$ in expectation, the round complexity and the span is $O(\log(n))$ w.h.p. and the computation distance induced by updating $k$ edges is $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ in expectation.

Let $F = (V, E)$ be the set of initial vertices and edges of the input tree, and denote by $F^i = (V^i, E^i)$, the set of remaining (alive) vertices and edges at round $i$. We use the term *at round i* to denote the beginning of round $i$, and *in round i* to denote an event that occurs during round $i$. For some vertex $v$ at round $i$, we denote the set of its adjacent vertices by $A^i(v)$, and its degree with $\delta^i(v) = \left|A^i(v)\right|$. A vertex is *isolated* at round $i$ if $\delta^i(v) = 0$. When multiple

**Algorithm 3** Link and cut operations for round-synchronous tree contraction

```
procedure BUILD($V, E$)
    for each vertex $v \in V$ do in parallel
        write($A[0][v]$, $\{u : (u, v) \in E\}$)
        write(leaf[0][$v$], ($|A[0][v]| = 1$))
    RUN($|V|$)

procedure BATCHLINK($E^+ = \{(u_1, v_1), \ldots (u_k, v_k)\}$)
    local $U \leftarrow \cup_{(u,v)\in E^+} \{u, v\}$
    for each vertex $u \in U$ do in parallel
        write($A[0][u]$, $A[0][u] \cup \{v : (u, v) \in E^+\}$)
        write(leaf[0][$u$], ($|A[0][u]| = 1$))
    local $M = \cup_{u\in U} A[0][u] \cup \{$leaf[0][$u$] such that leaf[0][$u$] changed$\}$
    PROPAGATE($M, \emptyset, \emptyset$)

procedure BATCHCUT($E^- = \{(u_1, v_1), \ldots (u_k, v_k)\}$)
    local $U \leftarrow \cup_{(u,v)\in E^-} \{u, v\}$
    for each vertex $u \in U$ do in parallel
        write($A[0][v]$, $A[0][v] \setminus \{u : (u, v) \in E^-\}$)
        write(leaf[0][$u$], ($|A[0][u]| = 1$))
    local $M = \cup_{u\in U} A[0][u] \cup \{$leaf[0][$u$] such that leaf[0][$u$] changed$\}$
    PROPAGATE($M, \emptyset, \emptyset$)
```

forests are in play, it will be necessary to disambiguate which is in focus. For this, we will use subscripts: for example, $\delta^i_F(v)$ is the degree of $v$ in the forest $F^i$, and $E^i_F$ is the set of edges in the forest $F^i$.

## 4.5.1 Analysis of Construction

We first show that the static tree contraction algorithm is efficient. This argument is similar to Miller and Reif's argument in Theorem 2.1 of [133].

**Lemma 6.** For any forest $(V, E)$, there exists $\beta \in (0, 1)$ such that $\mathbf{E}\left[\left|V^i\right|\right] \le \beta^i |V|$, where $V^i$ is the set of vertices remaining after $i$ rounds of contraction.

*Proof.* We begin by considering trees, and then extend the argument to forests. Given a tree $(V, E)$, consider the set $V'$ of vertices after one round of contraction. We would like to show there exists $\beta \in (0, 1)$ such that $\mathbf{E}[|V'|] \le \beta |V|$. If $|V| = 1$, then this is trivial since the vertex finalizes (it is deleted with probability 1). For $|V| \ge 2$, Consider the following sets, which partition the vertex set:

1. $H = \{v : \delta(v) \ge 3\}$
2. $L = \{v : \delta(v) = 1\}$

**Algorithm 4** Tree contraction algorithm

```
procedure ComputeRound($(i, u)$)
  local $((v_1, p_1), \ldots, (v_t, p_t)), \ell \leftarrow$ read($A[i][u]$, leaf$[i][u]$)
  if $v_i = \perp \forall i$ then                                  // A vertex with no neighbors finalizes
    DoFinalize($(i, u)$)
  else if $\ell$ then                                              // A leaf vertex rakes if its neighbor is
    local $(v, p) \leftarrow (v_i, p_i)$ such that $v_i \neq \perp$   //  not a leaf, or if it has the lower ID
    local $\ell' \leftarrow$ read(leaf$[i][v]$)
    if $\neg\ell' \vee u < v$ then DoRake($(i, u, (v, p))$)
    else DoAlive($(i, u, ((v_1, p_1), \ldots, (v_t, p_t)))$)
  else
    if $\exists (v, p), (v', p') : \{v_1, \ldots, v_t\} \setminus \{\perp\} = \{v, v'\}$ then   // If the vertex has exactly two neighbors,
      local $\ell', \ell'' \leftarrow$ read(leaf$[i][v]$, leaf$[i][v']$)     //  it will compress if neither of them are
      local $c \leftarrow$ Heads($(i, u)$) $\wedge \neg$Heads($(i, v)$) $\wedge \neg$Heads($(i, v')$)   // leaves and it flips heads and they both
      if $(\neg\ell' \wedge \neg\ell'' \wedge c)$ then                  //  flip tails
        DoCompress($(i, u, (v, p), (v', p'))$)
      else
        DoAlive($(i, u, ((v_1, p_1), \ldots, (v_t, p_t)))$)
    else
      DoAlive($(i, u, ((v_1, p_1), \ldots, (v_t, p_t)))$)

procedure DoRake($(i, u, (v, p))$)                     // When a vertex rakes, it replaces itself with
  write($A[i+1][v][p]$, $\perp$)                       //  null ($\perp$) in its neighbor's adjacency list in
  write($D[u]$, $i$)                                    // in the next round
  retire process

procedure DoCompress($(i, u, (v, p), (v', p'))$)       // When a vertex compresses, it replaces itself
  write($A[i+1][v][p]$, $(v', p')$)                    // with its opposite neighbors in each neighbor's
  write($A[i+1][v'][p']$, $(v, p)$)                    // adjacency list in the next round
  write($D[u]$, $i$)
  retire process

procedure DoFinalize($(i, u)$)
  write($D[u]$, $i$)
  retire process

procedure DoAlive($(i, u, ((v_1, p_1), \ldots, (v_t, p_t)))$)   // If a vertex remains alive, it writes itself into
  local nonleaves $\leftarrow 0$                                 // its neighbors' adjacency lists in the next
  for $j \leftarrow 1$ to $t$ do                                 // round. It must also determine whether it
    if $v_j \neq \perp$ then                                     // it will be a leaf in the next round
      write($A[i+1][v_j][p_j]$, $(u, j)$)
      nonleaves += 1 - read(leaf$[i][v_j]$)
    else
      write($A[i+1][u][j]$, $\perp$)
  write(leaf$[i+1][u]$, nonleaves $= 1$)
```

3. $C = \{v : \delta(v) = 2 \wedge \forall u \in A(v), u \notin L\}$
4. $C' = \{v : \delta(v) = 2\} \setminus C$

At least half of the vertices in $L$ must be deleted, since all leaves are deleted, except those that are adjacent to another leaf, in which case exactly one of the two is deleted. In expectation, 1/8 of the vertices in $C$ are deleted. Vertices in $H$ and $C'$ necessarily do not get deleted. Now, observe that $|C'| \le |L|$, since each vertex in $C'$ is adjacent to a distinct leaf. Finally, we also have $|H| < |L|$, which follows from standard arguments about compact trees. Therefore in expectation,

$$\frac{1}{2}|L| + \frac{1}{8}|C| \ge \frac{1}{4}|L| + \frac{1}{8}|H| + \frac{1}{8}|C'| + \frac{1}{8}|C| \ge \frac{1}{8}|V|$$

vertices are deleted, and hence

$$\mathbf{E}\left[\left|V'\right|\right] \le \frac{7}{8}|V|.$$

Equivalently, for $\beta = \frac{7}{8}$, for every $i$, we have $\mathbf{E}\left[\left|V^{i+1}\right| \;\middle|\; V_i\right] \le \beta \left|V^i\right|$, where $V^i$ is the set of vertices after $i$ rounds of contraction. Therefore $\mathbf{E}\left[\left|V^{i+1}\right|\right] \le \beta\, \mathbf{E}\left[\left|V^i\right|\right]$. Expanding this recurrence, we have $\mathbf{E}\left[\left|V^i\right|\right] \le \beta^i |V|$. To extend the proof to forests, simply partition the forest into its constituent trees and apply the same argument to each tree individually. Due to linearity of expectation, summing over all trees yields the desired bounds. □

**Lemma 7.** On a forest of $n$ vertices, after $O(\log n)$ rounds of contraction, there are no vertices remaining w.h.p.

*Proof.* For any $c > 0$, consider round $r = (c+1) \cdot \log_{1/\beta}(n)$. By Lemma 6 and Markov's inequality, we have

$$\mathbf{P}[|V^r| \ge 1] \le \beta^r n = n^{-c}.$$

□

*Proof of initial work, rounds, and span in Theorem 20.* At each round, the construction algorithm performs $O\left(\left|V^i\right|\right)$ work, and so the total work is $O\left(\sum_i \mathbf{E}\left[\left|V^i\right|\right]\right)$ in expectation. By Lemma 6, this is $O(|V|) = O(n)$. The round complexity and the span follow from Lemma 7. □

### 4.5.2 Analysis of Dynamic Updates

Intuitively, tree contraction is efficiently dynamizable due to the observation that, when a vertex locally makes a choice about whether or not to delete, it only needs to know who its neighbors are, and whether or not its neighbors are leaves. This motivates the definition of the *configuration* of a vertex $v$ at round $i$, denoted $\kappa^i_F(v)$, defined as

$$\kappa^i_F(v) = \begin{cases} (\{(u, \ell^i_F(u)) : u \in A^i_F(v)\}), & \text{if } v \in V^i_F \\ \text{dead}, & \text{if } v \notin V^i_F, \end{cases}$$

where $\ell_F^i(u)$ indicates whether $\delta_F^i(u) = 1$ (the *leaf status* of $u$). Consider some input forest $F = (V, E)$, and let $F' = (V, (E \setminus E^-) \cup E^+)$ be the newly desired input after a batch cut with edges $E^-$ and/or a batch-link with edges $E^+$. We say that a vertex $v$ is *affected* at round $i$ if $\kappa_F^i(v) \neq \kappa_{F'}^i(v)$.

**Lemma 8.** The execution in the tree contraction algorithm of process $p$ at round $r$ is an affected computation if and only if $p$ is an affected vertex at round $r$.

*Proof.* The code for ComputeRound for tree contraction reads only the neighbours, and corresponding leaf statuses, which are precisely the values encoded by the configuration. Hence if vertex $p$ is alive in both forests the computation $p$ is affected if and only if vertex $p$ is affected. If instead $p$ is dead in one forest but not the other, vertex $p$ is affected, and the process $p$ will have retired in one computation but not the other, and hence it will be an affected computation. Otherwise, if vertex $p$ is dead in both forests, then the process $p$ will have retired in both computations, and hence be unaffected. $\square$

This means that we can bound the computation distance by bounding the number of affected vertices. First, we show that vertices that are not affected at round $i$ have nice properties.

**Lemma 9.** If $v$ is unaffected at round $i$, then either $v$ is dead at round $i$ in both $F$ and $F'$, or $v$ is adjacent to the same set of vertices in both.

*Proof.* Follows directly from $\kappa_F^i(v) = \kappa_{F'}^i(v)$. $\square$

**Lemma 10.** If $v$ is unaffected at round $i$, then $v$ is deleted in round $i$ of $F$ if and only if $v$ is also deleted in round $i$ of $F'$, and in the same manner (finalize, rake, or compress).

*Proof.* Suppose that $v$ is unaffected at round $i$. Then by definition it has the same neighbours at round $i$ in both $F$ and $F'$. The contraction process depends only on the neighbours of the vertex, and hence proceeds identically in both cases. $\square$

If a vertex $v$ is not affected at round $i$ but is affected at round $i+1$, then we say that $v$ *becomes affected in round* $i$. A vertex can become affected in many ways.

**Lemma 11.** If $v$ becomes affected in round $i$, then at least one of the following holds:
1. $v$ has an affected neighbor $u$ at round $i$ which was deleted in either $F^i$ or $(F')^i$.
2. $v$ has an affected neighbour $u$ at round $i+1$ where $\ell_F^{i+1}(u) \neq \ell_{F'}^{i+1}(u)$.

*Proof.* First, note that since $v$ becomes affected, we know $v$ does not get deleted, and furthermore that $v$ has at least one neighbor at round $i$. If $v$ were to be deleted, then by Lemma 10 it would do so in both forests, leading it to being dead in both forests at the next round and therefore unaffected. If $v$ were to have no neighbors, then $v$ would finalize, but we just argued that $v$ cannot be deleted.

Suppose that the only neighbors of $v$ which are deleted in round $i$ are unaffected at round $i$. Then $v$'s set of neighbors in round $i+1$ is the same in both forests. If all of these are unaffected at round $i+1$, then their leaf statuses are also the same in both forests at round $i+1$, and hence $v$ is unaffected, which is a contradiction. Thus case 2 of the lemma must hold. In any other scenario, case 1 of the lemma holds. □

**Lemma 12.** If $v$ is not deleted in either forest in round $i$ and $\ell_F^{i+1}(v) \neq \ell_{F'}^{i+1}(v)$, then $v$ is affected at round $i$.

*Proof.* Suppose $v$ is not affected at round $i$. If none of $v$'s neighbors are deleted in this round in either forest, then $\ell_F^{i+1}(v) = \ell_{F'}^{i+1}(v)$, a contradiction. Otherwise, if the only neighbors that are deleted do so via a compression, since compression preserves the degree of its endpoints, we will also have $\ell_F^{i+1}(v) = \ell_{F'}^{i+1}(v)$ and thus a contradiction. So, we consider the case of one of $v$'s neighbors raking. However, since $v$ is unaffected, we know $\ell_F^i(u) = \ell_{F'}^i(u)$ for each neighbor $u$ of $v$. Thus if one of them rakes in round $i$ in one forest, it will also do so in the other, and we will have $\ell_F^{i+1}(v) = \ell_{F'}^{i+1}(v)$. Therefore $v$ must be affected at round $i$. □

Lemmas 11 and 12 give us tools to bound the number of affected vertices for a consecutive round of contraction: each affected vertex that is deleted affects its neighbors, and each affected vertex whose leaf status is different in the two forests at the next round affects its neighbor. This strategy actually overestimates which vertices are affected, since case 1 of Lemma 11 does not necessarily imply that $v$ is affected at the next round. We wish to show that the number of affected vertices at each round is not large. Intuitively, we will show that the number of affected vertices grows only arithmetically in each round, while shrinking geometrically, which implies that their total number can never grow too large. Let $A^i$ denote the set of affected vertices at round $i$. We begin by bounding the size of $|A^0|$.

**Lemma 13.** For a batch update of size $k$, we have $|A^0| \leq 4k$.

*Proof.* The computation for a given vertex $u$ at most reads its neighbors, and if it has a single neighbor, its neighbor's leaf status. Therefore, the addition/deletion of a single edge affects its endpoints and at most one other vertex per endpoint, for at most 4 vertices at round 0. Hence $|A^0| \leq 4k$. □

We say that an affected vertex $u$ *spreads to* $v$ in round $i$, if $v$ was unaffected at round $i$ and $v$ becomes affected in round $i$ in either of the following ways:

1. $v$ is a neighbor of $u$ at round $i$ and $u$ is deleted in round $i$ in either $F$ or $F'$, or
2. $v$ is a neighbor of $u$ at round $i+1$ and the leaf status of $u$ changes in round $i$, i.e., $\ell_F^{i+1}(v) \neq \ell_{F'}^{i+1}(v)$.

Let $s = |A^0|$. For each of $F$ and $F'$, we now inductively construct $s$ disjoint sets for each round $i$, labeled $A_1^i, A_2^i, \ldots A_s^i$. These sets will form a partition of $A^i$. First, arbitrarily partition $A^0$ into $s$ singleton sets, and let $A_1^0, \ldots, A_s^0$ be these singleton sets. In other words, each affected vertex in $A^0$ is assigned a unique number $1 \leq j \leq s$, and is then placed in $A_j^0$.

Given sets $A_1^i, \ldots, A_s^i$, we construct sets $A_1^{i+1}, \ldots, A_s^{i+1}$ as follows. Consider some $v \in A^{i+1} \backslash A^i$. By Lemmas 11 and 12, there must exist at least one $u \in A^i$ such that $u$ spreads to $v$. Since there could be many of these, let $S^i(v)$ be the set of vertices which spread to $v$ in round $i$. Define:

$$j^i(v) = \begin{cases} j, & \text{if } v \in A_j^i \\ \min_{u \in S^i(v)} \left( j \text{ where } u \in A_j^i \right), & \text{otherwise} \end{cases}$$

In other words, $j^i(v)$ is $v$'s set identifier if $v$ is affected at round $i$, or otherwise the minimum set identifier $j$ such that a vertex from $A_j^i$ spread to $v$ in round $i$. We can then produce the following for each $1 \le j \le k$, the *affected components*:

$$A_j^{i+1} = \{ v \in A^{i+1} \mid j^i(v) = j \}$$

Informally, each affected vertex from round $i$ which stays affected also stays in the same place, and each newly affected vertex picks a set to join based on which vertices spread to it.

We say that a vertex $v$ is a *frontier* at round $i$ if $v$ is affected at round $i$ and at least one of its neighbors in either $F$ or $F'$ is unaffected at round $i$. It is easy to show that any frontier at any round is alive in both forests and has the same set of unaffected neighbors in both at that round, and thus, the set of frontier vertices at any round is the same in both forests. It is also easy to show that if a vertex $v$ spreads to some other vertex in round $i$, then $v$ is a frontier at round $i$. We describe next the structure of the affected components and show that the number of frontier vertices within each is bounded.

**Lemma 14.** For any $i, j$, the subforests induced by $A_j^i$ in each of $F^i$ and $(F')^i$ are trees.

*Proof.* This follows from rake and compress preserving connectedness, and the fact that if $u$ spreads to $v$ then $u$ and $v$ are neighbors in both forests either at round $i$ or round $i+1$. □

**Lemma 15.** For any $i, j$, each of the following statements hold:
1. $A_j^i$ contains at most 2 frontier vertices.
2. $|A_j^{i+1} \setminus A_j^i| \le 2$.

*Proof.* We prove statement 1 by induction on $i$, and conclude statement 2 in the process. At round 0, each $A_j^0$ contains at most 1 frontier. We now consider some $A_j^i$. Suppose there is a single frontier vertex $v$ in $A_j^i$. If $v$ compresses in one of the forests, then $v$ will not be a frontier in $A_j^{i+1}$, but it will spread to at most two newly affected vertices which may be frontiers at round $i+1$. Thus the number of frontiers in $A_j^{i+1}$ is at most 2, and $|A_j^{i+1} \setminus A_j^i| \le 2$.

If $v$ rakes in one of the forests, then $v$ must also rake in the other forest (if not, then $v$ could not be a frontier, since its neighbor would be affected). It spreads to one newly affected vertex (its neighbor) which may be a frontier at round $i+1$. Thus the number of frontiers in $A_j^{i+1}$ will be at most 1, and $|A_j^{i+1} \setminus A_j^i| \le 1$.

Now suppose there are two frontiers $u$ and $v$ in $A_j^i$. Due to statement 1 of the Lemma, each of these must have at least one affected neighbor at round $i$. Thus if either is deleted, it will cease to be a frontier and may add at most one newly affected vertex to $A_j^{i+1}$, and this newly affected vertex might be a frontier at round $i+1$. The same can be said if either $u$ or $v$ spreads to a neighbor due to a leaf status change. Thus the number of frontiers either remains the same or decreases, and there are at most 2 newly affected vertices. Hence statements 1 and 2 of the Lemma hold. □

Now define $A_{F,j}^i = A_j^i \cap V_F^i$, that is, the set of vertices from $A_j^i$ which are alive in $F$ at round $i$. We define $A_{F',j}^i$ similarly for forest $F'$.

**Lemma 16.** For every $i, j$, we have

$$\mathbf{E}\left[\left|A_{F,j}^i\right|\right] \leq \frac{6}{1-\beta},$$

and similarly for $A_{F',j}^i$.

*Proof.* Let $F_{A,j}^i$ denote the subforest induced by $A_{F,j}^i$ in $F^i$. By Lemma 15, this subforest is a tree, and has at most 2 frontier vertices. By Lemma 6, if we applied one round of contraction to $F_{A,j}^i$, the expected number of vertices remaining would be at most $\beta \cdot \mathbf{E}[|A_{F,j}^i|]$. However, some of the vertices that are deleted in $F_{A,j}^i$ may not be deleted in $F^i$. Specifically, any vertex in $A_{F,j}^i$ which is a frontier or is the neighbor that spread to a frontier might not be deleted. There are at most two frontier vertices and two associated neighbors. By Lemma 15, two newly affected vertices might also be added. We also have $|A_{F,j}^0| = 1$. Therefore we conclude the following, which similarly holds for forest $F'$:

$$\mathbf{E}\left[\left|A_{F,j}^{i+1}\right|\right] \leq \beta\,\mathbf{E}\left[\left|A_{F,j}^i\right|\right] + 6 \leq 6\sum_{r=0}^{\infty}\beta^r = \frac{6}{1-\beta}.$$

□

**Lemma 17.** For a batch update of size $k$, we have for every $i$,

$$\mathbf{E}\left[\left|A^i\right|\right] \leq \frac{48}{1-\beta}k.$$

*Proof.* Follows from Lemmas 13 and 16, and the fact that

$$\left|A^i\right| \leq \sum_{j=1}^{s}\left(\left|A_{F,j}^i\right| + \left|A_{F',j}^i\right|\right).$$

□

*Proof of computation distance in Theorem 20.* Let $F$ be the given forest and $F'$ be the desired forest. Since each process of tree contraction does constant work each round, Lemma 8 implies that the algorithm does $O\left(\left|A^i\right|\right)$ work at each round $i$, so $W_\Delta = \sum_i \left|A^i\right|$.

Since at least one vertex is either raked or finalized each round, we know that there are at most $n$ rounds. Consider round $r = \log_{1/\beta}\left(1+\frac{n}{k}\right)$, using the $\beta$ given in Lemma 6. We now split the rounds into two groups: those that come before $r$ and those that come after.

For $i < r$, we bound $\mathbf{E}\left[\left|A^i\right|\right]$ according to Lemma 17, yielding

$$\sum_{i<r} \mathbf{E}\left[\left|A^i\right|\right] = O(rk) = O\left(k \log\left(1+\frac{n}{k}\right)\right)$$

work. Now consider $r \le i < n$. For any $i$ we know $\left|A^i\right| \le \left|V_F^i\right| + \left|V_{F'}^i\right|$, because each affected vertex must be alive in at least one of the two forests at that round. We can then apply the bound given in Lemma 6, and so

$$\begin{aligned}
\sum_{r\le i<n} \mathbf{E}\left[\left|A^i\right|\right] &\le \sum_{r\le i<n} \left(\mathbf{E}\left[\left|V_F^i\right|\right] + \mathbf{E}\left[\left|V_{F'}^i\right|\right]\right) \\
&\le \sum_{r\le i<n} \left(\beta^i n + \beta^i n\right) \\
&= O(n\beta^r) \\
&= O\left(\frac{nk}{n+k}\right) \\
&= O(k),
\end{aligned}$$

and thus

$$\mathbf{E}[W_\Delta] = O\left(k \log\left(1+\frac{n}{k}\right)\right) + O(k) = O\left(k \log\left(1+\frac{n}{k}\right)\right).$$

□

## 4.6 Discussion

In Chapter 3, we discussed the RC-Tree framework, which produces a dynamic tree data structure on top of any algorithm for dynamic tree contraction. In this chapter, we have derived an algorithm for parallel batch-dynamic tree contraction that can handle batches of $k$ links or cuts in $O\left(k \log\left(1+\frac{n}{k}\right)\right)$ work in expectation and $O(\log n \log^* n)$ span w.h.p. This improves on the results of Reif and Tate [154] who gave an algorithm that runs in $O(k \log n)$ work and could only perform modifications to the leaves of the tree. This new result directly implies the existence of randomized parallel batch-dynamic RC-Trees with the same bounds for batch links and cuts. We showed in Lemma 6 that the tree contraction algorithm shrinks the forest geometrically, and hence we obtain a balanced RC-Tree. Formally, we have shown the following slightly weaker version of Theorem 1:

> *There is a randomized parallel batch-dynamic algorithm that maintains a balanced RC-Tree of a bounded-degree forest subject to batches of $k$ edge updates (insertions, deletions, or both) in $O\left(k \log\left(1+\frac{n}{k}\right)\right)$ work in expectation and $O(\log \log^* n n)$ span w.h.p.*

Theorem 1 asserts the existence of a balanced RC-Tree with the same bounds except $O(\log n)$ span. We will discuss how to optimize and improve the span from $O(\log n \log^* n)$ to just $O(\log n)$ in Chapter 5. Our algorithm, like all existing parallel dynamic tree algorithms is randomized. In Chapter 5, we will also design the first work-efficient deterministic algorithm for batch-dynamic tree contraction, which results in a deterministic algorithm for RC-Trees.

Combined with the results of Chapter 3, in this chapter we have provided the first proof of the existence of balanced parallel batch-dynamic RC-Trees. These are the second parallel batch-dynamic tree algorithm, following the results of Tseng et al. [174] on parallel batch-dynamic Euler-tour trees. Compared to Euler-tour trees, RC-Trees permit a much wider range of queries, being capable of solving both path and subtree queries as well as nonlocal queries such as centers, diameters, and medians. On the other hand, one downside of RC-Trees is that they are restricted to bounded-degree trees, while Euler-tour trees are not.

# Chapter 5
# Deterministic Batch-Dynamic Parallel Tree Contraction

## 5.1 Introduction

Work-efficient parallel batch-dynamic graph algorithms are a relatively recent idea, many of the first appearing in this thesis such as our earlier results on parallel batch-dynamic tree contraction and RC-Trees (Chapters 3 and 4), and work appearing in later chapters on connectivity (Chapter 6) and incremental MSTs (Chapter 7). Other results have recently been discovered by other researchers including for $k$-clique counting [52], $k$-core decomposition [126], as well as results that build on our own published work, such as fully dynamic MSTs [175], which extend our work on connectivity (Chapter 6). These are all parallel batch-dynamic algorithms for graphs, but they have something else in common: every one of them uses randomization. Randomization is a very powerful tool for parallel graph problems and more broadly because it allows for easy symmetry-breaking when faced with local decisions.

Perhaps for some problems, randomization is employed only as a convenience, but for others it appears necessary to obtain the most efficient algorithms. 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 [144], but no deterministic equivalent has been discovered. The best deterministic algorithm requires an additional $\alpha(n, m)$ factor of work [41].

In this chapter, we break the randomization barrier for work-efficient parallel batch-dynamic graph algorithms by designing the first efficient deterministic algorithm for parallel batch-dynamic tree contraction. We design a work-efficient algorithm for the batch-dynamic trees problem that is deterministic and runs in polylog $n$ span. As a byproduct, we obtain a deterministic algorithm for maintaining an RC-Tree under batch links and cuts. To the best of our knowledge, this is the first deterministic work-efficient parallel batch-dynamic algorithm for any graph problem.

Unlike the randomized algorithm, our deterministic algorithm is not based on change-propagation. Rather, we describe an update algorithm that takes batches of links and cuts and updates the tree contraction directly. We start by developing a simpler algorithm that is work efficient but $O(\log n \log\log k)$ span. We then discuss several improvements that show how to optimize the span. As a byproduct of our span optimization, we also optimize the span of the randomized variant of the problem from Chapter 4 and obtain our improved result (Theorem 1) which brings the span down from $O(\log n \log^* n)$ to $O(\log n)$.

The workings of the deterministic algorithm will be reminiscent the analysis of the randomized algorithm in Chapter 4. Many of the notations that were key to its analysis (affected vertices, spreading, affected components, etc.) are employed to build the deterministic direct-update update algorithm.

### Preliminary: Colorings and maximal independent sets

Parallel algorithms for graph coloring and maximal independent sets are well studied [39, 78, 79, 110, 127]. 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 [79] give an algorithm that finds an $O(\log^{(c)}(n))$-coloring of a constant-degree graph in $O(c \cdot n)$ work and $O(c)$ span in the EREW model. This gives a constant coloring in $O(n \log^* n)$ work and $O(\log^* n)$ span.

Given a $c$-coloring, one can easily obtain a maximal independent set as follows. Sequentially, for each color, look at each vertex of in parallel. If a vertex is the current color and 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 coloring, 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. We first produce a $\log^{(c)} n$ coloring in $O(1)$ time, then bucket sort the vertices by color, allowing us to perform the coloring-to-independent-set conversion work efficiently. Cole and Vishkin [40] show that the bucket sorting can be done efficiently in the EREW model. This results in an $O(n)$ work, $O(\log^{(c)} n)$ span algorithm for maximal independent set in a constant-degree graph.

**Lemma 18** ([40, 79])**.** There exists a deterministic algorithm that finds an MIS in a constant-degree graph in $O(n)$ work and $O(\log^{(c)} n)$ span for any constant $c$.

## 5.2 A Deterministic Contraction Algorithm

Like our randomized algorithm in Chapter 4, our deterministic algorithm maintains a *contraction data structure* which serves to record the process of tree contraction beginning from the input forest $F$ as it contracts to a forest of singletons. The contraction data structure also contains the RC-Tree as a byproduct. For each round in which a vertex is live, the contraction data structure stores an adjacency list for that vertex. We assume that the forests have bounded degree $t$, so each adjacency list is exactly $t$ slots large. Each entry in a vertex $v$'s adjacency list is one of four possible kinds of value:

1. Empty, representing no edge
2. A pointer to an edge of $F$ adjacent to $v$

3. A pointer to a binary cluster for which $v$ is one of the boundary vertices. This represents an edge between $v$ and the other boundary vertex in the contracted tree.
4. A pointer to a unary cluster for which $v$ is the boundary vertex. This does not represent any edge in the contracted tree, but is used to propagate augmented data from the child to the parent.

At round 0 (before any contraction), the adjacency list simply stores pointers to the edges adjacent to each vertex. At later rounds, in a partially contracted tree, some of the edges are not original edges of $F$, but are the result of a compress operation and represent a binary cluster of $F$ (Case 3), 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 4).

Clusters contain pointers to their child clusters alongside any augmented data. Each composite cluster corresponds uniquely to the vertex that contracted to form it, so counting them plus the base edge clusters, the RC-Tree contains exactly $n+m$ clusters. If the user wishes to store augmented data on the vertices, this can be stored on the unique composite cluster for which that vertex is the representative. Alternatively, explicit base vertex clusters may be used if desired.

## The static algorithm

We build a tree contraction, and as a byproduct an RC-Tree, deterministically using a variant of Miller and Reif's tree contraction algorithm. Recall that Miller and Reif's algorithm contracts all degree-one vertices (leaves) and a random independent set of degree-two vertices chosen by flipping coins to break symmetry. In our deterministic algorithm, instead of contracting all degree-one vertices 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. The reason for this will become clear during the analysis of the update algorithm, 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 would force it to contract, which may cause a chain reaction by forcing its neighbor to not contract (otherwise it would violate being an independent set), which may force the neighbor's neighbor to contract to maintain maximality and so on. Such a chain reaction is undesirable, and our variant avoids it.

**Definition 9.** 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.

Let us denote by $F_0$, the initial forest, then by $F_i$ 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 finding an $O(\log\log n)$ coloring of the vertices using the algorithm of Goldberg and Plotkin [79], then bucket sorting the vertices by colors and selecting a maximal independent set similar to the algorithm of Cole and Vishkin [40]. 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 set of $F_i$, filtering those which were selected to contract. This also takes $O(|F_i|)$ work and $O(\log\log n)$ span [81].

As is standard for parallel tree contraction implementations, each vertex writes into its neighbors' adjacency list for the next round. For vertices that do not contract, they can simply copy their corresponding entry in their neighbors' adjacency list to the next round. For vertices that do contract, they write the corresponding cluster pointers into their neighbors' adjacency lists. For example, if a vertex $v$ with neighbors $u$ and $w$ compresses, it writes a pointer to the binary cluster formed by $v$ (whose boundary vertices are $u$ and $w$) into the adjacency list slots of $u$ and $w$ that currently store the edge to $v$. The tree has constant degree, so identifying the slot takes constant time and suffers no issues of concurrency.

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 finalizes (creates a root cluster).

## 5.3 A Deterministic Dynamic Update Algorithm

**Algorithm 5** Batch update

```
1: procedure BatchUpdateForest($E^+, E^-$)
2:     Update $F_0$, adding edges in $E^+$ and removing edges in $E^-$
3:     Determine affected vertices from the endpoints of $E^+ \cup E^-$
4:     for each level $i$ from 1 to $\log_{6/5} n$ do
5:         Determine the new affected vertices from the previous set of affected vertices
6:         Find a maximal set of eligible affected vertices without an unaffected neighbor that contracts in $F_i$
7:         Obtain $F_i'$ from $F_i$ by uncontracting the affected vertices, then contracting the new MIS
```

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. The goal of the update algorithm is now to produce $F_i'$, an updated tree contraction for each level $i$, using $F_i, F_{i-1}$ and $F_{i-1}'$.

**Affected vertices** To perform the update efficiently, we define the notion of an *affected vertex*, which is very similar to the same notion that was defined for the randomized algorithm in Chapter 4. Note that in the randomized algorithm, the notion of an affected vertex was used only for analysis, since the change-propagation algorithm determined them automatically (via affected computations), whereas now, they are a key part of the algorithm itself.

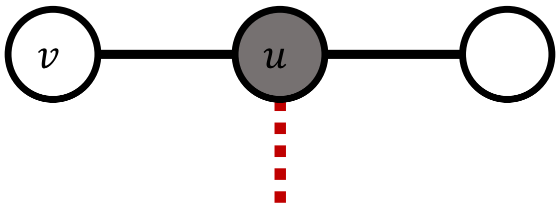


**Figure 5.1: A vertex $v$ is affected because its neighbor $u$ is affected by a newly inserted edge (red), and $v$ depends on $u$ contracting in order for the contraction to be maximal.**

**Definition 10** (Affected vertex)**.** A vertex is *affected* at level $i$ if any of:

1. it is alive in one of $F_i$ and $F_i'$ but not the other (which means it either contracted in an earlier round originally, but survived later after the update, or vice versa).
2. it is alive in both $F_i$ and $F_i'$ but has a different adjacency list
3. it is alive in both $F_i$ and $F_i'$, does not contract in $F_i$, but all of its neighbors $u$ in $F_i$ that contracted are affected.

The first two cases of affected vertices are intuitive. If a vertex used to exist at round $i$ but no longer does, or vice versa, it definitely needs to be updated in round $i$. 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, and is important for the correctness and efficiency of the algorithm. Suppose an eligible vertex $v$ doesn't contract in round $i$. Since the contraction forms a maximal independent set, at least one of $v$'s neighbors must contract. If all such neighbors are affected, they may no longer contract after in the updated forest, which would leave $v$ uncontracted and without a contracting neighbor, violating maximality. Therefore $v$ should also be considered in the update. Figure 5.1 shows an example scenario where this is important.

Note that by the definition, vertices only become affected because they have an affected neighbor either in the previous round or the same round. Similar to the randomized algorithm of Chapter 4, we call this *spreading* affection.

## The algorithm

With the definition of affected vertices, the update algorithm can be summarized as stated in Algorithm 5. Each level is processed sequentially, while the subroutines that run on each level are parallel. Line 5 is implemented by looking at each affected vertex of the previous round 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 Goldberg and Zwick's approximate compaction algorithm [81], this step takes linear work in the number of affected vertices and

$O(\log\log n)$ span. In Section 4.3.2, we show that the number of affected vertices at each level is $O(k)$, so this is efficient.

Line 6 is accomplished using Lemma 18. This finds a maximal independent set of eligible affected vertices in linear work and $O(\log^{(c)} n)$ span, so we can choose $c = 2$ and use the fact that there are $O(k)$ affected vertices to find the new maximal independent set in $O(k)$ work and $O(\log\log k)$ span.

Line 7 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 round $i+1$ 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.

## Correctness

We now argue that the algorithm is correct.

**Lemma 19.** 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.* The goal is to show that every vertex satisfies the necessary invariant, i.e. that every vertex either contracts or is adjacent to a vertex that contracts but not both. Call a vertex *update-eligible* if it has degree one or two in $F_i'$ (is eligible in $F_i'$) and is not adjacent to an unaffected vertex that contracts in $F_i$. The affected and update-eligible vertices are the ones that the update algorithm computes a new MIS on.

Consider any update-eligible unaffected vertex $v \in F_i'$. Since it is unaffected it exists in $F_i$. Suppose $v$ contracts in $F_i$, then it still contracts 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 doesn't contract in $F_i'$. If $u$ is an affected neighbor, it is not update-eligible and hence does not contract since $v$ is unaffected and contracted. Therefore none of $v$'s neighbors contract in $F_i'$, so the invariant is satisfied when $v$ contracts in $F_i'$.

Now suppose $v$ doesn't contract in $F_i'$. Since it is unaffected it exists and doesn't contract in $F_i$. Therefore it has a neighbor $u \in F_i$ that contracts. If $u$ is unaffected, then $u \in F_i'$ and contracts. If $u$ were affected, then $v$ would be affected by dependence, so $u$ must be unaffected. Therefore all update-eligible unaffected vertices $v \in F_i'$ satisfy the invariant.

Now consider any update-eligible affected vertex $v \in F_i'$. Since it is affected and update-eligible, $v$ participates in the MIS, and has no contracted neighbor in $F_i$. Since the algorithm finds an MIS on the candidates, $v$ either contracts and has no contracted neighbor, or doesn't contract and has a contracted candidate neighbor. Therefore $v$ satisfies the invariant.

Together, we can conclude that every update-eligible vertex satisfies the invariant. Lastly, if a vertex is not update-eligible, then it satisfies the invariant by definition since it either has degree greater than two and hence can not contract, or it is adjacent to an unaffected vertex that contracts in $F_i$, which must also contract in $F_i'$ by virtue of being unaffected. □

It remains to show that the algorithm is efficient.

## 5.4 Performance 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.

### 5.4.1 Round and Tree-Size Bounds

**Lemma 20.** Consider a tree $T$ and suppose a maximal indpendent set of degree one and two vertices is contracted via *rake* and *compress* contractions to obtain $T'$. Then

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

*Proof.* More than half of the vertices in any tree have degree one or two [183]. 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. Three important corollaries follow that allow us to bound the cost of various parts of the algorithm. Corollary 3 gives the number of rounds required to fully contract a forest, and Corollary 4 gives a bound on the number of rounds required to shrink a forest to size $n/\log n$. Lastly, Corollary 5 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 3.** Given a forest on $n$ vertices, maximal tree contraction completely contracts a forest of $n$ vertices in $\log_{6/5} n$ rounds.

**Corollary 4.** Given a forest on $n$ vertices, after performing at least $\log_{6/5} \log n$ rounds of maximal tree contraction, the number of vertices in the resulting forest is at most $n/\log n$.

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

*Proof of Corollary 3,4,5.* By Lemma 20, 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. □

### 5.4.2 Analysis of the Static Algorithm

With the lemmas and corollaries of Section 5.4.1, we can now analyze the static algorithm.

**Theorem 21.** The basic 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 20, 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 Lemma 18. By Corollary 3 there are $O(\log n)$ rounds, hence the total span is $O(\log n \log\log n)$. □

Section 5.5 explains how to improve the span to $O(\log n \log^{(c)} n)$ for any constant $c$.

### 5.4.3 Analysis of the Update Algorithm

The analysis of the update algorithm follows a similar pattern to the analysis of the randomized change propagation algorithm of Chapter 4. We sketch a summarised version below, which should be reminiscent of the randomized version, then present the full analysis.

We begin by establishing the criteria for vertices becoming affected. Initially, the end-points 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 allows us to establish that the algorithm is efficient.

## The proofs

We now prove the afformentioned facts.

**Lemma 21.** If $v$ is unaffected at round $i$, 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. □

If a vertex is not affected during round $i$ but is affected during round $i+1$, we say that $v$ *becomes affected in round $i$*. A vertex can become affected in two ways.

**Lemma 22.** 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 round $i$. Therefore, due to Lemma 21, $v$ does not contract, otherwise it would do so in both $F$ and $F'$ and hence be unaffected at round $i+1$. 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 21, in either case, $v$ has the same set of neighbors in $F_{i+1}$ and $F_{i+1}'$. Therefore, since $v$ is affected in round $i+1$, 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 $\neg(1) \Rightarrow (2)$, we must have $(1) \vee (2)$. □

**Definition 11** (Spreading)**.** An affected vertex $u$ *spreads to* $v$ if $v$ was unaffected at the beginning of round $i$ and became affected in round $i$ (is affected in round $i+1$) because

1. $v$ is a neighbor of $u$ at round $i$, and $u$ contracts in round $i$ in either $F_i$ or $F_i'$, or
2. $v$ does not contract in round $i+1$ and is a neighbor of $u$ which does.

We call Case (1), *spreading directly* and Case (2) *spreading by dependence*. Figure 5.2 shows two examples of directly spreading affection. Figure 5.3 shows an example of spreading affection by dependence. 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 $A^i$ denote the set of affected vertices in round $i$.

**Lemma 23.** For a batch update of $k$ edges, we have $|A^0| \leq 6k$.

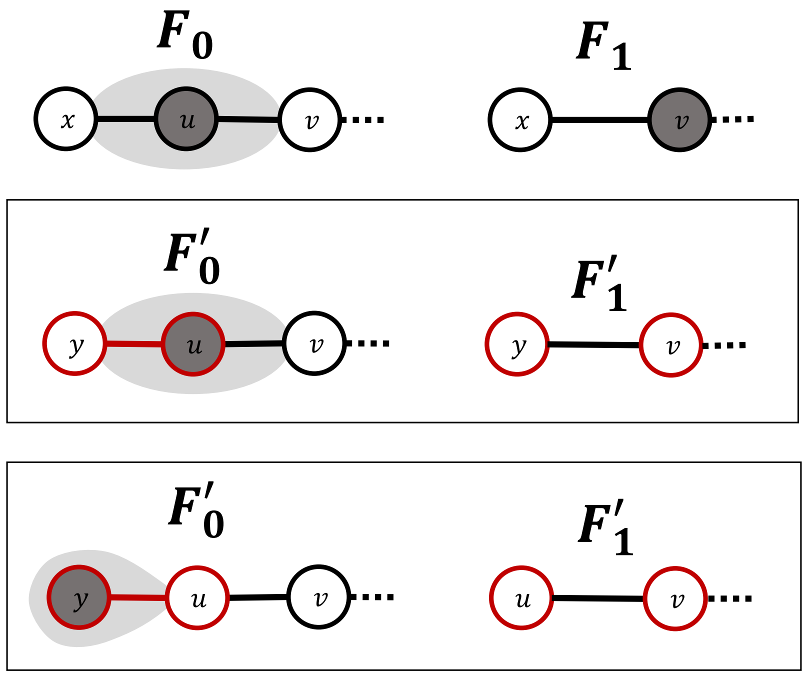


Figure 5.2: Direct affection (two possibilities): A vertex $u$ directly affects its neighbor $v$. $u$ is affected at round 0 since its adjacency list was changed. Since $u$ contracts in $F_0$ and changes the adjacency list of $v$ at round 1, $v$ becomes affected. Note that this can happen whether or not $u$ contracts in $F'_0$ as shown in the second possibility.

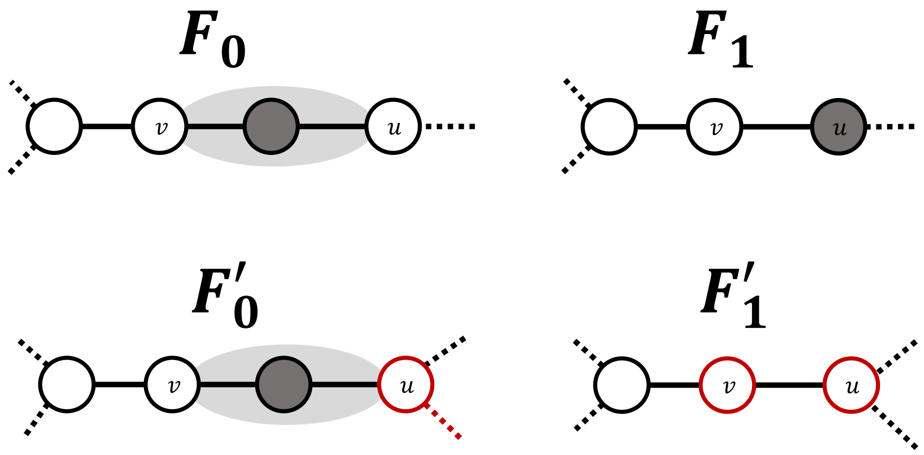


Figure 5.3: Affection by dependence: A vertex $u$ affects its neighbor $v$ by dependence. Even though $v$'s adjacency list is the same in both forests, it is affected because it depended on $u$ to contract in $F_1$ for maximality to be satisfied. Since $u$ can no longer contract in $F'_1$, it is important that $v$ is able to.

*Proof.* A single 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 in total. □

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

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

**Definition 12** (Frontier)**.** A vertex $v$ is a *frontier* at round $i$ if $v$ is affected at round $i$ and one of its neighbors in $F_i$ is unaffected at round $i$.

**Lemma 24.** If $v$ is a frontier vertex at round $i$, then it is alive in both $F_i$ and $F'_i$ at round $i$, 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 alive 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 to a vertex in round $i$, then $v$ must be a frontier. Our next goal is to analyze the structure of the affected sets and then show that the number of frontier vertices is small.

**Lemma 25.** For all $i, j$, the subforest induced by $A^i_j$ in $F_i$ is a tree

*Proof.* The rake and compress operations both preserve the connectedness of the underlying tree, and Lemma 22 shows that affection only spreads to neighboring vertices. □

**Lemma 26.** $A^i_j$ has at most two frontiers and $|A^{i+1}_j \setminus A^i_j| \le 4$.

*Proof.* We proceed by induction on $i$. At round 0, each component contains one vertex, so it definitely contains at most 2 frontier vertices. Consider some $A_j^i$ and suppose it contains one frontier vertex $u$, which may spread directly by contracting (Definition 11). 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 24, 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 24, 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 11) in $A_j^{i+1}$ and contracts in $F_{i+1}$. Since $u$ contracts in $F_{i+1}$, it has at most two neighbors, and by Lemma 24, 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, it may become a frontier and hence there are at most two frontiers. 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 frontiers, and $|A_j^{i+1} \setminus A_j^i| \le 3$.

Now consider some $A_j^i$ that contains two frontier vertices $u_1, u_2$. By Lemma 25, $u_1$ 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 $|A_j^{i+1} \setminus A_j^i| \le 4$. □

Now define $A_{F,j}^i = A_j^i \cap V_F^i$, the set of affected vertices from $A_j^i$ that are live in $F$ at round $i$, and similarly define $A_{F',j}^i$ for $F'$.

**Lemma 27.** For every $i, j$ we have

$$|A_{F,j}^i| \le 26.$$

*Proof.* Consider the subforest induced by the set of affected vertices $A_{F,j}^i$. By Lemmas 25 and 26, this is a tree with two frontier vertices. The update algorithm finds and contracts a maximal independent set of affected degree one and 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 25, so by Lemma 20, the size of the new affected set is

$$\begin{aligned}|A_{F,j}^{i+1}| &\le 4 + \frac{5}{6}\left(|A_{F,j}^i| - 2\right) + 2 \\ &= \frac{26}{6} + \frac{5}{6}|A_{F,j}^i|\end{aligned}$$

Since $|A_{F,j}^0| = 1$, we obtain

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

□

**Lemma 28.** Given a batch update of $k$ edges, for every $i$

$$|A^i| \le 312k$$

*Proof.* By Lemma 23, there are at most $6k$ affected components. At any level, every affected vertex must be live in either $F$ or $F'$, so $A^i_j = A^i_{F,j} \cup A^i_{F',j}$, and hence

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

□

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 22** (Update performance)**.** A batch update consisting of $k$ edge insertions or deletions takes $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ 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 round $r = \log_{6/5}\left(1 + \frac{n}{k}\right)$. By Lemma 28, there are $O(k)$ affected vertices per level, so the work performed on levels up to including $r$ is

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

By Corollary 5, after $r$ rounds of contraction, there are at most $k$ live vertices remaining 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 rounds 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

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

It takes $O(\log^{(c)} k)$ span to find a maximal independent set of the affected vertices for any constant $c$, so we can choose $c = 2$ to match the span of approximate compaction required to filter out the vertices that are no longer affected in the next round. Each round takes $O(\log\log k)$ span, so over $O(\log n)$ rounds, this results in $O(\log n \log k \log k)$ span. □

## 5.5 Optimizations

Our static tree contraction algorithm and our basic result on dynamically updating it are work-efficient ($O(n)$ and $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work respectively) and run in $O(\log n \log\log n)$ and $O(\log n \log\log k)$ span respectively. In both cases, there are two bottlenecks to the span: computing a maximal independent set, and performing approximate compaction to remove vertices that have contracted or are no longer affected. Improving the span of the maximal indpendent set is easy since it runs in $O(\log^{(c)} n)$ span for any $c$, and we can just choose a smaller $c$ (the basic algorithm chose $c = 2$ to match the span of approximate compaction).

Therefore, the only remaining bottleneck is the approximate compaction, which we will improve in this section. We first describe a faster static algorithm, which introduces the techniques we will use to improve the update algorithm. We also describe an improvement that eliminates the need for concurrent writes since approximate compaction requires the Common CRCW model, which is more powerful than necessary.

Lastly, we will also show that our span optimization technique can be used to speed up the randomized variant of the algorithm.

### 5.5.1 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, which is not work efficient. This leads to an $O(n)$ work and $O(\log n \log\log n)$ span algorithm in the Common CRCW model using the $O(\log\log n)$-span approximate compaction algorithm of Goldberg and Zwick [81]. We can improve the span easily as follows 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 takes at most $O(\log\log n)$ rounds. This is essentially the same strategy used by Gazit, Miller, and Teng [70]. The work of the first phase is therefore $O(n)$ and the span, using approximate compaction, is

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

**Phase Two** In the second phase, we run the "wasteful" algorithm, which is simply the same algorithm but not performing any compaction. Since the forest begins with $O(n/\log n)$ vertices in this phase, this takes $O(n)$ work and completes in $O(\log n)$ rounds. Since the span bottleneck is finding the maximal indepedent set in each round, the span is $O(\log n \log^{(c)} n)$ for any constant $c$. Putting these together, the total work is $O(n)$, and the span is

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

### 5.5.2 Eliminating Concurrent Writes

The above algorithm still uses approximate compaction which requires the power of the Common CRCW model. We now describe a variant without this requirement. Phase Two is the same since it performs no compaction, so we just have to improve Phase One. We do so by arbitrarily partitioning the vertices into $n/\log n$ buckets of size $O(\log n)$. Each round, the algorithm considers each bucket and each vertex within each bucket in parallel. After performing each round of contraction, each bucket independently filters the vertices that contracted using an exact filter algorithm instead of approximate compaction. Since each bucket has size $O(\log n)$, the span is still $O(\log\log n)$ without requiring concurrent writes.

Since there are $n/\log n$ buckets, each round takes an additional $O(n/\log n)$ work, but over $\log\log n$ rounds, this amounts to less than $O(n)$ additional work, so the algorithm is still work efficient. At the end of the phase, collect the vertices back into a single bucket in $O(n)$ work and $O(\log n)$ span, then proceed with Phase Two.

### 5.5.3 A Lower Span Dynamic Algorithm

The span of the dynamic algorithm is also bottlenecked by the span of approximate compaction, which is used on the affected vertices each round to remove vertices that are no longer affected. We optimize the dynamic algorithm similarly to the static algorithm, by splitting it into three phases this time. Each phase will maintain a set of buckets of affected vertices. Each round processes each bucket and each vertex within each bucket in parallel, placing newly affected vertices into one of the buckets from which affection spread to it. Since it is possible for multiple neighbors of a vertex to spread to it at the same time, 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 bucket. Since the forest has constant degree this can be checked in constant time. At the end of each round, no-longer-affected vertices are filtered from each bucket.

The key is varying the size of the buckets in each phase. Having too many buckets results in performing extra work and not being work efficient, while having too few buckets increases the span. By varying the number of buckets and hence the size of each bucket in each phase, we can obtain a tradeoff that leads to work efficiency and minimal span.

**Phase One** The algorithm will run Phase One for $\log_{6/5}\left(1+\frac{n}{k}\right)$ rounds. Note importantly that this depends on the batch size $k$, so the number of rounds per phase is not always the same for each update operation.

Phase one starts by creating a singleton bucket for each affected vertex. There are therefore $O(k)$ buckets in Phase One. In each round, the algorithm processes each bucket and each affected vertex within each bucket in parallel. This process is efficient because according to Lemma 27, each bucket will have constant size (the buckets contain the affected components from the analysis in Section 5.4), so filtering the no-longer-affected vertices from the buckets takes constant time per bucket.

Having to maintain this set of $k$ buckets adds an additional $O(k)$ work to each round since the algorithm may encounter empty buckets while looping over all of the vertices, but since we run Phase One for only $O\left(\log\left(1+\frac{n}{k}\right)\right)$ rounds, the algorithm is still work efficient. Furthermore, each round takes just an additional constant amount of span.

**Phase Two** Phase Two will redistribute the affected vertices into $k/\log k$ buckets of size $O(\log k)$. First, it collects the contents of each of the $O(k)$ buckets of Phase One back into a single array of $O(k)$ affected vertices. This takes $O(k)$ work and $O(\log k)$ span. Then, it partitions them into $k/\log k$ buckets of size $O(\log k)$.

We then run the update algorithm for $\log\log k$ more rounds. Again, in each round, the algorithm processes each bucket and each affected vertex within each bucket in parallel. Due to Lemma 27, the size of each bucket can not grow beyond a constant-factor larger, so every bucket remains at most $O(\log k)$ large across all rounds. The additional work due to maintaining the buckets over all rounds is $O((k/\log k)\log\log k) = O(k)$, and each round takes an additional $O(\log\log k)$ span due to the filtering.

**Phase Three** After completing Phase Two, by Corollaries 4 and 5, there can be at most $O(k/\log k)$ vertices alive in the forest, and hence at most twice that many affected vertices (affected vertices may be alive in either the new or old forest). Phase Three continues to use $O(k/\log k)$ buckets, but begins by collecting the remaining affected vertices from the buckets of Phase Two and load balancing them into constant-size buckets. As usual, in each round, the algorithm processes each bucket and each vertex within each bucket in parallel. Each bucket remains constant size by Lemma 27 and the work performed in each round is at most $O(k/\log k)$ for $O(\log k)$ rounds, a total of $O(k)$ work. Since each bucket is constant size, filtering them takes constant time and hence adds just a constant amount of span. After $O(\log k)$ rounds of Phase three, the forest is fully contracted and the update is complete.

**Putting it together** In total, at most $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ additional work is added by the bucket maintenance, so the algorithm is still work efficient. The algorithm performs a total of $O(\log n)$ rounds, in each of which the span is dominated by computing the MIS, which costs $O(\log^{(c)} n)$ for any constant $c$. Adding the additional span of each round, which was constant in Phase One for $O\left(\log\left(1+\frac{n}{k}\right)\right)$ rounds, $O(\log\log k)$ in Phase Two for $O(\log\log k)$ rounds, and constant in Phase Three for $O(\log k)$ rounds, the final span is

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

$$= O\left(\log n \log^{(c)} k\right).$$

Since this is equal to the span of computing the MIS in each round, any further span improvement would require a more efficient algorithm for MIS, or a completely different algorithm for tree contraction all together, so we essentially have an optimal implementation of this particular algorithm.

### 5.5.4 A Lower Span Randomized Algorithm

With the span optimization above, the remaining bottleneck is entirely due to the subroutine for finding the MIS. Our optimization was designed to remove the span caused by approximate compaction. In the randomized variant of the algorithm from Chapter 4, the total span is $O(\log n \log^* n)$, where the $\log^* n$ factor also comes from performing approximate compaction in each round (which is $O(\log^* n)$ when randomization is allowed). In the randomized variant, however, finding the independent set takes constant span rather rather than $O(\log^{(c)} k)$ since it relies only on local coin flips. Our span optimization can therefore be applied to the randomized algorithm as well in exactly the same way.

Recall that a constant fraction of the vertices contract on each round, and that the contraction process takes $O(\log n)$ rounds w.h.p. We can therefore substitute our deterministic MIS with the randomized independent set based on coin flips and use the definition of affected vertices from Chapter 4 to obtain a more efficient randomized direct-update algorithm which no longer uses self-adjusting computation. The analysis of Chapter 4 showed that the randomized variant has all of the same necessary properties and implies that the resulting algorithm is work efficient, running in $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ expected work for a batch of $k$ updates, and in just $O(\log n)$ span.

## 5.6 Discussion

In this chapter we broke the curse of randomization that haunted work-efficient parallel batch-dynamic graph algorithms by designing the first such algorithm for parallel tree contraction, and to the best of our knowledge, the first for any graph problem. This immediately gives as a consequence, a corresponding deterministic parallel batch-dynamic algorithm for RC-Trees. Our algorithm performs $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work for a batch of $k$ links and cuts and runs in $O(\log n \log^{(c)} k)$ span for any constant $c$. We also applied our techniques to improve the span of the randomized variant from $O(\log n \log^* n)$ to just $O(\log n)$.

Several interesting questions still remain open. Our deterministic algorithm requires $O(\log n \log^{(c)} k)$ 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. The tree contraction needs only to have the property that it contracts a constant fraction of the vertices in any subtree in order to obtain our bounds, so any constant ruling set would suffice if one could compute it in $O(1)$ span.

Prior algorithms for deterministic tree contraction are based on Cole and Vishkin's deterministic coin tossing technique [40] (which happens to be a subroutine used by our maximal independent set algorithm). It would be interesting to investigate whether this could be used directly to obtain a more efficient dynamic algorithm. Lastly, it would be interesting to explore which other parallel batch-dynamic graph problems can be derandomized, perhaps using our deterministic RC-Trees as an ingredient, or independently.

# Part II

# Parallel Batch-Dynamic Graph Algorithms

# Chapter 6
# Parallel Batch-Dynamic Graph Connectivity

## 6.1 Introduction

Understanding the connectivity structure of graphs is of significant practical interest, for example, due to its use as a primitive for clustering the vertices of a graph [156]. Due to the importance of connectivity there are several implementations of parallel batch-dynamic connectivity algorithms [106, 107, 130, 158, 181, 184]. In the worst case, however, these algorithms may recompute the connected components of the entire graph even for very small batches. Since this requires $O(m+n)$ work, it makes the worst-case performance of the algorithms no better than running a static parallel algorithm. On the theoretical side, existing batch-dynamic efficient connectivity algorithms have only been designed for restricted settings, e.g., in the incremental setting when all updates are edge insertions [162] (usually referred to as the union-find problem), or when the underlying graph is a forest, such as batch-dynamic Euler-tour trees [174], and our batch-dynamic RC-Trees (Chapters 3–5).

In the sequential setting, dynamic connectivity has received tremendous amounts of attention for decades. Frederickson gave the first efficient algorithm for fully dynamic connectivity which can process updates in $O(\sqrt{m})$ time, and queries in $O(1)$ time using his *topology tree* data structure [63]. Eppstein et al. [55] developed the *sparisification technique* which improves the running time of Frederickson's algorithm to $O(\sqrt{n})$ per update.

Henzinger and King were then the first to break the polylogarithmic barrier by giving an algorithm for fully dynamic connectivity that runs in $O(\log^3 n)$ amortized expected update time and $O(\log n/\log\log n)$ time per query. Unlike previous results, they utilized amortization and randomization. The update time was then improved to $O(\log^2 n)$ expected amortized time by Henzinger and Thorup. Soon after, Holm, De Lichtenberg, and Thorup gave a deterministic version of the algorithm with $O(\log^2 n)$ amortized update times.

As the landscape of dynamic connectivity evolved, it became clear that there would not be one single "best" algorithm, since algorithms with incomparable runtimes were being developed; some deterministic, some randomized, some amortized, some worst case. Thorup [173] quickly developed a faster randomized-amortized algorithm, processing updates in $O(\log n(\log\log n)^3)$ expected amortized time and queries in $O(\log n/\log\log\log n)$, while Wulff [185] took the deterministic crown with an $O(\log^2 n/\log\log n)$ amortized update.

The landscape was about to evolve even further by adding another category of algorithms. Kapron, King, and Mountjoy [111] escaped the prison of amortization and developed the first polylogarithmic worst-case algorithm that runs updates in $O(\log^5 n)$ time. Queries are

Monte Carlo (only correct w.h.p.), and run in $O(\log n/\log\log n)$ time. The update time was improved to $O(\log^4 n)$ by Gibb et al. [75] and Wang [182] independently.

After seeing no attention for decades, a breakthrough was then achieved in the deterministic worst-case category, with an $O\left(\sqrt{n(\log\log(n))^2/\log(n)}\right)$-time update algorithm with $O(1)$ queries by Kejlberg-Rasmussen et al. [120]. Wulff-Nilsen then gave an $O(n^{0.5-c})$ w.h.p. (Las Vegas) update algorithm.

Most recently, the fastest amortized result is $O(\log n(\log\log n)^2)$ expected amortized time per update by Huang et al. [103] who improve Thorup's earlier result [173]. Lastly, Nanongkai and Saranurak give a Monte Carlo algorithm with $O(n^{0.4+o(1)})$ worst-case update time and a Las Vegas algorithm with $O(n^{0.49306})$ worst-case update time w.h.p., both of which work against an adaptive adversary, meaning that the adversary may request updates that depend on the previous outputs of the algorithms.

**Lower bounds** Pătrașcu and Demaine [149] give an $\Omega(\log n)$ lower bound for the dynamic connectivity problem. More precicely, they show that for any $t(n) = \Omega(1)$, an algorithm with update time $O(t(n)\log n)$ requires a corresponding query time of $\Omega(\log n/\log t(n))$. Though existing algorithms are coming close, it remains an open problem whether there exists an algorithm with update and query times of $O(\log n)$ in any setting (amortized or randomized or otherwise).

**Our approach** Parallel algorithms for connectivity have a long history [18, 41, 97, 118, 145, 160, 180], and there are many existing algorithms that solve the problem work-efficiently and in low-span [42, 69, 86, 87, 144, 147, 161]. However, there is no obvious way to adapt existing parallel connectivity algorithms to the dynamic setting, particularly for batch updates.

Our strategy will be to start with the sequential dynamic algorithm of Holm, De Lichtenberg, and Thorup [100], which we refer to as the HDT algorithm, and adapt it for the parallel batch-dynamic setting. The algorithm maintains a set of $O(\log n)$ spanning forests, each of which is represented by an Euler-tour tree. Since we have existing parallel batch-dynamic algorithms for Euler-tour trees, this seems like a promising direction in which to search for a batch-dynamic algorithm for connectivity. We will describe two algorithms. First, a simplified algorithm that illustrates some of the key ideas for obtaining efficient batching, then a more optimized algorithm with better work efficiency and better span.

## 6.2 The Sequential Algorithm and Data Structure

We will first give an overview of the sequential HDT algorithm. The HDT algorithm assigns to each edge in the graph, an integer *level* from 1 to $\log n$. The levels correspond to sequence of subgraphs $G_1 \subset G_2 \subset \ldots \subset G_{\log n} = G$, such that $G_i$ contains all edges with level at most $i$. The algorithm also maintains a spanning forest $F_i$ of each $G_i$ such that $F_1 \subset F_2 \subset \ldots \subset F_{\log n}$. Each forest is maintained using a set of augmented Euler-tour trees. Throughout the algorithm, the following invariants are maintained.

**Invariant 1.** $\forall i = 1 \dots \log n$, the connected components of $G_i$ have size at most $2^i$.

**Invariant 2.** $F_{\log n}$ is a minimum spanning forest where an edge is weighted by its level.

**Connectivity Queries** To perform a connectivity query in $G$, it suffices to query $F_{\log n}$, which takes $O(\log n)$ time by querying for the root of each Euler-tour tree and returning whether the roots are equal. We note that in [100], a query time of $O\left(\log n / \log\log n\right)$ is achieved by storing the Euler tour of $F_{\log n}$ in a B-tree with branching factor $\log n$.

**Inserting an Edge** An edge insertion is handled by assigning the edge to level $\log n$. If the edge connects two currently disconnected components, then it is added to $F_{\log n}$.

**Deleting an Edge** Deletion is the most interesting part of the algorithm. If the deleted edge is not in the spanning forest $F_{\log n}$, the algorithm removes the edge and does nothing to $F_{\log n}$ as the connectivity structure of the graph is unchanged. Otherwise, the component containing the edge is split into two. The goal is to find a *replacement edge*, that is, an edge crossing the split component.

If the deleted edge had level $i$, then the *smaller* of the two resulting disconnected components is searched starting at level $i$ in order to locate a replacement edge. Before searching this component, all tree edges whose level is equal to $i$ have their level decremented by one. As the smaller of the split components at level $i$ has size $\le 2^{i-1}$, pushing the entire component to level $i-1$ does not violate Invariant 1. Next, the non-tree edges at level $i$ are considered one at a time as possible replacement edges. Each time the algorithm examines an edge that is not a replacement edge, it decreases the level of the edge by one. If no replacement is found, it moves up to the next level and repeats. Note that because the algorithm first pushes all tree edges to level $i-1$, any subsequent non-tree edges that may be pushed from level $i$ to level $i-1$ will not violate Invariant 2.

## Implementation and Cost

To efficiently search for replacement edges, the Euler-tour trees are augmented with two additional pieces of information. The first augmentation is to maintain the number of non-tree edges whose level equals the level of the tree. The second augmentation maintains the number of tree-edges whose level is equal to the level of the tree.

Using these augmentations, each successive non-tree edge (or tree edge) whose level is equal to the level of the tree can be found in $O(\log n)$ time. Furthermore, checking whether the edge is a replacement edge can be done in $O(\log n)$ time. Lastly, the cost of pushing an edge that is not a replacement edge to the lower level is $O(\log n)$, since it corresponds to inserting the edge into an adjacency structure and updating the augmented values. Since each edge can be processed at most once per level, paying a cost of $O(\log n)$, and there are $\log n$ levels, the overall amortized cost per edge is $O(\log^2 n)$.

## 6.3 Parallel Data Structures

The sequential algorithm represents the spanning forest using Euler-tour trees. In our parallel batch-dynamic algorithm, each spanning forest, $F_i$, will be represented using a parallel batch-dynamic Euler-tour tree [174]. We could alternatively use our own parallel batch-dynamic RC-Trees from Chapter 3, but the augmentations required by the algorithm are simpler with Euler-tour trees since the original algorithm used them. Using RC-Trees would also require ternarizing the graph, while Euler-tour trees avoid this. In this section we will describe the parallel data structures used by our algorithm, including our data structure for storing the edges of the graph and the necessary augmentations to the batch-dynamic Euler-tour tree data structure.

### 6.3.1 Adjacency Arrays

We represent the edges of the graph in a parallel dictionary $E_D$ for convenience. We also store an adjacency array, $A_i[u]$, at each level $i$, and for each vertex $u$ to store the tree and non-tree edges incident on $u$ with level $i$. Note that tree and non-tree edges are stored separately so that they can be accessed separately. The adjacency arrays support batch insertion and deletion of edges, as well as the ability to fetch a batch of edges of a desired size.

- **INSERTEDGES**($\{e_1, \ldots, e_l\}$): Insert a batch of edges adjacent to this vertex.
- **DELETEEDGES**($\{e_1, \ldots, e_l\}$): Delete a batch of edges adjacent to this vertex.
- **FETCHEDGES**($l$): Return a set of $l$ arbitrary edges adjacent to this vertex.

**Lemma 29.** INSERTEDGES, DELETEEDGES, and FETCHEDGES can be implemented in $O(1)$ amortized work per edge and in $O(\log n)$ span.

*Proof.* For a given vertex, the data structure stores a list of pointers to each adjacent edge in a resizable array. Each edge correspondingly stores its positions in the adjacency arrays of its two endpoints. Since each vertex can have at most $O(n)$ edges adjacent to it, the adjacency arrays are of size at most $O(n)$.

Insertions are handled by inserting the batch onto the end of the array, and resizing if necessary. This costs $O(1)$ amortized work per edge and $O(\log n)$ span. To fetch $l$ elements, we return the first $l$ elements of the array, which takes $O(1)$ work per edge and $O(\log n)$ span.

Finally, to delete a batch of $l$ edges, the algorithm first determines which of the edges to be deleted are contained within the final $l$ elements of the array. It then compacts the final $l$ elements of the array, removing those edges. Compaction costs $O(l)$ work and $O(\log n)$ span. The algorithm then considers the remaining $l'$ edges to be deleted, and in parallel, swaps these elements with the final $l'$ elements of the array. The final $l'$ elements in the array can then be safely removed. Note that any operation that moves an element in the array also updates the corresponding position value stored in the edge. Swapping and deleting can be implemented in $O(l')$ work and $(\log n)$ span, and hence all operations cost $O(1)$ amortized work per edge and $O(\log n)$ span. □

### 6.3.2 Augmented Euler-Tour Trees

The parallel batch-dynamic Euler-tour trees used in our algorithm augment each node in the tree with two values indicating the number of tree and non-tree edges whose level is equal to the level of the forest currently stored in that subtree. The augmentation is necessary for efficiently fetching the tree edges that need to be pushed down before searching the data structure, and for fetching a subset of non-tree edges in a tree. We extend the data structure with operations which enable efficiently retrieving, removing and pushing down batches of tree or non-tree edges.

These primitives are all similar and can be implemented as follows. We first describe the primitives which fetch and remove a set of $l$ tree (or non-tree) edges. The algorithm starts by finding a set of vertices containing $l$ edges. To do this we perform a binary search on the skip-list in order to find the first node that has augmented value greater than $l$. The idea is to sequentially walk at the highest level, summing the augmented values of nodes we encounter and marking them, until the first node that we hit whose augmented value makes the counter larger than $l$, or we return to $v$. In the former case, we descend a level using this node's downwards pointer, and repeat, until we reach a level 0 node. We also keep a counter, $c$, indicating the number of tree (non-tree) edges to take from the rightmost marked node at level 0. Otherwise, all nodes at the topmost level are marked. The last step is to find all descendants of marked nodes that have a non-zero number of tree (non-tree) edges, and return all tree (non-tree) edges incident on them. The only exception is the rightmost marked node, from which we only take $c$ many tree (non-tree) edges

Insertions are handled by first inserting the edges into the adjacency list data structure. We then update the augmented values in the Euler-tour tree using the primitive from Tseng et al. [174]. We now argue that these implementations achieves good work and span bounds.

**Lemma 30.** Given some vertex, $v$ in a parallel batch-dynamic Euler-tour tree, we can fetch the first $l$ tree (or non-tree) edges referenced by the augmented values in the tree in $O\left(l\log\left(1+\frac{n_c}{l}\right)\right)$ work and $O(\log n)$ span w.h.p. where $n_c$ is the number of vertices in the Euler-tour tree at the current level. Furthermore, removing the edges can be done in the same bounds.

*Proof.* Standard proofs about skip-lists shows that the number of nodes traversed in the binary search is $O(\log n)$ w.h.p. [151, 174]. We can fetch $l$ edges from each vertex's adjacency list data structure in $O(l)$ amortized work and $O(\log n)$ span by Lemma 29. The total work is therefore $O\left(l\log\left(1+\frac{n_c}{l}\right)\right)$ in expectation, and the span is $O(\log n)$ w.h.p. since the span of the adjacency list access is an additive increase of $O(\log n)$. Observe that removing the edges can be done in the same bounds since updating the augmented values after deleting the edges costs $O\left(l\log\left(1+\frac{n_c}{l}\right)\right)$ expected work. □

**Lemma 31.** Decreasing the level of $l$ tree (or non-tree) edges in a parallel batch-dynamic Euler-tour tree can be performed in $O\left(l\log\left(1+\frac{n_c}{l}\right)\right)$ expected work and $O(\log n)$ span w.h.p. where $n_c$ is the number of nodes in the Euler-tour tree at the current level.

*Proof.* The proof is identical to the proof of Lemma 30. The only difference is that the augmented values of the nodes that receive an edge must be updated after insertion which costs at most $O\left(l \log\left(1+\frac{n_c}{l}\right)\right)$ in expectation. Note that since the forest on the lower level is a subgraph of the tree at the current level, it has size at most $n_c$, proving the bounds. □

## 6.4 A Simple Parallel Algorithm

In this section, we give a simple parallel batch-dynamic connectivity algorithm based on the HDT algorithm. The underlying invariants maintained by our parallel algorithm are identical to the sequential HDT algorithm: we maintain $\log n$ levels of spanning forests subject to Invariants 1 and 2. The main challenge, and where our algorithm departs from the HDT algorithm is in how we search for replacement edges in parallel, and how we search multiple components in parallel. We show by a charging argument that this parallel algorithm is work-efficient with respect to the HDT algorithm—it performs $O(\log^2 n)$ amortized work per edge insertion or deletion. Furthermore, we show that the span of this algorithm is $O(\log^4 n)$. Although these bounds are subsumed by the improved parallel algorithm we describe in Section 6.5, the parallel algorithm in this section is useful to illustrate the main ideas.

### 6.4.1 Connectivity Queries

As in the sequential algorithm, a connectivity query can be answered by simply performing a query on $F_{\log n}$. Algorithm 6 gives pseudocode for the batch connectivity algorithm. The bound we achieve follows from the batch bounds on batch-dynamic Euler-tour trees [174].

**Algorithm 6** The batch query algorithm

```
1: procedure BatchQuery($\{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$)
2:     return $F_{\log n}$.BatchQuery($\{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$)
```

**Theorem 23.** A batch of $k$ connectivity queries can be processed in $O\left(k + k \log\left(1+\frac{n}{k}\right)\right)$ expected work and $O(\log n)$ span w.h.p.

*Proof.* A batch of $k$ connectivity queries reduces to a batch of at most $k$ distinct **BatchFindRep** queries on the Euler-tour tree, which costs $O\left(k \log\left(1+\frac{n}{k}\right)\right)$ expected work and $O(\log n)$ span w.h.p. [174]. The connectivity queries can then be answered by comparing the representatives for each query. □

### 6.4.2 Inserting Batches of Edges

To perform a batch insertion, we first determine a set of edges in the batch that increase the connectivity of the graph. To do so, we treat each current connected component of the graph as a vertex, and build a spanning forest of the edges being inserted over this contracted graph. The edges in the resulting spanning forest are then inserted into the topmost level.

**Algorithm 7** The batch insertion algorithm

```
1: procedure BatchInsert( $U = \{(u_1, v_1), \ldots, (u_k, v_k)\}$ )
2:     For all $e_i \in U$, set $l(e_i) \leftarrow \log n$ in parallel
3:     Update $A_{\log n}[u]$ for edges incident on $u$
4:     $R \leftarrow \{(F_{\log n}.\textsc{FindRepr}(u), F_{\log n}.\textsc{FindRepr}(u)) \mid (u, v) \in U\}$
5:     $T' \leftarrow \textsc{SpanningForest}(R)$
6:     $T \leftarrow$ edges in $U$ corresponding to $T'$
7:     Promote edges in $T$ to tree edges
8:     $F_{\log n}.\textsc{BatchInsert}(T)$
```

Algorithm 7 gives pseudocode for the batch insertion algorithm. We assume that the edges given as input in $U$ are not present in the graph. Each vertex $u$ that receives an updated edge inserts its edges into $A_{\log n}[u]$ (Line 3). This step can be implemented by first running a semisort to collect all edges incident on $u$.

The last step is to insert edges that increase the connectivity of the graph as tree edges (Lines 4–8). The algorithm starts by computing the representatives for each edge (Line 4). The output is an array of edges, $R$, which maps each original $(u, v)$ edge in $U$ to the pair $(\textsc{FindRepr}(u), \textsc{FindRepr}(v))$ (note that these calls can be batched using BatchFindRepr). Next, it computes a spanning forest over the tree edges (Line 5). Finally, the algorithm promotes the corresponding edges in $U$ to tree edges. This is done by updating the appropriate adjacency lists and inserting them into $F_{\log n}$ (Lines 7–8).

**Theorem 24.** A batch of $k$ edge insertions can be processed in $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\log n)$ span w.h.p.

*Proof.* Lines 2–3 cost $O(k)$ work and $O(\log k)$ span w.h.p. using our bounds for updating $A$ (see Lemma 29). The find representative queries (Line 4) can be implemented using a BatchFindRepr call on the Euler-tour tree, which costs $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\log n)$ span w.h.p [174]. Computing a spanning forest (Line 5) can be done in $O(k)$ expected work and $O(\log k)$ span w.h.p. using Gazit's connectivity algorithm [69]. Finally, updating the adjacency lists and inserting the spanning forest edges into $F_{\log n}$ (Lines 7–8) costs $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ expected work and $O(\log n)$ span w.h.p. □

## 6.4.3 Deleting Batches of Edges

As in the sequential HDT algorithm, searching for replacement edges after deleting a batch of tree edges is the most interesting part of our parallel algorithm. A natural idea for parallelizing the HDT algorithm is to simply scan all non-tree edges incident on each disconnected component in parallel. Although this approach has low span per level, it may examine a huge number of candidate edges, but only push down a few non-replacement edges. In general, it is unable to amortize the work performed checking all candidate edges at a level to the edges that experience level decreases. To amortize the work properly while also searching the edges in parallel we must perform a more careful exploration of the non-tree edges. Our approach is to use a *doubling* technique, in which we geometrically increase the number of non-tree edges explored as long as we have not yet found a replacement edge. We show that using the doubling technique, the work performed (and number of non-tree edges explored) is dominated by the work of the last phase, when we either find a replacement edge, or run out of non-tree edges. Our amortized work-bounds follow by a per-edge charging argument, as in the analysis of the HDT algorithm.

## The Deletion Algorithm

Algorithm 8 shows the pseudocode for our parallel batch deletion algorithm. As with the batch insertion algorithm, we assume that each edge is present in $U$ in both directions. Given a batch of $k$ edge deletions, the algorithm first deletes the given edges from their respective adjacency lists in parallel (Line 2). It then filters out the tree edges (Line 3) and deletes each tree edge $e$ from $F_i \dots, F_{\log n}$, where $i$ is the level of $e$ (Line 4). Next, it computes $C$, a set of *components* (representatives) from the deleted tree edges (Line 5). For each deleted tree edge, $e$, the algorithm includes the representatives of both endpoints in the forest at $l(e)$, which must be in different components as $e$ is a deleted tree edge. Finally, the algorithm loops over the levels, starting at the lowest level where a tree edge was deleted (Line 7), and calls ParallelLevelSearch at each level. Each ParallelLevelSearch takes $i$, the level to search, $C$, the current set of disconnected components, and $S$, an initially empty set of replacement edges that the algorithm discovers (Line 8).

**Algorithm 8** The batch deletion algorithm

```
1: procedure BatchDeletion($U = \{e_1, \dots, e_k\}$)
2:    Delete $e \in U$ from $A_0, \dots, A_{\log n}$
3:    $T \leftarrow \{e \in U \mid e \in F_{\log n}\}$                    // Tree edges to delete
4:    Delete $e \in T$ from $F_0, \dots, F_{\log n}$
5:    $C \leftarrow \cup_{e=(u,v) \in T}(F_{l(e)}.\text{FindRepr}(u), F_{l(e)}.\text{FindRepr}(v))$
6:    $S \leftarrow \emptyset$
7:    for $i \in [min_l \leftarrow \min_{e \in T}, \log n]$ do
8:       $(C, S) \leftarrow$ ParallelLevelSearch($i, C, S$)
```

## Searching a Level in Parallel

The bulk of the work done by the deletion algorithm is performed by Algorithm 9, which implements a subroutine that searches the disconnected components at a given level of the data structure in parallel. The input to ParallelLevelSearch is an integer $i$, the level to search, a set of representatives of the disconnected components, $L$, and the set of replacement spanning forest edges that were found in levels lower than $i$, $S$. The output of ParallelLevelSearch is the set of components that are still disconnected after considering

**Algorithm 9** The parallel level search algorithm

```
 1: procedure ComponentSearch($i, c$)
 2:     $w \leftarrow 1,\ w_{\max} \leftarrow c$.NumNonTreeEdges
 3:     while $w \le w_{\max}$ do
 4:         $w \leftarrow \min(w, w_{\max})$
 5:         $E_c \leftarrow$ First $w$ non-tree edges in $c$
 6:         Push all non-replacement edges in $E_c$ to level $i-1$
 7:         if $E_c$ contains a replacement edge then
 8:             return $\{r\}$, where $r$ is any replacement edge in $E_c$
 9:         $w \leftarrow 2w$
10:     return $\emptyset$
11: procedure ParallelLevelSearch($i,\ L = \{c_1, c_2, \ldots\},\ S$)
12:     $F_i$.BatchInsert($S$)
13:     $C \leftarrow c \in L$ with size $\le 2^{i-1}$
14:     $D \leftarrow c \in L$ with size $> 2^{i-1}$
15:     while $|C| > 0$ do
16:         Push level $i$ tree edges of components in $C$ to level $i-1$
17:         $R \leftarrow \cup_{c \in C}$ ComponentSearch($i, c$)                // In parallel
18:         $R' \leftarrow \{(F_i.\text{FindRepr}(u), F_i.\text{FindRepr}(v)) \mid (u, v) \in R\}$
19:         $T' \leftarrow$ SpanningForest($R'$)
20:         $T \leftarrow$ Edges in $R$ corresponding to edges in $T'$
21:         Promote edges in $T$ to tree edges
22:         $F_i$.BatchInsert(T)
23:         $S \leftarrow S \cup T$
24:         $C \leftarrow \{F_i.\text{Repr}(c) \mid c \in C\}$
25:         $Q \leftarrow \{c \in C$ with no non-tree edges, or size $> 2^{i-1}\}$
26:         $D \leftarrow D \cup Q$
27:         $C \leftarrow C \setminus Q$
28:     return $(D, S)$
```

the non-tree edges at this level, and the set of replacement spanning forest edges found so far.

PARALLELLEVELSEARCH starts by inserting the new spanning forest edges in $S$ into $F_i$ (Line 12). Next, it computes $C$ and $D$, which are the components that are active and inactive at this level, respectively (Lines 13–14). The main loop of the algorithm (Lines 15–27) operates in a number of *rounds*.

Each round first pushes down all tree edges at level $i$ of every active component. It then finds a single replacement edge incident to each active component, searching the active components in parallel, pushing any non-replacement edge to level $i-1$. It then promotes a maximal acyclic subset of the replacement edges found in this round to tree edges, and proceeds to the next round. The rounds terminate once all components at this level are deactivated by either becoming too large to search at this level, or because the algorithm finished examining all non-tree edges incident to the component at this level.

The main loop (Lines 15–27) works as follows. The algorithm first pushes any level $i$ tree edges in an active component down to level $i-1$. The active components in $C$ have size at most $2^{i-1}$, meaning that any tree edges they have at level $i$ can be pushed to level $i-1$ (Line 16) without violating Invariant 1. Next, the algorithm searches each active component for a replacement edge in parallel by calling the COMPONENTSEARCH procedure in parallel over all components (Line 17). This procedure either returns an empty set if there are no replacement edges incident to the component, or a set containing a single replacement edge. Next, the algorithm maps the replacement edge endpoints to their current component's representatives by calling FINDREPR on each endpoint (Line 18). It then computes a spanning forest over these replacement edges (Line 19) and maps the edges included in the spanning forest back to their original endpoints ids (Line 20). Observe that the edges in $T$ constitute a maximal acyclic subset of replacement edges of $R$ in $F_i$. The algorithm therefore promotes the edges in $T$ to tree edges (Lines 21– 22). Note that the new tree edges are not immediately inserted into all higher level spanning trees. Instead, the edges are buffered by adding them to $S$ (Line 23) so that they will be inserted when the higher level is reached in the search. Finally, the algorithm updates the set of components by computing their representatives on the updated $F_i$ (Line 24), and filtering out any components which have no remaining non-tree edges, or become larger than $2^{i-1}$ (i.e., become unsearchable) into $D$ (Lines 25–27).

We now describe the COMPONENTSEARCH procedure (Lines 1–10). The search consists of a number of *phases*, where the $i$'th phase searches the first $2^i$ non-tree edges, or all of the non-tree edges if $2^i$ is larger than the number of non-tree edges in $c$. The search terminates either once a replacement edge incident to $c$ is found (Line 7), or once the algorithm unsuccessfuly examines all non-tree edges incident to $c$ (Line 3). Initially $w$, the search size, is set to 1 (Line 4). On each phase, the algorithm retrieves the first $w$ many non-tree edges, $E_c$ (Line 5). It pushes all non-tree edges that are not replacements to level $i-1$ (Line 6). It then checks whether any of the edges in $E_c$ are a replacement edge, and if so, returns one of the replacement edges in $E_c$ (Line 8). Note that checking whether an edge is a replacement edge is done using BATCHFINDREPR. Otherwise, if no replacement edge was found it doubles $w$ (Line 9) and continues.

### 6.4.4 Cost Bounds

The following lemmas are useful for analyzing the work bounds of our parallel algorithms.

**Lemma 32.** Let $n_1, n_2, \ldots, n_c$ and $k_1, k_2, \ldots, k_c$ be sequences of non-negative integers such that $\sum k_i = k$, and $\sum n_i = n$. Then

$$\sum_{i=1}^{c} k_i \log\left(1 + \frac{n_i}{k_i}\right) \leq k \log\left(1 + \frac{n}{k}\right).$$

*Proof.* We proceed by induction on $c$. When $c = 1$, the quantities are equal. For $c > 1$, we can write

$$\begin{aligned}\sum_{i=1}^{c} k_i \log\left(1 + \frac{n_i}{k_i}\right) &= \sum_{i=1}^{c-1} k_i \log\left(1 + \frac{n_i}{k_i}\right) + k_c \log\left(1 + \frac{n_c}{k_c}\right), \\ &\leq (k - k_c) \log\left(1 + \frac{n - n_c}{k - k_c}\right) + k_c \log\left(1 + \frac{n_c}{k_c}\right).\end{aligned}$$

Then, using the concavity of the logarithm function, we have

$$\begin{aligned}\sum_{i=1}^{c} k_i \log\left(1 + \frac{n_i}{k_i}\right) &\leq k \log\left(\frac{k - k_c}{k}\left(1 + \frac{n - n_c}{k - k_c}\right) + \frac{k_c}{k}\left(1 + \frac{n_c}{k_c}\right)\right), \\ &= k \log\left(\frac{k - k_c}{k} + \frac{n - n_c}{k} + \frac{k_c}{k} + \frac{n_c}{k}\right), \\ &= k \log\left(1 + \frac{n}{k}\right),\end{aligned}$$

which concludes the proof. □

**Lemma 33.** For any non-negative integers $n$ and $r$,

$$\sum_{w=0}^{r} 2^w \log\left(1 + \frac{n}{2^w}\right) = O\left(2^r \log\left(1 + \frac{n}{2^r}\right)\right).$$

*Proof.* First, write

$$\begin{aligned}\log\left(1 + \frac{n}{2^w}\right) &= \log\left(1 + 2^{r-w} \frac{n}{2^r}\right), \\ &\leq \log\left(2^{r-w}\left(1 + \frac{n}{2^r}\right)\right), \\ &= \log(2^{r-w}) + \log\left(1 + \frac{n}{2^r}\right), \\ &= (r - w) + \log\left(1 + \frac{n}{2^r}\right).\end{aligned}$$

Now substitute this into the sum to obtain

$$\begin{aligned}\sum_{w=0}^{r} 2^w \log\left(1+\frac{n}{2^w}\right) &\le \sum_{w=0}^{r}(r-w)2^w + \log\left(1+\frac{n}{2^r}\right)\sum_{w=0}^{r} 2^w,\\ &= \sum_{w=0}^{r}(r-w)2^w + O\left(2^r \log\left(1+\frac{n}{2^r}\right)\right),\end{aligned}$$

We evaluate the remaining sum by writing

$$\sum_{w=0}^{r}(r-w)2^w = \sum_{w=0}^{r}\frac{r-w}{2^{r-w}}2^r,$$

and then use the fact that

$$\sum_{w=0}^{r}\frac{r-w}{2^{r-w}} = O(1)$$

to conclude that

$$\begin{aligned}\sum_{w=0}^{r} 2^w \log\left(1+\frac{n}{2^w}\right) &= O(2^r) + O\left(2^r \log\left(1+\frac{n}{2^r}\right)\right),\\ &= O\left(2^r \log\left(1+\frac{n}{2^r}\right)\right),\end{aligned}$$

as desired. □

**Lemma 34.** For any $n \ge 1$, the function $x \log\left(1+\frac{n}{x}\right)$ is strictly increasing with respect to $x$ for $x \ge 1$.

*Proof.* The derivative of the function with respect to $x$ is

$$\log\left(1+\frac{n}{x}\right) - \frac{n}{n+x}.$$

We must show that this quantity is strictly positive for all $x \ge 1$. First, we use a well-known inequality that states

$$a^y \le 1+(a-1)y,$$

for $a \ge 1$ and $y \in [0,1]$. Using $a=2$ and $y = n/(n+x)$, we obtain

$$2^{\frac{n}{n+x}} \le 1 + \frac{n}{n+x}.$$

Since $n \ge 1$ and $x \ge 1$, we have

$$1+\frac{n}{n+x} < 1+\frac{n}{x},$$

and hence by transitivity,

$$2^{\frac{n}{n+x}} < 1+\frac{n}{x}.$$

Taking logarithms on both sides yields

$$\frac{n}{n+x} < \log\left(1+\frac{n}{x}\right),$$

which implies the desired result. □

We can now prove that our parallel algorithm has low span, and is work-efficient with respect to the sequential HDT algorithm. For simplicity, we assume that we start with no edges in a graph on $n$ vertices.

**Theorem 25.** A batch of $k$ edge deletions can be processed in $O(\log^4 n)$ span w.h.p.

*Proof.* The algorithm doubles the number of edges searched in each phase. Therefore, after $\log m = O(\log n)$ phases, all non-tree edges incident on the component will be searched.

In every round, each active component is either deactivated, or has a replacement edge found. In the worst case, the edges found for each active component pair the components off, leaving us with half as many active components in the subsequent round. As we lose a constant fraction of the active components per round, the algorithm takes $O(\log n)$ rounds.

A given level can therefore perform at most $O(\log^2 n)$ phases. Each phase consists of fetching, examining, and pushing down non-tree edges, and hence can be implemented in $O(\log n)$ span w.h.p. by Lemma 30 and Lemma 31. Therefore, the overall span for a given level is $O(\log^3 n)$ w.h.p. As all $\log n$ levels will be processed in the worst case, the overall span of the algorithm is $O(\log^4 n)$ w.h.p. □

We now analyze the work performed by the algorithm.

**Lemma 35.** The work performed by the BatchDeletion algorithm excluding the calls to ParallelLevelSearch is

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

in expectation.

*Proof.* The edge deletions performed by Line 2 cost $O(k)$ work by Lemma 29. Filtering the tree edges (Line 3) can be done in $O(k)$ work. Deleting the tree edges costs $O\left(k \log(1+n/k)\right)$ work by Lemma 34 (Line 4).

Line 5 perform a FindRepr call for each endpoint of each deleted tree edge. These calls can be implemented as a single BatchFindRepr call which costs $O\left(k \log(1+n/k)\right)$ work in expectation [174]. Since in the worst case each tree edge must be deleted from $\log n$ levels, the overall cost of this step is $O\left(k \log n \log(1+n/k)\right)$ in expectation. Summing up the costs for each level proves the lemma. □

**Theorem 26.** The expected amortized cost per edge insertion or deletion is $O(\log^2 n)$.

*Proof.* Algorithm 8 takes as input a batch of $k$ edge deletions. By Lemma 35, the expected work performed by BATCHDELETION excluding the calls to PARALLELLEVELSEARCH is

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

which is at most $O(k\log^2 n)$ in expectation. We now consider the cost of the calls to PARALLELLEVELSEARCH. Specifically, we show that the work performed during the calls to PARALLELLEVELSEARCH can either be charged to level decreases on edges, or is at most $O(k\log n)$ per call in expectation. Since the total number of calls to PARALLELLEVELSEARCH is at most $\log n$, the bounds follow.

First, observe that the number of spanning forest edges we discover, $|S|$, is at most $k$, since at most $k$ tree edges were deleted initially. Therefore, the batch insertion on Line 12 costs $O(k\log n)$ in expectation [174]. Similarly, $L$, the number of components that are supplied to PARALLELLEVELSEARCH, is at most $k$. Therefore, the cost of filtering the components in $L$ based on their size, and checking whether their representative exists in $F_i$ is at most $O(k\log n)$ in expectation (Lines 13–14).

To fetch, examine, and push down $l$ tree or non-tree edges costs

$$O\left(l \log\left(1+\frac{n}{l}\right)\right),$$

work in expectation, by Lemma 30, and Lemma 31. Note that this is at most $O(\log n)$ per edge. In particular, the cost of retrieving and pushing the tree edges of active components to level $i-1$ (Line 6) is therefore at most $O(\log n)$ per edge in expectation, which we charge to the corresponding level decreases.

We now show that all work done while searching for replacement edges (Lines 15–27) can be charged to level decreases. Consider an active component, $c$ in some round. Suppose the algorithm performs $q > 0$ phases before either the component is exhausted (all incident non-tree edges have been checked), or a replacement edge is found. First consider the case where it finds a replacement edge. If $q = 1$, only a single edge was inspected, so then we charge the $\log n$ work for the round to the edge, which will become a tree edge. Otherwise, it performs $q-1$ phases which do not produce any replacement edge.

Since phase $w$ inspects $2^w$ edges, it costs $O(2^w \log n)$ work. The total work over all $q$ phases is therefore

$$\sum_{w=0}^{q} 2^w \log n = O(2^q \log n)$$

in expectation. However, since no replacement was found during the first $q-1$ phases, there are at least $2^{q-1} = O(2^q)$ edges that will be pushed down, so we can charge $O(\log n)$ work to each such edge to pay for this. In the other case, $q$ phases run without finding a replacement edge. In this case, all edges inspected are pushed down, and hence each assumes a cost of $O(\log n)$ in expectation.

Now, we argue that the work done while processing the replacement edges is $O(k \log n)$ in expectation over all rounds. Since $k$ edges were deleted, the algorithm discovers at most $k$ replacement edges. We charge the work in these steps to the replacement edges that we find. Let $k'$ be the number of replacement edges that we find. Filtering the edges, and computing a spanning forest all costs $O(k')$ work. Promoting the edges to tree edges (inserting them into $F_i$ and updating the adjacency lists) costs $O(k' \log n)$ work in expectation. Finally, updating the components costs $O(k' \log n)$ work in expectation, which we can charge to either the component, if it is removed from $C$ in this round, or to the replacement edge that it finds, which is promoted to a tree edge. Since the algorithm can find at most $k$ replacement edges, the cost per level is $O(k \log n)$ in expectation for these steps as necessary.

In total, on each level the algorithm performs $O(k \log n)$ expected work that is not charged to a level decrease. Summing over $\log n$ levels, this yields an amortized cost of $O(\log^2 n)$ expected work per edge deletion. Finally, since the level of an edge can decrease at most $\log n$ times, and an edge is charged $O(\log n)$ expected work each time its level is decreased, the expected amortized cost per edge insertion is $O(\log^2 n)$.

□

## 6.5 A Faster Parallel Algorithm

In this section we design an improved version of the parallel algorithm that performs less work than the algorithm from Section 6.4. Furthermore, the improved algorithm runs in $O(\log^3 n)$ span w.h.p., improving on Algorithm 9 which runs in $O(\log^4 n)$ span w.h.p.

The key ideas in the improved algorithm involve interleaving the replacement edge search phases with the spanning forest computation, and adding replacement edges to the spanning forest more lazily to obtain better batch sizes for Euler-tour tree operations.

### 6.5.1 The Interleaved Deletion Algorithm

Algorithm 10 is based on *interleaving* the phases of doubling that search for replacement edges with the spanning forest computation performed on the replacement edges. Recall that in Algorithm 9, the number of edges examined in each round *is reset*, and the doubling algorithm must therefore start with an initial search size of 1 on the next round. Because the doubling resets from round to round, the number of phases per round can be $O(\log n)$ in the worst case, making the total number of phases per level $O(\log^2 n)$. Instead, the interleaved algorithm avoids resetting the search size by maintaining a *single*, geometrically increasing search size over all rounds of the search.

The second important difference in Algorithm 10 compared with Algorithm 9 is that it *defers* inserting tree edges found on this level until the end of the search. Instead, it continues to search for replacement edges from the *initial* components until the component is deactivated. This property is important to show that the work done for a component across all rounds is dominated by the cost of the last round, since the number of vertices in the component is fixed, but the number of non-tree edges examined doubles in each round. For

**Algorithm 10** The interleaved level search algorithm

```
1: procedure ComponentSearch($i, c, s$)
2:     $w_{\max} \leftarrow c$.NumNonTreeEdges
3:     $w \leftarrow \min(s, w_{\max})$
4:     $E_c \leftarrow$ First $w$ non-tree edges in $c$
5:     return {All replacement edges in $E_c$}
6: procedure PushEdges($i, c, s, M$)
7:     $w_{\max} \leftarrow c$.NumNonTreeEdges
8:     $w \leftarrow \min(s, w_{\max})$
9:     $E_c \leftarrow$ {First $w$ non-tree edges in $c$}
10:    if $M[c]$.size $\leq 2^{i-1}$ and $w < w_{\max}$ then
11:        Remove edges in $E_c$ from level $i$
12:        return $E_c$
13:    return $\emptyset$
14: procedure InterleavedLevelSearch($i$, $L = \{c_1, c_2, \ldots\}$, $S$)
15:    $F_i$.BatchInsert($S$)
16:    $C \leftarrow c \in L$ with size $\leq 2^{i-1}$
17:    $D \leftarrow c \in L$ with size $> 2^{i-1}$
18:    Push level $i$ tree edges of all components in $C$ to level $i-1$
19:    $r \leftarrow 0,\ T \leftarrow \emptyset,\ E_P \leftarrow \emptyset$
20:    $M \leftarrow \{c \rightarrow c \mid c \in C\}$
21:    while $|C| > 0$ do
22:        $w \leftarrow 2^r$
23:        $R \leftarrow \cup_{c \in C}$ ComponentSearch($i, c, w$)                    // In parallel
24:        $R' \leftarrow \{(F_i.\text{FindRepr}(u), F_i.\text{FindRepr}(v)) \mid (u, v) \in R\}$
25:        $T'_r \leftarrow$ SpanningForest($R'$)
26:        $T_r \leftarrow$ Edges in $R$ corresponding to edges in $T'_r$
27:        $T \leftarrow T \cup T_r$
28:        Update $M$, the map of supercomponents and their sizes
29:        $E_P \leftarrow E_P \cup_{c \in C}$ PushEdges($i, c, w, M$)                 // In parallel
30:        $D_r \leftarrow \{c \in C$ with no non-tree edges, or size $> 2^{i-1}\}$
31:        $D \leftarrow D \cup D_r$
32:        $C \leftarrow C \setminus D_r$
33:        $r \leftarrow r + 1$
34:    Promote edges in $T$ not in $E_p$ to tree edges at level $i$
35:    $F_i$.BatchInsert($T$)
36:    Insert non-tree and tree edges in $E_P$ to level $i-1$
37:    return $(D, S \cup T)$
```

the same reason, it also defers inserting the pushed edges onto level $i-1$. We crucially use this property to obtain improved batch work bounds in Section 6.6.

Another difference in the modified algorithm is that if a component is still active after adding the replacement edges found in this round (i.e., the component on level $i$ still has size at most $2^{i-1}$), then *all* of the edges found in this round can be pushed to level $i-1$ without violating Invariant 1. Notice now that when pushing down edges, both the *tree and non-tree* edges that are found in this round are pushed. Pushing down all edges ensures that the algorithm performs enough level decreases to which to charge the work performed during the next round. The component deactivates either once it runs out of incident non-tree edges, or when it becomes too large. Since the algorithm defers adding the new tree edges found until the end of the level, it also maintains an auxiliary data structure that dynamically tracks the size of the resulting components as new edges are found.

## The Deletion Algorithm

We briefly describe the main differences between INTERLEAVEDLEVELSEARCH, the new level search procedure, and PARALLELLEVELSEARCH. The algorithm consists of a number of *rounds* (Lines 21–33). We use $r$ to track the round numbers, and we use $E_P$ to store the set of both tree and non-tree edges that will be pushed to level $i-1$ at the end of the search at this level (Line 19). $T$ stores the set of tree edges that have been selected, which will be added to the spanning forest at the end of the level. Lastly, we use $M$ to maintain a dynamic mapping from all the components in $L$ to a unique representative for their contracted supercomponent (initially itself), and the size of the contracted supercomponent.

In round $r$, the algorithm first retrieves the first $2^r$ (or fewer) edges of each the active components in parallel, and finds replacement edges. All replacement edges are added to the set $R$ (Line 23).

The algorithm then computes a spanning forest over the edges in $R$, and computes $T_r$, which are the original replacement edges in $R$ that were selected as spanning forest edges (Lines 25–27). The spanning forest computation returns, in addition to the tree edges, a mapping from the vertices in $R'$ to their connectivity label (Line 25), which can be used on Line 28 to efficiently update the representatives of all affected components and the sizes of the supercomponents.

The next step maps over the components in parallel again, calling PUSHEDGES on each active component, and checks whether the edges searched in this round can be (lazily) pushed to level $i-1$ (Line 29).[1] If a component is still active (its new size is small enough to still be searched, and the component still has some non-tree edges remaining) (Line 10), all of the searched edges are removed from the adjacency lists at level $i$ (Line 11) and are added to the set of edges that will be pushed to level $i-1$ at the end of the level (Lines 12 and 29). Note that this set of edges contains both replacement tree edges we discovered, and non-tree edges. The tree-edges can be pushed down to level $i-1$ because the component with the tree edges added has size $\leq 2^{i-1}$.

[1]Note that the set of edges retrieved by PUSHEDGES in Line 9 is assumed to be the same as the one in Line 4. This assumption is satisfied by using our FETCHEDGES primitive on a parallel batch-dynamic Euler-tour tree, and can be satisfied in general by associating the edges retrieved in COMPONENTSEARCH to be used in PUSHEDGES.

The end of the round (Lines 30–33) handles updating the set of components and incrementing the round number, as in Algorithm 9.

Finally, once all components are inactive, the tree edges found at this level that are not contained in $E_p$ are promoted (the tree edges added to $E_p$ have their level decreased to $i-1$) and inserted into $F_i$ (Lines 34–35), and all edges added to $E_P$ in Line 29 are pushed down to level $i-1$ (Line 36). Note that any tree-edges found in this set are promoted in level $i-1$ and added to $F_{i-1}$. The procedure returns the set of components and all replacement edges found at this level and levels below it (Line 37).

## 6.5.2 Cost Bounds

We start by showing that the span of Algorithm 10 is $O(\log^3 n)$.

**Lemma 36.** The number of rounds performed by Algorithm 10 is $O(\log n)$ and the span of each round is $O(\log n)$ w.h.p.. The span of the InterleavedLevelSearch is therefore $O(\log^2 n)$ w.h.p..

*Proof.* Each round of the algorithm increases the search size of a component by a factor of 2. Therefore, after $O(\log n)$ rounds, every non-tree edge incident on a component will be considered and the algorithm will terminate.

To argue the span bound, we consider the main steps performed during a round. Fetching, examining and removing the edges from level $i$ takes $O(\log n)$ span w.h.p. by Lemma 30b and Lemma 31. Computing a spanning forest on the replacement edges and filtering the components (at most $k$ replacement edges, or components) can be done in $O(\log k)$ span. The span per round is therefore $O(\log n)$ w.h.p. and the span of InterleavedLevelSearch is $O(\log^2 n)$ w.h.p. □

Combining Lemma 36 with the fact that there are $\log n$ levels gives the following theorem.

**Theorem 27.** A batch of $k$ edge deletions can be processed in $O(\log^3 n)$ span w.h.p.

We now consider the work performed by the algorithm. We start with a lemma showing that the search-size for a component increases geometrically until the round where the component is deactivated.

**Lemma 37.** Consider a component, $c$, that is active at the end of round $r-1$. If $c$ is not removed from $C$, then it examines $\geq 2^{r-1}$ edges that are pushed down to level $i-1$ at the end of the search.

*Proof.* We prove the contrapositive. Suppose that $< 2^{r-1}$ edges are pushed down in total by $c$ in the last round. Then, we will show that $c$ cannot be active in the next round (i.e., it is removed from $C$ in round $r-1$).

Notice that $c$ must be active at the start of round $r-1$. Consider the check on Line 10, which checks whether $w \leq 2^{r-1}$ and $w < w_{\max}$ on this round. Suppose for the same of contradiction that both conditions are true. Then, by the fact that $w < w_{\max}$, it must be the case that $w = 2^{r-1}$ by Line 8. If the condition is true, then on Line 11 the algorithm adds $2^{r-1}$ edges to be pushed to level $i-1$, contradicting our assumption that $< 2^{r-1}$ edges are pushed.

Therefore the check on Line 10 must be false, giving that either $w > 2^{i-1}$, or $w = w_{\max}$. This means that $c$ will be marked as inactive on Line 30, and then become deactivated on Line 32. Therefore, if $< 2^{r-1}$ edges are pushed down by $c$ in round $r-1$, $c$ is deactivated at the end of the round, concluding the proof.

□

**Lemma 38.** Consider the work done by some component $c$ over the course of INTERLEAVEDLEVELSEARCH at a given level. Let $R$ be the total number rounds that $c$ is active. Then, $c$ pushes down $p_c = 2^R - 1$ edges in total. Furthermore, the total cost of searching for and pushing down replacement edges performed by $c$ is

$$O\left(p_c \log\left(1 + \frac{n_c}{p_c}\right)\right)$$

in expectation, where $n_c$ is the number of vertices in $c$.

*Proof.* By Lemma 37, for each round $r < R$, $c$ adds $2^r$ edges to be pushed down. Summing over all rounds shows that the total number of edges added to be pushed down is $2^R - 1$. The cost of pushing down these edges at the end of the search at this level is exactly

$$O\left(p_c \log\left(1 + \frac{n_c}{p_c}\right)\right).$$

by Lemma 31, since the size of the tree that is affected is $n_c$. We now consider the cost of fetching and examining the edges over all rounds. The cost of fetching and examining $2^r$ edges is

$$O\left(2^r \log\left(1 + \frac{n_c}{2^r}\right)\right),$$

in expectation by Lemma 30. Summing over all rounds $r < R$, the work is

$$\sum_{r=1}^{R-1} O\left(2^r \log\left(1 + \frac{n_c}{2^r}\right)\right)$$

in expectation to fetch and examine edges in the first $R-1$ rounds, which is equal to

$$O\left(2^R \log\left(1 + \frac{n_c}{2^R}\right)\right),$$

by Lemma 33. Since on round $R$, the algorithm searches at most $2^R$ edges, the total cost of searching for replacement edges over all rounds is at most

$$O\left(2^R \log\left(1 + \frac{n_c}{2^R}\right)\right) = O\left(p_c \log\left(1 + \frac{n_c}{p_c}\right)\right).$$

□

**Lemma 39.** The cost of INTERLEAVEDLEVELSEARCH is at most

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

in expectation where $p$ is the total number of edges pushed down.

*Proof.* First consider Lines 2–5. Since we are deleting a batch of $k$ edges, we can find at most $k$ replacement edges to reconnect these components, so Line 2 performs $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ expected work [174]. Pushing $t$ spanning tree edges to the next level (Line 5) can be done in $O\left(t\log\left(\frac{n}{t}+1\right)\right)$ expected work by Lemmas 30, 31, and 32. Hence in total, Lines 2–5 perform at most $O\left(k\log\left(1+\frac{n}{k}\right)+t\log\left(1+\frac{n}{t}\right)\right)$ work in expectation.

Now, consider the cost of the steps which scan or update the components that are active in each round. On the first round, this cost is $O(k)$. In every subsequent round, $r$, by Lemma 37 each currently active component must have added $2^{r-1}$ edges to be pushed down on the previous round. Therefore, we can charge the $O(1)$ work per component performed in this round to these edge pushes.

Next, we analyze the work done while searching for and pushing replacement edges. Consider some component $c \in C$ that is searched on this level. By Lemma 38, the cost of searching for and pushing down the replacement edges incident on this component is

$$O\left(p_c\log\left(1+\frac{n_c}{p_c}\right)\right)$$

in expectation, where $n_c$ is the number of vertices in $c$ and $p_c$ is the total number of edges pushed down by $c$. The total work done over all components to search for replacement edges and push down both the original tree edges, and the edges in each round is therefore

$$O\left(t\log\left(1+\frac{n}{t}\right)+\sum_{c\in C}p_c\log\left(1+\frac{n_c}{p_c}\right)\right).$$

in expectation. Since $\sum n_c = n$, by Lemma 32 this costs

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

work in expectation, where $p = t+\sum p_c$ is the total number of edges pushed, including tree and non-tree edges. Therefore, the total cost is

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

in expectation. □

**Theorem 28.** The expected amortized cost per edge insertion or deletion is $O(\log^2 n)$.

*Proof.* The proof follows from the same argument as Theorem 26, by using Lemma 39. □

## 6.6 Analysis of Batching

We now show that by a more careful analysis, we can obtain a tighter bound on the amount of work performed by the interleaved algorithm. In particular, we show in this section that the algorithm performs

$$O\left(\log n \log\left(1+\frac{n}{\Delta}\right)\right)$$

amortized work per edge in expectation, where $\Delta$ is the average batch size of all batches of deletions. Therefore, if we process batches of deletions of size $O(n/\text{polylog}(n))$ on average, our algorithm performs $O(\log n \log\log n)$ expected amortized work per edge, rather than $O(\log^2 n)$. Furthermore, if we have batches of size $O(n)$, the cost is just $O(\log n)$ per edge.

At a high level, our proof formalizes the intuition that in the worst case, all edges are pushed down at every level, and that performing fewer deletion operations results in larger batches of pushes which take advantage of work bounds of the Euler-tour tree. Our proof crucially relies on the fact that although the deletion algorithm at a level can perform $O(\log n)$ Euler-tour tree operations per component, since the batch sizes are geometrically increasing, these operations have the cost of a single Euler-tour tree operation per component. Furthermore, Lemma 39 shows that the costs per component can be combined so that the total cost is equivalent to the cost of a single Euler-tour tree operation on all the vertices. Therefore, the number of deletion operations can be exactly related to the effective number of Euler-tour tree operations at a level. We relate the number of deletions to the average batch size, which lets us obtain a single unified bound for both insertions and deletions.

**Theorem 29.** Using the interleaved deletion algorithm, the amortized work performed by BATCHDELETION and BATCHINSERTION on a batch of $k$ edges is

$$O\left(k \log n \log\left(1+\frac{n}{\Delta}\right)\right),$$

in expectation where $\Delta$ is the average batch size of all batch deletions.

*Proof.* Batch insertions perform only $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work by Theorem 24, so we focus on the cost of deletion since it dominates. Consider the total amount of work performed by all batch deletion operations at any given point in the lifetime of the data structure. We will denote by $k_b$, the size of batch $b$, and by $p_{b,i}$, the number of edges pushed down on level $i$ during batch $b$. Combining Lemmas 35, and 39, the total work is bounded above by

$$O\left(\sum_{\text{batch } b}\sum_{\text{level } i} k_b \log\left(1+\frac{n}{k_b}\right) + p_{b,i}\log\left(1+\frac{n}{p_{b,i}}\right)\right).$$

We begin by analyzing the first term, which is paid for by the deletion algorithm. Let

$$K = \sum_{\text{batch } b} k_b$$

denote the total number of deleted edges. Applying Lemma 32, and using the fact that there are $\log n$ levels, we have

$$O\left(\sum_{\text{batch } b}\sum_{\text{level } i} k_b \log\left(1+\frac{n}{k_b}\right)\right) = O\left(K \log n \log\left(1+\frac{n\cdot d}{K}\right)\right),$$

where $d$ is the number of batches of deletions. Since $K/d = \Delta$, this is equal to

$$O\left(K \log n \log\left(1+\frac{n}{\Delta}\right)\right),$$

work in expectation. Each batch can therefore be charged a cost of $\log n \log(1+n/\Delta)$ per edge, and hence the amortized cost of batch deletion is

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

in expectation. The remainder of the cost, which comes entirely from searching for replacement edges, is charged to the insertions. Consider this cost and let

$$P = \sum_{\text{batch } b}\sum_{\text{level } i} p_{b,i}$$

denote the total such number of edge pushes. Since the total number of terms in the double sum is $d \log n$, Lemma 32 allows us to bound the total work of all pushes by

$$\sum_{\text{batch } b}\sum_{\text{level } i} p_{b,i} \log\left(1+\frac{n}{p_{b,i}}\right) = O\left(P \log\left(1+\frac{nd\log n}{P}\right)\right).$$

in expectation. Since every edge can only be pushed down once per level, we have

$$P \le m \log n,$$

where $m$ is the total number of edges ever inserted. Therefore by Lemma 34, the total work is at most

$$O\left(m \log n \log\left(1+\frac{nd\log n}{m\log n}\right)\right) = O\left(m \log n \log\left(1+\frac{nd}{m}\right)\right)$$

in expectation. Since $d = K/\Delta$, this is equal to

$$O\left(m \log n \log\left(1+\frac{nK}{m\Delta}\right)\right)$$

in expectation. Since each edge can be deleted only once, we have $K \le m$, and hence we obtain that the total work to push all tree edges down is at most

$$O\left(m \log n \log\left(1+\frac{n}{\Delta}\right)\right).$$

in expectation. We can therefore charge $O\big(\log n \log(1+n/\Delta)\big)$ per edge to each batch insertion. Since this dominates the cost of the insertion algorithm itself, the amortized cost of batch insertion is therefore

$$O\left(k \log n \log\left(1+\frac{n}{\Delta}\right)\right),$$

in expectation as desired, concluding the proof. □

## 6.7 Discussion

In this chapter, we presented a novel batch-dynamic algorithm for the connectivity problem. Our algorithm is always work-efficient with respect to the Holm, de Lichtenberg and Thorup dynamic connectivity algorithm, and is asymptotically faster than their algorithm when the average batch size is sufficiently large. A parallel implementation of our algorithm achieves $O(\log^3 n)$ span w.h.p., and is, to the best of our knowledge, the first parallel algorithm for the dynamic connectivity problem performing $O(T \operatorname{polylog}(n))$ total expected work, where $T$ is the total number of edge operations.

There are several natural questions to address in future work. For example, can the span of our algorithm be improved to $O(\log^2 n)$ without increasing the work, or can the work be improved to match the best sequential running time of $O(\log n (\log\log n)^2)$ per edge [103]? Similarly, can our ideas be extended to obtain worst-case bounds? It seems possible that ideas from our work could be used to give a parallel batch-dynamic Monte-Carlo connectivity algorithm based on the Kapron-King-Mountjoy algorithm [111].

In the sequential setting, dynamic connectivity is also used as a building block for efficient algorithms for dynamic 2-edge connectivity and biconnectivity. It could therefore also be interesting to consider whether our batch-dynamic algorithm could be used as an ingredient in efficient parallel batch-dynamic algorithms for those problems. Existing sequential 2-edge connectivity and biconnectivity algorithms require a dynamic tree data structure supporting path queries which are not supported by Euler-tour trees, however, our RC-trees could be used instead, which makes them a possible candidate for this line of work.

Since the publication of this work, our algorithm has been extended by Tseng et al. [175] who use it to implement a parallel batch-dynamic algorithm for MST. Their algorithm processes batches of $k$ updates in $O(k \log^6 n)$ amortized work and $O(\log^4 n)$ span w.h.p. It is the first ever parallel batch-dynamic algorithm for MST with polylogarithmic work per edge and polylogarithmic span. It is however, not as efficient as the best sequential algorithm for the problem, so it remains to see whether this bound could be improved.

# Chapter 7
# Parallel Batch-Incremental Minimum Spanning Trees

## 7.1 Introduction

MSTs have a long and interesting history. The problem of *dynamically* maintaining the MST under modifications to the underlying graph has been well studied. Spira and Pan [165] were the first to tackle the dynamic problem, and give an $O(n)$ sequential algorithm for vertex insertion that is based on Boruvka's algorithm. The first sublinear time algorithm for edge updates was given by Frederickson [63], who gave an $O(\sqrt{m})$ algorithm. A well-celebrated improvement to Frederickson's algorithm was given by Eppstein et. al [55], who introduced the *sparsification* technique to reduce the cost to $O(\sqrt{n})$. A great number of subsequent dynamic algorithms, including parallel ones, take advantage of Eppstein's sparsification. The sequential incremental MST problem, i.e., maintaining the MST subject to new edge insertions but no deletions, requires $O(\log(n))$ time per update using dynamic trees [15, 171] to find the heaviest weight edge on the cycle induced by the new edge and remove it. Holm et al. gave the first polylogarithmic time algorithm for fully dynamic MST [99], supporting updates in $O(\log^4(n))$ amortized time per operation, later improved by a $\log\log n$ factor [101] in expectation. No worst-case polylogarithmic time algorithm is known for the fully dynamic case. The dynamic MST problem has also been studied quite extensively in parallel, even in the batch-dynamic setting.

**Parallel single-update algorithms** Work by Pawagi and Ramakrishnan [143] gives a parallel algorithm for vertex insertion (with an arbitrary number of adjacent edges) and edge-weight updates in $O(\log(n))$ span but $O(n^2 \log(n))$ work. Varman and Doshi [178, 179] improve this to $O(n \log(n))$ work. Jung and Mehlhorn [110] give an algorithm for vertex insertion in $O(\log(n))$ span, and $O(n)$ work. While this bound is optimal for dense insertions, i.e. inserting a vertex adjacent to $\Theta(n)$ edges, it is inefficient for sparse graphs.

Tsin [176] extended the work of Pawagi and Ramakrishnan [143] to handle vertex deletions in the same time bounds, thus giving a fully vertex-dynamic parallel algorithm that runs in $O(n^2 \log(n))$ work and $O(\log(n))$ span. Das and Ferragina [45] give algorithms for inserting and deleting edges in $O(\log(\frac{m}{n})\log(n))$ span and $O(n^{2/3} \log(\frac{m}{n}))$ work. Subsequent improvements by Ferragina [59, 60], and Das and Ferragina [47] improve the span bound to $O(\log(n))$ with the same work bound. A recent result by Kopelowitz et al. [124] gives an algorithm that takes $O(\sqrt{n} \log(n))$ work and $O(\log(n))$ span.

**Parallel batch-dynamic algorithms** The above are all algorithms for single vertex or edge updates. To take better advantage of parallelism, some algorithms that process batch updates have been developed. Pawagi [141] gives an algorithm for batch vertex insertion that inserts $k$ vertices in $O(\log(n)\log(k))$ span and $O(nk\log(n)\log(k))$ work. Johnson and Metaxas [109] give an algorithm for the same problem with $O(\log(n)\log(k))$ span and $O(nk)$ work.

Pawagi and Kaser [142] were the first to give parallel batch-dynamic algorithms for fully-dynamic MSTs. For inserting $k$ independent vertices, inserting $k$ edges, or decreasing the cost of $k$ edges, their algorithms takes $O(\log(n)\log(k))$ span and $O(nk)$ work. Their algorithms for increasing the cost of or deleting $k$ edges, or deleting a set of vertices with total degree $k$ take $O(\log(n)+\log^2(k))$ span and $O\left(n^2\left(1+\frac{\log^2(k)}{\log(n)}\right)\right)$ work. Shen and Liang [159] give an algorithm that can insert $k$ edges, modify $k$ edges, or delete a vertex of degree $k$ in $O(\log(n)\log(k))$ span and $O(n^2)$ work. Ferragina and Luccio [61, 62] give algorithms for handling $k = O(n)$ edge insertions in $O(\log(n))$ span and $O(n\log\log\log(n)\log(m/n))$ work, and $k$ edge updates in $O(\log(n)\log(m/n))$ span and $O(kn\log\log\log(n)\log(m/n))$ work. Lastly, Das and Ferragina's algorithm [45] can be extended to the batch case to handle $k$ edge insertions in $O(k+\log(m/n)\log(n))$ span and $O(n^{2/3}(k+\log(m/n)))$ work.

For a thorough and well written survey on the techniques used in many of the above algorithms, see Das and Ferragina [46].

**Sliding window dynamic graphs** Dynamic graphs in the sliding window model were studied by Crouch et. al [44]. In the sliding window model, there is an infinite stream of edges $\langle e_1, e_2, ...\rangle$, and the goal of queries is to compute some property of the graph over the edges $\langle e_{t-L+1}, e_{t-L+2}, \ldots, e_t\rangle$, where $t$ is the current time and $L$ is the fixed length of the window. Crouch et. al showed that several problems, including $k$-connectivity, bipartiteness, sparsifiers, spanners, MSFs, and matchings, can be efficiently computed in this model. Several of these results used a data structure for incremental MSF as a key ingredient. All of these results assumed a sequential model of computation.

## Our contributions

In this chapter, we start by presenting a parallel data structure for the *batch-incremental MSF problem*. It is the first such data structure that achieves polylogarithmic work per edge insertion. The data structure is work efficient with respect to the fastest sequential single-update data structure, and even more efficient for large batch sizes, achieving optimal linear expected work [117] when inserting all edges as a batch.

We then use our batch-incremental MSF data structure to develop various data structures for graph problems in a batch variant of the sliding-window model. In the sliding-window model [48], one keeps a fixed-size window that supports adding new updates to the new side of the window and dropping them from the old side. Each insertion on the new side does a deletion of the oldest element on the old side. In general, this can be more difficult than pure incremental algorithms, but not as difficult as supporting arbitrary deletion in fully dynamic algorithms. This setup has become popular in modeling an infinite stream of data when there is only bounded memory, and a desire to "forget" old updates in favor of

| | Incremental (This work) | Sliding window (This work) | Fully dynamic |
|---|---|---|---|
| Connectivity | $O(k\alpha(n))^*$ | $O(k\log(1+n/k))^*$ | $O(k\log(n)\log(1+n/k))^{*,\dagger}$ |
| $k'$-certificate | $O(k'k\alpha(n))^*$ | $O(k'k\log(1+n/k))^*$ | - |
| Bipartiteness | $O(k\alpha(n))^*$ | $O(k\log(1+n/k))^*$ | - |
| Cycle-freeness | $O(k\alpha(n))^*$ | $O(k\log(1+n/k))^*$ | - |
| MSF | $O(k\log(1+n/k))^*$ | $O(\varepsilon^{-1}k\log(n)\log(1+n/k))^{*,\ddagger}$ | $O(kn\log^{(3)}(n)\log(m/n))$ |
| $\varepsilon$-sparsifier | $O(\varepsilon^{-2}k\log^4(n)\alpha(n))^*$ | $O(\varepsilon^{-2}k\log^4(n)\log(1+n/k))^*$ | - |

**Table 7.1: Work bounds for new and known parallel batch-dynamic graph algorithms in the incremental (insert-only), sliding window, and fully dynamic settings. All algorithms run in $O(\text{polylog}(n))$ span and use $O(n\ \text{polylog}(n))$ space. $k$ denotes the batch size of updates. Note that the algorithms in the sliding window model are also applicable to the incremental setting, by simply never moving the left endpoint of the window. For large batch sizes $k$, these algorithms sometimes achieve better bounds. Some bounds given are randomized ($*$), amortized ($\dagger$), or give $(1+\varepsilon)$-approximate solutions ($\ddagger$)**

newer ones. There have been many dozens, perhaps hundreds, of papers using the model in general. Crouch et. al. [44] have derived several algorithms for graph problems in this model. For graph algorithms, the goal is typically to use only $\tilde{O}(n)$ memory.

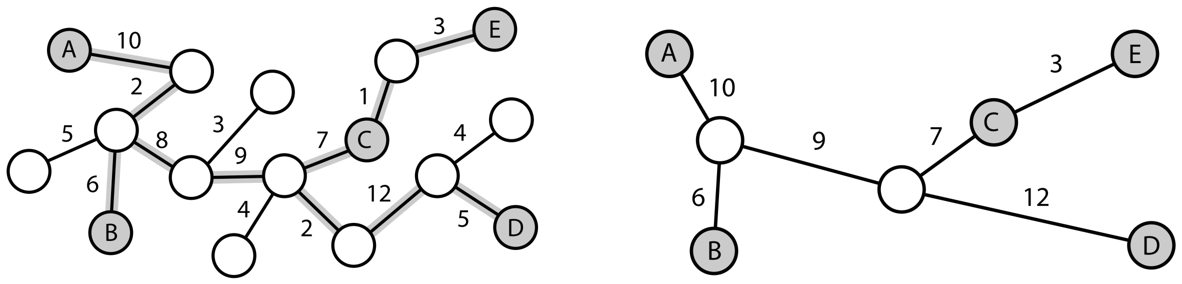
Here we extend the model to allow for rounds of batch (edge) insertions on the new side of the window, and batch (edge) deletions from the old side. Our results allow for arbitrary interleavings of batch insertions or deletions, and each of arbitrary size. Matching up equal sized inserts and deletes gives a fixed sized window, but we do not require this. Based on our batch-incremental MSF data structure, we are able to efficiently solve a variety of problems in the batch sliding-window model, including connectivity, $k$-certificate, bipartiteness, $(1+\epsilon)$-MSF, cycle-freeness, and sparsification. This uses an approach similar to the one of Crouch et. al. [44], which is based on sequential incremental MSF. In this work, other than using the batch-incremental MSF data structures, more work is required to augment their data structures in several ways.

Finally, we note that we can also apply these techniques to the incremental setting, and, using existing results on batch-incremental graph connectivity [162], obtain fast algorithms there as well. Table 7.1 gives specifics on the individual results and compares them to the existing bounds for parallel dynamic graphs in the incremental and fully dynamic settings.

## 7.2 The Compressed Path Tree

The key ingredient in our batch-incremental MSF data structure is a data structure for dynamically producing a *compressed path tree*. Given a weighted tree with some marked vertices, the compressed path tree with respect to the marked vertices is a minimal tree on the marked vertices and some additional "Steiner vertices" such that for every pair of marked vertices, the heaviest edge on the path between them is the same in the compressed tree as in the original tree. That is, the compressed path tree represents a summary of all possible pairwise heaviest edge queries on the marked vertices. An example of a compressed path tree is shown in Figure 7.1. More formally, consider the subgraph consisting of the union of the paths between every pair of marked vertices. The compressed path tree is precisely this subgraph but with all of the non-marked vertices of degree at most two spliced out.

This idea of compressing the input tree to one with no paths of degree two has been used before in various works under many different names. Komlós [123] and King [121] refer to such a tree as a “full branching tree”, and recently Gawrychowski et al. [68] called it a “topologically induced subtree”. Our main contribution is the efficient parallel construction of such a tree from a dynamic tree. To produce the compressed path tree, we leverage our parallel RC-Tree data structure from Chapter 3.



**(a) A weighted tree, with some important vertices marked (in gray). The paths between the marked vertices are highlighted.**

**(b) The corresponding compressed path tree. The edges are weighted to represent the heaviest edge on the corresponding path.**

**Figure 7.1: A weighted tree and its corresponding compressed path tree with respect to some marked vertices.**

### 7.2.1 A Parallel Algorithm for Compressed Path Trees

Given a balanced RC-Tree and a set of $k$ marked vertices, our algorithm produces the compressed path tree in $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work and $O(\log(n))$ span (in expectation and w.h.p. respectively if the RC-Tree is randomized).

Broadly, our algorithm for producing the compressed path tree works as follows. The algorithm begins by marking the clusters in the RC-Tree that contain a marked vertex, which is achieved by a simple bottom-up traversal of the tree. It then traverses the clusters of the RC-Tree in a recursive top-down manner. When the algorithm encounters a cluster that contains no marked vertices, instead of recursing further, it can simply generate a compressed representation of the contents immediately. The algorithm uses the following recursive subroutine, which, when called on the root cluster produces the answer.

- $\textsc{ExpandCluster}(C : \textbf{\textit{Cluster}}) : \textbf{\textit{Graph}}$

  Return the compressed path tree of the subgraph $C \cup \textsc{Boundary}(C)$, assuming that the boundary vertices of $C$ are marked in addition to the actual marked vertices.

We use the following primitives to interact with the RC-Tree. As the RC-Tree has bounded degree, each of them takes constant time.

- **BOUNDARY**($C$ : ***Cluster***) : ***vertex list***

  Given a cluster in the RC-Tree, return its boundary vertices.
- **CHILDREN**($C$ : ***Cluster***) : ***Cluster list***

  Given a cluster in the RC-Tree, return its child clusters.
- **REPRESENTATIVE**($C$ : ***Cluster***) : ***vertex***

  Given a non-leaf cluster in the RC-Tree, return its representative.
- **WEIGHT**($B$: ***BinaryCluster***) : ***number***

  Given a binary cluster in the RC-Tree, return the weight of the heaviest edge on the path between its two boundary vertices.

Lastly, we use the following primitives for constructing the resulting compressed path tree.

- **SPLICEOUT**($G$ : ***Graph***, $v$ : ***vertex***) : ***Graph***

  If $v$ has degree two in $G$ and is not marked, splice $v$ out by replacing its two incident edges with a contracted edge. The weight of the new edge is the heaviest of the two removed edges.
- **PRUNE**($G$ : ***Graph***, $v$ : ***vertex***) : ***Graph***

  If $v$ has degree two in $G$, return **SPLICEOUT**($G$). Otherwise, if $v$ has degree one in $G$, with neighbor $u$, and is not marked, remove $v$ and the edge $(u, v)$, and return **SPLICEOUT**($G'$, $u$), where $G'$ is the graph remaining after removing $v$ and $(u, v)$.

The intuition behind the **PRUNE** primitive is that without it, our algorithm could add redundant vertices to the compressed path tree. The proof of Lemma 40 illuminates the reason for the precise behavior of **PRUNE**. We give pseudocode for **EXPANDCLUSTER** in Algorithm 11. The compressed path tree of a marked tree is obtained by calling **EXPANDCLUSTER**(root), where root is the root cluster of the correspondingly marked RC-Tree. For a disconnected forest, simply call **EXPANDCLUSTER** on the root cluster of each component.

## 7.2.2 Analysis

### Correctness

We first argue that our algorithm for producing the compressed path tree is correct.

**Lemma 40.** Given a marked tree $T$ and its RC-Tree, for any cluster $C$, **EXPANDCLUSTER**($C$) returns the compressed path tree of the graph $C \cup$ **BOUNDARY**($C$), assuming the boundary vertices of $C$ are marked.

*Proof.* We proceed by structural induction on the clusters, with the inductive hypothesis that **EXPANDCLUSTER**($C$) returns the compressed path tree for the subgraph $C \cup$ **BOUNDARY**($C$), assuming that, in addition to the marked vertices of $T$, the boundary vertices of $C$ are marked. First, consider an unmarked cluster $C$.

**Algorithm 11** Compressed path tree algorithm

```
1: // Returns a graph $G$, which is represented by a pair of sets $(V, E)$, where $V$ is the vertex set and
   $E$ is a set of weighted edges. Edges are represented as pairs, the first element of which is the set of
   endpoints of the edge, and the second of which is the weight
2: procedure ExpandCluster($C$ : Cluster): Graph
3:   if not Marked($C$) then
4:     local $V \leftarrow$ Boundary($C$)
5:     if $C$ is a BinaryCluster then
6:       local $e \leftarrow (V,$ Weight$(C))$
7:       return $(V, \{e\})$
8:     else
9:       return $(V, \{\})$
10:  else if $C$ is a vertex $v$ then
11:    return $(\{v\}, \{\})$
12:  else
13:    local $G \leftarrow \bigcup_{c \in \text{Children}(C)}$ ExpandCluster$(c)$
14:    return Prune($G$, Representative($C$))
```

1. If $C$ is a ***NullaryCluster***, then it has no boundary vertices, and since no vertices are marked, the compressed path tree should be empty. Line 9 therefore returns the correct result.
2. If $C$ is a ***UnaryCluster***, then it has as single marked boundary vertex and no other marked vertices. Therefore the compressed path tree consists of the just the boundary vertex, so Line 9 returns the correct result.
3. If $C$ is a ***BinaryCluster***, the compressed path tree contains its endpoints, and an edge between them annotated with the weight of the corresponding heaviest edge in the original tree. Line 7 returns this.

Suppose $C$ is a leaf cluster. Since edges cannot be marked, it must be a base vertex cluster $v$. Since $v$ is marked, the compressed path tree just contains $v$ (returned by Line 11).

We now consider the inductive case, where $C$ is a marked cluster that is not a leaf of the RC-Tree. Recall the important facts that the boundary vertices of the children of $C$ consist precisely of the boundary vertices of $C$ and the representative of $C$, and that the disjoint union of the children of $C$ is $C$. Using these two facts and the inductive hypothesis, the graph $G$ (Line 13) is the compressed path tree of the graph $C \cup$ **Boundary**$(C)$, assuming that the boundary vertices of $C$ and the representative of $C$ are marked.

It remains to prove that the **Prune** procedure (Line 14) gives the correct result, i.e., it should produce the compressed path tree without the assumption that **Representative**$(C)$ is necessarily marked. Recall that the compressed path tree is characterized by having no unmarked vertices of degree less than three. If **Representative**$(C)$ is marked, or if **Representative**$(C)$ has degree at least three, then **Prune** does nothing, which is correct. Suppose **Representative**$(C)$ has degree two and is unmarked. **Prune** will splice out this vertex and combine its adjacent edges. Observe that splicing out a vertex does not change the degree of any other vertex in the tree. By the inductive hypothesis, no other vertex of $G$ (Line 13) was unmarked and had degree less than three, hence the result of Line 14 is the correct compressed path tree. Lastly, consider the case where **REPRESENTATIVE**$(C)$ has degree one and is not marked. **PRUNE** will correctly remove it from the tree, but this will change the degree of its neighboring vertex by one. If the neighbor was marked or had degree at least four, then it correctly remains in the tree. If the neighbor had degree three and was not marked, then it will now have degree two, and hence should be spliced out. As before, this does not change the degree of any other vertex in the tree, and hence is correct. By the inductive hypothesis, the neighbor cannot have had degree less than three and been unmarked before calling **PRUNE**. Therefore, in all cases, Line 14 returns the correct compressed path tree.

By induction on the clusters, the algorithm returns the compressed path tree of the graph $C \cup$ **BOUNDARY**$(C)$, assuming that the boundary vertices of $C$ are marked. □

**Theorem 30.** Given a marked tree $T$ and its RC-Tree, **EXPANDCLUSTER**(root), where root is the root of the RC-Tree, produces the compressed path tree of $T$ with respect to the marked vertices.

*Proof.* This follows from Lemma 40 and the fact that the root cluster is a nullary cluster and hence has no boundary vertices. □

## Efficiency

We now show that the compressed path tree can be computed efficiently.

**Lemma 41.** A compressed path tree for $k$ marked vertices has at most $O(k)$ vertices.

*Proof.* Since a compressed path tree has no non-marked leaves, it has at most $k$ leaves. Similarly, by definition, the compressed path tree has at most $k$ internal nodes of degree at most two. The result then follows from the fact that a tree with $k$ leaves and no internal nodes of degree less than two has $O(k)$ vertices. □

**Theorem 31.** Given a balanced (resp. in expectation) RC-Tree of a tree on $n$ vertices, producing the compressed path tree for $k$ marked vertices takes $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ work (resp. in expectation) and $O(\log(n))$ span (resp. w.h.p.).

*Proof.* The algorithm for producing the compressed path tree consists of two bottom-up traversals of the RC tree from the $k$ marked vertices to mark and unmark the clusters, and a top-down traversal of the same paths in the tree. Non-marked paths in the RC-Tree are only visited if their parent is marked, and since the RC tree has constant degree, work performed here can be charged to the parent. Also due to the constant degree of the RC-Tree, at each node during the traversal, the algorithm performs a constant number of recursive calls. Assuming that Lines 13 and 14 can be performed in constant time (to be shown), Theorem 8 implies the work bound of $O\left(k\log\left(1+\frac{n}{k}\right)\right)$ (in expectation if randomized).

To perform Line 13 in constant time, our algorithm can perform the set union of the vertex set lazily. That is, first run the algorithm to determine the sets of vertices generated by all of the base cases, and then flatten these into a single set by making another traversal of the tree. Duplicates can be avoided by noticing that the only duplicate in a union of clusters is the representative of their parent cluster. Line 14 can be performed by maintaining the edge set as an adjacency list. Since the underlying tree is always converted to a bounded-degree equivalent by the RC-Tree, the adjacency list can be modified in constant time.

The span bound follows from the fact that the RC-Tree has height $O(\log(n))$ (w.h.p. if randomized) and that each recursive call takes constant time.

Lastly, note that this argument also holds for disconnected graphs by simply traversing each component (i.e. each root cluster) in parallel after the marking phase. □

**Remark 2** (Building compressed path trees concurrently)**.** As described, since the algorithm for producing a compressed path tree marks the underlying RC-Tree, this method can not be used to construct multiple compressed path trees concurrently. Although our algorithm does not need this feature, we remark on it here since other applications may wish to take advantage, and indeed we will use this feature in Chapter 9. To support the ability to build multiple compressed path trees concurrently, we can instead use a local hashtable to remember the marked nodes. Since hashtable operations can be supported in $O(1)$ expected work and $O(\log(n))$ span w.h.p., this leaves the work of the algorithm unaffected, but increases the span to $O(\log^2(n))$ w.h.p.

## 7.3 Batch-Incremental Minimum Spanning Forest

Armed with the compressed path tree, our algorithm for batch-incremental MSF is a natural generalization of the standard sequential algorithm: Use a dynamic tree data structure [171] to find the heaviest edge on the cycle created by the newly inserted edge. By the classic "red rule," delete this edge to obtain the new MSF.

In the batch setting, when multiple new edges are added, many cycles may be formed, but the same idea still applies. Broadly, our algorithm takes the batch of edges and produces the compressed path trees with respect to all of their endpoints. The key observation here is that the compressed path trees will represent all of the possible paths between the new edge endpoints, and hence, all possible cycles that could be formed by their inclusion. Taking the compressed path tree and adding the newly inserted edges therefore results in a small graph that represents all possible cycles made by the new edges. To determine which edges should be added to the MSF, it is then a matter of computing the MSF of this representative graph, and taking the newly inserted edges that were selected. Conversely, the edges to be removed from the MSF are those corresponding to the compressed path tree edges that were not selected for the MSF of the representative graph.

We express the algorithm in pseudocode in Algorithm 12. It takes as input, an RC-Tree of the current MSF, and the new batch of edges to insert, and modifies the RC-Tree to represent the new MSF. The subroutine COMPRESSEDPATHTREES computes the compressed path trees for all components containing a marked vertex (in $K$) using Algorithm 11. We simplify the pseudocode by referring to edges in the compressed path trees and the corresponding edges in the MSF interchangably. That is, when we say to insert edges from the compressed path tree into the MSF, we really mean to insert the edges from $E^+$ whose heaviest weight that they correspond to, and similarly for deletion.

**Algorithm 12** Batch-incremental MSF

```
1: procedure BatchInsert(T : RCTree, $E^+$ : edge list)
2:     local $K \leftarrow \bigcup_{(\{u,v\},w)\in E^+} \{u,v\}$
3:     local $C \leftarrow$ CompressedPathTrees(RC, $K$)
4:     local $M \leftarrow$ MSF($C \cup E^+$)
5:     T.BatchDelete($E(C) \setminus E(M)$)
6:     T.BatchInsert($E(M) \cap E^+$)
```

### 7.3.1 Analysis

#### Correctness

We first argue that our algorithm for updating the MSF is correct. We will invoke a classic staple of MST algorithms and their analysis, the "cycle rule" (called the "red rule" by Tarjan).

**Lemma 42** (Red rule [171]). For any cycle $C$ in a graph, and a heaviest edge $e$ on that cycle, there exists a minimum spanning forest of $G$ not containing $e$.

**Theorem 32.** Let $G$ be a connected graph. Given a set of edges $E^+$, let $C$ be the compressed path tree of $G$ with respect to the endpoints of $E^+$, and let $M$ be the MST of $C \cup E^+$. Then a valid MST of $G \cup E^+$ is

$$M' = \text{MST}(G) \cup (E(M) \cap E^+) \setminus (E(C) \setminus E(M)),$$

where the edges of $C$ are identified with their corresponding heaviest edges in $G$ whose weight they are labeled with.

*Proof.* First, we use the fact that $A \cup (B \setminus C) = (A \cup B) \setminus C$ as long as $A$ and $C$ are disjoint. Then, by some simple Boolean algebra, since $E(M) \cap E^+ = E^+ \setminus (E^+ \setminus E(M))$, we have

$$M' = (\text{MST}(G) \cup E^+) \setminus (E(C) \setminus E(M)) \setminus (E^+ \setminus E(M)).$$

We will prove the result using the following strategy. We will begin with the graph MST$(G) \cup E^+$, and then show, using the red rule, that we can remove all of the edges in $E(C) \setminus E(M)$ and $E^+ \setminus E(M)$, such that the resulting graph is still a superset of an MST. We will then show that $M'$ has the same number of edges as an MST, and hence is in fact an MST.

Let $e = (u, v)$ be an edge in $E(C) \setminus E(M)$. We want to show that $e$ is a heaviest edge on a cycle in $C \cup E^+$. To do so, consider the cycle formed by inserting $e$ into $M$. If $e$ was not a heaviest edge on the cycle, then we could replace the heavier edge with $e$ in $M$ and reduce its weight, which would contradict $M$ being an MST. Therefore, $e$ is a heaviest edge on a cycle in $C \cup E^+$. Since every edge in $C$ represents a corresponding heaviest edge on a path in $G$, $e$ must also correspond to a heaviest edge on the corresponding cycle in $G \cup E^+$. Since $e$ is a heaviest edge on some cycle of $G \cup E^+$, the red rule says that it can be safely removed. Since we never remove an edge in $M$, the graph remains connected, and hence we can continue to apply this argument to remove every edge in $E(C) \setminus E(M)$, as desired.

The exact same argument also shows that we can remove all of the edges in $E^+ \setminus E(M)$, and hence, we can conclude that $M'$ is a superset of an MST. It remains to show, lastly, that $M'$ is an MST, i.e. contains no cycles. To do so, we will show that the algorithm removes the same number of edges that it inserts. First, since we assume that $G$ is connected, $|E(M)| = |E(C)| = |V(C)| - 1$. Then, since $E(C)$ and $E^+$ are disjoint, and $E(M) \subset E(C) \cup E^+$, simple Boolean algebra yields $|E(M) \cap E^+| = |E(C) \setminus E(M)|$, which shows that the algorithm inserts and removes the same number of edges. Therefore, since $M'$ is a superset of an MST and has the same number of edges as an MST, it must be an MST. $\square$

**Corollary 6.** Algorithm 12 correctly updates the MSF.

*Proof.* Theorem 32 shows that the algorithm is correct for connected graphs. For disconnected graphs, apply the same argument for each component, and observe that the previously disconnected components that become connected are connected by an MSF. $\square$

## Efficiency

We now show that the batch-incremental MSF algorithm achieves low work and span.

**Theorem 33.** Batch insertion of $k$ edges using Algorithm 12 takes $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work in expectation and $O(\log^2(n))$ span w.h.p.

*Proof.* Collecting the endpoints of the edges (Line 2) takes $O(k)$ work in expectation and $O(\log(k))$ span w.h.p. using a semisort [84]. By Theorem 31, Line 3 takes $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work in expectation and $O(\log(n))$ span w.h.p. By Lemma 41, the graph $C \cup E^+$ is of size $O(k)$, and hence by using the MSF algorithm of Cole et. al. [42], which runs in linear work in expectation and logarithmic span w.h.p., Line 4 takes $O(k)$ work in expectation and $O(\log(k))$ span w.h.p. Then, since $C \cup E^+$ is of size $O(k)$, the batch updates to the RC-Tree (Lines 5 and 6) take $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work in expectation and $O(\log^2(n))$ span w.h.p. Lastly, since $O(\log(k)) = O(\log(n))$, summing these up, we can conclude that Algorithm 12 takes $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$ work in expectation and $O(\log^2(n))$ span w.h.p. $\square$

## 7.4 Applications to the Sliding Window Model

We apply our batch-incremental MSF algorithm to efficiently solve a number of graph problems on a sliding window. For each problem, we present a data structure that implements the following operations to handle the arrival and departure of edges:

- **BatchInsert**($B$: ***edge list***): Insert the set of edges $B$ into the underlying graph.
- **BatchExpire**($\Delta$: ***int***): Delete the oldest $\Delta$ edges from the underlying graph.

Additionally, the data structure provides query operations specific to the problem. For example, the graph connectivity data structure offers an **Connected** query operation.

This formulation is a natural extension of the sequential sliding-window model. Traditionally, the sliding-window model [48] entails maintaining the most recent $W$ items, where $W$ is a fixed size. Hence, an explicit expiration operation is not necessary. More recently, there has been interest in maintaining variable-sized sliding windows. The interface used in this work allows for rounds of batch inserts (to accept new items) and batch expirations (to evict items from the old side). Notice that **BatchExpire** differs from a delete operation in dynamic algorithms in that it only expects a count, so the user does not need to know the actual items being expired to call this operation. Our results allow for arbitrary interleavings of batch insertions or expirations, and each of arbitrary size.

Small space is a hallmark of streaming algorithms. For insert-only streams, Sun and Woodruff [166] show a space lower-bound of $\Omega(n)$ words for connectivity, bipartiteness, MSF, and cycle-freeness, and $\Omega(kn)$ words for $k$-certificate assuming a word of size $O(\log n)$ bits. All our results below, which support not only edge insertions but also expirations, match these lower bounds except for MSF, which is within a logarithmic factor.

## 7.4.1 Graph Connectivity

We begin with the problem of sliding-window graph connectivity: to maintain a data structure so the users can quickly test whether a given pair of vertices can reach each other in the graph defined by the edges in the sliding window. We prove the following theorem:

**Theorem 34** (Connectivity)**.** For an $n$-vertex graph, there is a data structure, SW-Conn, that requires $O(n)$ words of space and supports the following:

- **BatchInsert**($B$) handles $k = |B|$ new edges in $O(1 + k\log(n/k))$ expected work and $O(\log^2 k)$ span w.h.p.
- **BatchExpire**($\Delta$) expires the $\Delta$ oldest edges in $O(1)$ worst-case work and span.
- **Connected**($u, v$) returns whether $u, v$ are connected in $O(\log n)$ work and span w.h.p.

Following Crouch et al. [44], we will prove this by reducing it to the problem of incremental minimum spanning tree. Let $\tau(e)$ be the index that edge $e$ appears in the whole stream. (The $i^{\text{th}}$ edge has index $i$.) Then, implicit in their paper is the following lemma:

**Lemma 43** (Recent Edge [44])**.** If $F$ is a minimum spanning forest (MSF) of the edges in the stream so far, where each edge $e$ carries a weight of $-\tau(e)$, then any pair of vertices $u$ and $v$ are connected if and only if (1) there is a path between $u$ and $v$ in $F$ and (2) the heaviest edge $e^*$ (i.e., the oldest edge) on this path satisfies $\tau(e^*) \geq T_W$, where $T_W$ is the $\tau(\cdot)$ of the oldest edge in the window.

*Proof of Theorem 34.* We maintain (i) an incremental MSF data structure from Theorem 5 and (ii) a variable $T_W$, which tracks the arrival time $\tau(\cdot)$ of the oldest edge in the window. The operation **BATCHINSERT**$(B)$ is handled by performing a batch insert of $k = |B|$ edges, where an edge $e \in B$ is assigned a weight of $-\tau(e)$. The operation **BATCHEXPIRE**$(\Delta)$ is handled by advancing $T_W$ by $\Delta$. The cost of these operations is clearly as claimed.

The query **CONNECTED**$(u, v)$ is answered by finding the heaviest edge on the path between $u$ and $v$ in the RC-Tree maintained and applying the conditions in the recent edge lemma (Lemma 43). The claimed cost bound follows because the MSF is maintained as an RC-Tree, which supports path queries in $O(\log n)$ time (Corollary 1). □

Often, applications depend on an operation **NUMCOMPONENTS()** that returns the number of connected components in the graph. It is unclear how to efficiently support this query using the above algorithm, which uses lazy deletion. Below is a variant, known as SW-Conn-Eager, which supports **NUMCOMPONENTS()** in $O(1)$ work.

The number of connected components can be computed from the number of edges in the minimum spanning forest (MSF) that uses only unexpired edges as # of components $=$ $n -$ # of MSF edges.

To this end, we modify SW-Conn to additionally keep a parallel ordered-set data structure $\mathscr{D}$, which stores all unexpired MSF edges ordered by $\tau(\cdot)$. This is maintained as follows: The **BATCHINSERT** operation causes some sets of edges to be added to and removed from the MSF (Algorithm 12, Lines 5-6). We can then adjust $\mathscr{D}$ using cost at most $O(n \log(n/t))$ work and $O(\log^2 n)$ span (e.g., [26, 27]). The **BATCHEXPIRE** operation applies SPLIT to find expired edges (costing $O(\log n)$ work and span) and explicitly deletes these edges from the MSF (costing expected $O(\Delta \log(n/\Delta))$ work and $O(\log^2 n)$ span w.h.p.). With these changes, **NUMCOMPONENTS()** is answered by returning $n - |\mathscr{D}|$ and SW-Conn-Eager has the following cost bounds:

**Theorem 35** (Connectivity With Component Counting)**.** For an $n$-vertex graph, there is a data structure, SW-Conn-Eager, that requires $O(n)$ space and supports the following:

- **BATCHINSERT**$(B)$ handles $k = |B|$ new edges in $O(1 + k \log(n/k))$ expected work and $O(\log^2 n)$ span w.h.p.
- **BATCHEXPIRE**$(\Delta)$ expires the $\Delta$ oldest edges in $O(\Delta \log(1 + n/\Delta) + \log n)$ expected work and $O(\log^2 n)$ span w.h.p.
- **CONNECTED**$(u, v)$ returns whether $u, v$ are connected in $O(\log n)$ work and span w.h.p.
- **NUMCOMPONENTS**() returns the number of connected components in $O(1)$ worst-case work and span.

### 7.4.2 Bipartiteness

To check bipartiteness, we apply a known reduction [14, 44]: a graph $G$ is bipartite if and only if its cycle double cover $D(G)$ has exactly twice as many connected components as $G$. A cycle double cover is a graph in which each vertex $v$ is replaced by two vertices $v_1$ and $v_2$, and each edge $(u, v)$, by two edges $(u_1, v_2)$ and $(u_2, v_1)$. Hence, $D(G)$ has twice as many vertices as $G$.

We can track the number of connected components of both the graph in the sliding window and its double cover by running two parallel instances of SW-Conn-Eager. Notice the edges of the cycle double cover $D(G)$ can be managed on the fly during **BatchInsert** and **BatchExpire**. Hence, we have the following:

**Theorem 36** (Bipartite Testing)**.** For an $n$-vertex graph, there is a data structure, SW-Bipartiteness, that requires $O(n)$ space and supports the following:

- **BatchInsert**$(B)$ handles $k = |B|$ new edges in $O(k \log(1+n/k))$ expected work and $O(\log^2 n)$ span w.h.p.
- **BatchExpire**$(\Delta)$ expires the $\Delta$ oldest edges in $O(\Delta \log(1+n/\Delta)+\log n)$ expected work and $O(\log^2 n)$ span w.h.p.
- **isBipartite**() returns a Boolean indicating whether the graph is bipartite in $O(1)$ worst-case work and span.

## 7.4.3 Approximate MSF Weight

For this problem, assume that the edge weights are between 1 and $n^{O(1)}$. Using known reductions [14, 33, 44], the weight of the MSF of $G$ can be approximated up to $1+\varepsilon$ by tracking the number of connected components in graphs $G_0, G_1, \ldots,$ where $G_i$ is a subgraph of $G$ containing all edges with weight *at most* $(1+\varepsilon)^i$. Specifically, the MSF weight is given by

$$(n - \mathsf{cc}(G_0)) + \sum_{i \geq 1} (\mathsf{cc}(G_{i-1}) - \mathsf{cc}(G_i))(1+\varepsilon)^i, \tag{7.1}$$

where $\mathsf{cc}(G)$ is the number of connected components in graph $G$.

Let $R = O(\varepsilon^{-1} \log n)$. We maintain $R$ instances of SW-Conn-Eager $F_1, \ldots, F_{R-1}$ corresponding to the connectivity of $G_0, G_1, \ldots, G_{R-1}$. The arrival of $k$ new edges involves batch-inserting into $R$ SW-Conn-Eager instances in parallel. Symmetrically, edge expiration is handled by batch-expiring edges in $R$ instances in parallel. Additionally, at the end of each update operation, we recompute equation (7.1), which involves $R$ terms and calls to **numComponents()**. This recomputation step requires $O(R)$ work and $O(\log R) = O(\log^2 n)$ span. Overall, we have the following:

**Theorem 37** (Approximate MSF)**.** Fix $\varepsilon > 0$. For an $n$-vertex graph, there is a data structure for approximate MSF weight that uses $O(\varepsilon^{-1} n \log n)$ space and supports:

- **BatchInsert**$(B)$ handles $k = |B|$ new edges in $O(\varepsilon^{-1} k \log n \log(1+n/k))$ expected work and $O(\log^2 n)$ span w.h.p.
- **BatchExpire**$(\Delta)$ expires the $\Delta$ oldest edges in $O(\varepsilon^{-1} \Delta \log n \log(1+n/\Delta))$ expected work and $O(\log^2 n)$ span w.h.p.
- **weight**() returns an $(1+\varepsilon)$-approximation to the weight of the MSF in $O(1)$ worst-case work and span.

### 7.4.4 $k$-Certificate and Graph $k$-Connectivity

For a graph $G$, a pair of vertices $u$ and $v$ are $k$-connected if there are $k$ edge-disjoint paths connecting $u$ and $v$. Extending this, a graph $G$ is $k$-connected if all pairs of vertices are $k$-connected. This generalizes the notion of connectivity, which is 1-connectivity. To maintain a "witness" for $k$-connectivity, we rely on a maximal spanning forest decomposition of order $k$, also known as a $k$-jungle, which decomposes $G$ into $k$ edge-disjoint spanning forests $F_1, F_2, \ldots, F_k$ such that $F_i$ is a *maximal* spanning forest of $G \setminus (F_1 \cup F_2 \cup \cdots \cup F_{i-1})$. This yields a number of useful properties, notably:

(P1) if $u$ and $v$ are connected in $F_i$, then they are at least $i$-connected;

(P2) $u$ and $v$ are $k$-connected in $F_1 \cup F_2 \cup \cdots \cup F_k$ iff. they are at least $k$-connected in $G$; and

(P3) $F_1 \cup F_2 \cup \cdots \cup F_k$ is $k$-connected iff. $G$ is at least $k$-connected.

Crouch et al. [44] show how to maintain such decomposition on a sliding window. When extended to the batch setting, the steps are depicted in Algorithm 13.

**Algorithm 13**

```
Let $O_0$ be the new batch of edges $B$
for $i = 1, \ldots, k$ do
    insert $O_{i-1}$ into $F_i$
    capture the edges being replaced as $F_i^-$ and the edges from $O_{i-1}$ that become part of $F_i$ as $F_i^+$
    set $O_i = F_i^- \cup (O_{i-1} \setminus F_i^+)$
```

Via known reductions [14, 44], we have that the $F_i$'s are maximal spanning forests and the unexpired edges of $F_1 \cup F_2 \cup \cdots \cup F_k$ form a $k$-certificate in the sense of properties (P1)–(P3) above. Additionally, this preserves all cuts of size at most $k$. In the following results, note that we momentarily use $b$ for batch size rather than $k$ since we are talking about the $k$-certificate and $k$-connectivity problems.

**Theorem 38** (***k***-Certificate)**.** There is a data structure for $k$-certificate for an $n$-vertex graph that requires $O(kn)$ space and supports the following:

- **BatchInsert**$(B)$ handles $b = |B|$ new edges in $O(kb\log(1 + n/b))$ expected work and $O(k\log^2 n)$ span w.h.p.
- **BatchExpire**$(\Delta)$ expires the $\Delta$ oldest edges in $O(k\Delta\log(1 + n/\Delta))$ expected work and $O(\log^2 n)$ span w.h.p.
- **makeCert**() returns a $k$-certificate involving at most $k(n-1)$ edges in $O(kn)$ work and $O(\log n)$ span.

*Proof.* We maintain each $F_i$ using a batch incremental MSF data structure from Theorem 5. To allow eager eviction of expired edges, we keep for each $F_i$ a parallel ordered-set data structure (e.g., [26, 27]) $\mathscr{D}_i$, which stores all unexpired edges of $F_i$. The operation **BatchInsert** is handled by sequentially working on $i = 1, 2, \ldots, k$, where for each $i$, edges are bulk-inserted into the MSF data structure for $F_i$, propagating replaced edges to $F_{i+1}$. The ordered-set data structure $\mathscr{D}_i$ can be updated accordingly. Note that the size of the changes to $\mathscr{D}_i$ never exceeds $O(b)$. The operation **BATCHEXPIRE** involves expiring edges in all $\mathscr{D}_i$'s. Finally, the operation **MAKECERT** is supported by copying and returning $\cup_{i=1}^{k}\mathscr{D}_i$. Because each $F_i$ is a forest, it has at most $n-1$ edges, for a total of at most $k(n-1)$ edges across $k$ spanning forests. □

Testing whether a graph is $k$-connected appears to be difficult in the fully-dynamic setting. Sequentially, an algorithm with $O(n \log n)$ time per update is known [55]. By contrast, for the incremental setting, there is a recent algorithm with $\widetilde{O}(1)$ time per update [83]. In the sliding window model, as a corollary of Theorem 38, the $k$-certificate can be used to test $k$-connectivity via a parallel global min-cut algorithm (e.g., [71, 73]). Because there are $O(kn)$ edges, this takes $O(kn \log n + n \log^4 n)$ work and $O(\log^3 n)$ span [73].

### 7.4.5 Cycle-Freeness

To monitor whether a graph contains a cycle, we observe that a graph that has no cycles is a spanning forest. Hence, if $F_1$ is a maximal spanning forest of a graph $G$, then $G \setminus F_1$ must not have any edges provided that $G$ has no cycles. To this end, we use the data structure from Theorem 38 with $k = 2$, though we are not interested in making a certificate. To answer whether the graph has a cycle, we check to see if $F_2$ is empty, which can be done in $O(1)$ work and span. Hence, we have the following:

**Theorem 39** (Cycle-freeness)**.** For an $n$-vertex graph, there is a data structure for cycle-freeness that requires $O(n)$ space and supports the following:

- **BATCHINSERT**$(B)$ inserts $k = |B|$ new edges in $O(k \log(1 + n/k))$ expected work and $O(\log^2 n)$ span w.h.p.
- **BATCHEXPIRE**$(\Delta)$ expires the $\Delta$ oldest edges in $O(\Delta \log(1 + n/\Delta))$ expected work and $O(\log^2 n)$ span w.h.p.
- **HASCYCLE**() returns whether the graph has a cycle in $O(1)$ work and span.

### 7.4.6 Graph Sparsification

The graph sparsification problem is to maintain a small, space-bounded subgraph so as to, when queried, produce a sparsifier of the graph defined by the edges of the sliding window. An $\varepsilon$-sparsifier of a graph $G$ is a weighted graph on the same set of vertices that preserves all cuts of $G$ up to $1 \pm \varepsilon$ but has only about $O(n \cdot \text{polylog}(n))$ edges. Existing sparsification algorithms commonly rely on sampling each edge with probability inversely proportional to that edge's connectivity parameter. We use the following result:

**Theorem 40** (Fung et al. [66])**.** Given an undirected, unweighted graph $G$, let $c_e$ denote the edge connectivity of the edge $e$. If each edge $e$ is sampled independently with probability $p_e \geq \min\left(1, \frac{253}{c_e \varepsilon^2} \log^2 n\right)$ and assigned a weight of $1/p_e$, then w.h.p., the resulting graph is an $\varepsilon$-sparsifier of $G$.

In the context of streaming algorithms, implementing this has an important challenge: the algorithm has to decide whether to sample/keep an edge before that edge's connectivity is known. Our aim is to show that the techniques developed in this chapter enable maintaining an $\varepsilon$-sparsifier with $O(n \cdot \text{polylog}(n))$ edges in the batch-parallel sliding-window setting. To keep things simple, the bounds, as stated, are not optimized for polylog factors.

We support graph sparsification by combining and adapting existing techniques for fast streaming connectivity estimation [77] and sampling sufficiently many edges at geometric probability scales (e.g., [14, 44]).

The key result is as follows: For $i = 1, 2, \ldots, L = O(\log n)$ and $j = 1, 2, \ldots, K = O(\log n)$, let $G_i^{(j)}$ denote a subgraph of $G$, where each edge of $G$ is sampled independently with probability $1/2^i$ and $G_0^{(j)} = G$. Then, the *level* $L(u, v)$, defined to be the largest $i$ such that $u$ and $v$ are connected in $G_i^{(j)}$ for all $0 \le j \le K$, gives an estimate of $uv$ connectivity:

**Lemma 44** ([77]). W.h.p., for every edge $e$ of $G$, $\Theta(s_e/\log n) \le 2^{L(e)} \le 2c_e$, where $s_e$ denotes strong connectivity and $c_e$ denotes edge connectivity.

The same argument also gives $c_e \le \Theta(2^{L(e)} \log n)$ w.h.p. While we cannot explicitly store all these $G_i^{(j)}$'s, it suffices to store each $G_i^{(j)}$ as a SW-Conn data structure (Theorem 34), requiring a total of $O(K \cdot L \cdot n) = O(n \log^2 n)$ space.

When an edge $e$ is inserted, if the algorithm were able to determine that edge's connectivity, it would sample that edge with the right probability ($p_e$) and maintain exactly the edges in the sparsifier. The problem, however, is that connectivity can change until the query time. Hence, the algorithm has to decide how to sample/keep an edge without knowing its connectivity. To this end, we resort to a technique adapted from Ahn et al. [14]: Let $H_0$ be the graph defined by the edges of the sliding window and for $i = 1, 2, \ldots, L$, let $H_i \subseteq H_0$ be obtained by independently sampling each edge of $H_0$ with probability $1/2^i$. Intuitively, every edge is sampled at many probability scales upon arrival.

Storing all these $H_i$'s would require too much space. Instead, we argue that keeping each $H_i$ as $\mathcal{Q}_i$, where $\mathcal{Q}_i$ is a $k$-SW-Certificate data structure (Theorem 38) with $k = O(\frac{1}{\varepsilon^2} \log^3 n)$ is sufficient[1]. Maintaining these requires a total of $O(knL) = O(\varepsilon^{-2} n \log^4 n)$ space.

Ultimately, our algorithm simulates sampling an edge $e$ with probability $2^{-\lfloor \log_2 \widetilde{p}_e \rfloor}$, where

$$\widetilde{p}_e = \min\big(1, O(2^{-L(e)} \varepsilon^{-2} \log^2 n)\big),$$

which uses an estimate of $2^{L(e)}$ in place of $c_e$. It answers a **SPARSIFY** query as follows:

**Algorithm 14**

```
for $e \in \bigcup_{i=1}^{L} \mathcal{Q}_i$ do
    output $e$ in the sparsifier with weight $1/\widetilde{p}_e$ if $e$ appears in $\mathcal{Q}_{\beta(e)}$, where $\beta(e) = \lfloor \log_2 \widetilde{p}_e \rfloor$
```

We now show that the $\mathcal{Q}_i$'s retain sufficient edges.

[1]We remark that the $\mathcal{Q}_i$ instances themselves contain enough information to estimate $c_e$ for all edges, but we do not know how to do so efficiently.

**Lemma 45.** W.h.p., an edge $e$ that is sampled into $H_{\beta(e)}$ is retained in $\mathcal{Q}_{\beta(e)}$.

*Proof.* Consider an edge $e = \{u, v\}$. There are $c_e$ disjoint paths between $u$ and $v$. W.h.p., due to the fact that $c_e \le \Theta(2^{L(e)} \log n)$, the expected number of paths that stay connected in $H_{\beta(e)}$ is at most $2\widetilde{p}_e \cdot c_e \le O(\varepsilon^{-2} \log^3 n)$. By Chernoff bounds, it follows that w.h.p., $e$ has edge connectivity in $H_{\beta(e)}$ at most $k = O(\varepsilon^{-2} \log^3 n)$ for sufficiently large constant. Hence, $e$ is retained in $\mathcal{Q}_{\beta(e)}$ w.h.p. □

This means that at query time, w.h.p., every edge $e$ is sampled into the sparsifier with probability $2^{-\lfloor \log_2 \widetilde{p}_e \rfloor} \ge p_e$, so the resulting graph is an $\varepsilon$-sparsifier w.h.p. (Theorem 40). Moreover, the number of edges in the sparsifier is, in expectation, at most

$$\sum_{e \in E(G)} 2\widetilde{p}_e = O(\varepsilon^{-2} \log^3 n) \sum_{e \in E(G)} \tfrac{1}{s_e} = O(\varepsilon^{-2} n \log^3 n),$$

where we used Lemma 44 and the fact that $\sum_e 1/s_e \le n-1$ [20, 66].

All the ingredients developed so far are combined as follows: The algorithm maintains a SW-Conn data structure for each $G_i^{(j)}$ and a $k$-SW-Certificate $\mathcal{Q}_i$ for each $H_i$. The **BatchInsert** operation involves inserting the edges into $KL + L$ data structures and the same number of independent coin flips. The cost is dominated by the cost of inserting into the $\mathcal{Q}_i$'s, each of which takes $O(kk \log(1 + n/k))$ expected work and $O(k \log^2 n)$ span w.h.p. The **BatchExpire** operation involves invoking **BatchExpire** on all the data structures maintained; the dominant cost here is expiring edges in the $\mathcal{Q}_i$'s. Finally, the query operation **sparsify** involves considering the edges of $\bigcup_{i=1}^{L} \mathcal{Q}_i$ in parallel, each requiring a call to $L(e)$, which can be answered in $O(LK \log n) = O(\log^3 n)$ work and span. In total, this costs $O(nkL \log^3 n) = O(n \operatorname{polylog}(n))$ work and $O(\operatorname{polylog}(n))$ span. The following theorem summarizes our result for sparsification:

**Theorem 41** (Graph Sparsification)**.** For an $n$-vertex graph, there is a data structure for graph (cut) sparsification that requires $O(\varepsilon^{-2} n \log^4 n)$ space and supports the following:

- **BatchInsert**$(B)$ handles $k = |B|$ new edges in $O(\frac{1}{\varepsilon^2} k \log(1 + \frac{n}{k}) \log^4 n)$ expected work and $O(\varepsilon^{-2} \log^5 n)$ span w.h.p.
- **BatchExpire**$(\Delta)$ expires the $\Delta$ oldest edges in $O(\frac{1}{\varepsilon^2} \Delta \log(1 + \frac{n}{\Delta}) \log^4 n)$ expected work and $O(\log^2 n)$ span w.h.p.
- **sparsify**() returns an $\varepsilon$-sparsifier w.h.p. The sparsifier has $O(\varepsilon^{-2} n \log^3 n)$ edges and is produced in $O(n \operatorname{polylog}(n))$ work and $O(\operatorname{polylog}(n))$ span w.h.p.

### 7.4.7 Connection to Batch-Incremental Algorithms

All applications studied here were built on top of the connectivity data structures (Theorem 34). In the related batch incremental setting, an analog of Theorem 34 was given by Simsiri et al. [162], where **BatchInsert** takes $O(k\alpha(n))$ expected work and $O(\operatorname{polylog}(n))$ span, and **Connected** takes $O(\alpha(n))$ work and span.

With this result, we can derive an analog of Theorem 35 using the following ideas: (i) maintain a component count variable, which is decremented every time a union successfully joins two previously disconnected components; and (ii) maintain a list of inserted edges that make up the spanning forest. This can be implemented as follows: Simsiri et al. maintains a union-find data structure and handles batch insertion by first running a find on the endpoints of each inserted edge and determining the connected components using a spanning forest algorithm due to Gazit [69]. Notice that the edges that Gazit's algorithm returns are exactly the new edges for the spanning forest we seek to maintain and can simply be appended to the list. This yields an analog of Theorem 35, where **BatchInsert** still takes $O(k\alpha(n))$ expected work and $O(\text{polylog}(n))$ span, **Connected** takes $O(\alpha(n))$ work and span, and **numComponents** takes $O(1)$ work and span. Ultimately, this means that replacing Theorems 34 and 35 with their analogs in each application effectively replaces the $\log(1+n/k)$ factor in the work term with an $\alpha(n)$ term, leading to the cost bounds presented in Table 7.1.

## 7.5 Discussion

This chapter presented the first work-efficient parallel algorithm for batch-incremental MSF. The algorithm is even asymptotically faster than the sequential algorithm for sufficiently large batch sizes. A key ingredient was the construction of a compressed path tree—a tree that summarizes the heaviest edges on all pairwise paths between a set of marked vertices. We demonstrated the usefulness of our algorithm by applying it to a range of problems in a generalization of the sliding-window model.

It would be interesting to explore other applications of our batch-incremental MST algorithm, or possibly even the compressed path tree by itself. In fact, in Chapter 9, we will do exactly that. The compressed path tree will turn out to be useful again when we study the minimum cut problem.

We are, to the best of our knowledge, the first to tackle sliding window dynamic graph problems in the parallel setting. Investigating other algorithms in this setting could lead to a variety of new problems, tools, and solutions.

# Part III

# Parallel Minimum Cuts

# Chapter 8
# Batch-Dynamic Trees with Mixed Queries and Updates

## 8.1 Introduction

The batched mixed operation problem is to take an off-line sequence of mixed operations on a data structure, usually a mix of queries and updates, and process them as a batch. The primary reason for batch processing is to allow for parallelism on what would otherwise be a sequential execution of the operations. Note that there is a key difference here between a mixed batch of operations and the standard batch-dynamic algorithms that we studied all throughout Chapters 3–7. A mixed batch can consist of *both* updates and queries that are interleaved with each other, and each query must be answered exactly as if every update *preceding it* has been evaluated, but none of the updates after it have. In other words, the queries must all be answered as if the entire set of operations was executed sequentially in the exact order given. Our regular batch-dynamic algorithms only handle batches of entirely updates or batches of entirely read-only queries, so they can not handle this type of problem.

With this in mind, the applications of an algorithm that could execute such a mixed batch of operations should be clear. It would allow us to convert an entirely sequential algorithm that is not obviously parallelisable, because each of the operations depends on those before, automatically into a parallel algorithm. We are not the first to make this observation. This technique was employed by Geissmann and Gianinazzi [71] in the first near-linear work parallel algorithm for minimum cuts. They observed that Karger's near-linear time sequential algorithm [114] consists of a large number of operations performed on a sequential dynamic tree data structure, and realized that if this set of operations could be pre-computed and then evaluated all in parallel, that the algorithm would be highly parallel. Indeed, Karger himself may have almost realized this, since he at least conjectured that the runtime of the algorithm could be improved since *"We are not using the full power of dynamic trees (in particular, the tree on which we operate is static, and the sequence of operations is known in advance)"* [114], though he does not explicitly mention this as an avenue for parallelism.

Geissmann and Gianinazzi's algorithm is specifically designed to evaluate a batch of $k$ path-weight updates and queries on a static weighted rooted tree in $\Omega(k \log^2 n)$ work. This makes it work-inefficient compared to a sequential evaluation of the operations, so their algorithm incurs an $O(\log n)$ penalty in the work compared to Karger's sequential algorithm. Their algorithm works by computing a heavy-light decomposition [163], i.e., a decomposition of a tree into vertex disjoint paths such that each root-to-leaf path in the tree intersects at most $O(\log n)$ of the paths in the decomposition. These paths are then stored inside binary trees, which leads to the work of $O(\log^2 n)$ per operation since there are up to $O(\log n)$ intersecting paths and an $O(\log n)$-work tree operation must be performed for each of them in the worst case.

**Our contributions** In this chapter, we are interested in designing an algorithm that is *both* more efficient and more general than that of Geissmann and Gianinazzi's. Our goal is to support evaluating a batch of mixed tree operations not limited to just path-weight updates and path queries, but also subtree-based operations and potentially other kinds of tree operations. We are also aiming for work efficiency, so we'd like to evaluate a batch of $k$ operations in or close to $O(k \log n)$ work and low polylog $n$ span since sequential operations take $O(\log n)$ time. Our result in this chapter will achieve $O(k \log(kn))$ work and $O(\log n \log k)$ span. Since $k = O(\text{poly}\, n)$, we have $O(\log(kn)) = O(\log n)$ so this is good.

To do so, we are going to make use of our Parallel RC-Trees from Chapter 3. Unlike Geissmann and Gianinazzi's algorithm which requires two layers (a set of path decompositions stored inside balanced trees), our algorithm for evaluating mixed batches operates directly on the RC-Tree representing the underlying tree, making it more efficient by a $\Theta(\log n)$ factor. Additionally, by exploiting the cluster-based decomposition of RC-Trees rather than a path-based decomposition, our algorithm is not limited to path-based operations but can handle a much more diverse range of updates and queries including subtree operations and more. Specifically, our framework can handle any operations sets that we call *RC-simple* operation sets, defined in the next section.

## 8.2 RC-Simple Operation Sets

We use the term *operation-set* to refer to the set of operations that can be applied. We are interested in operations on trees, and our results apply to operation-sets that can be implemented on an RC-Tree in a particular way, defined as follows.

**Definition 13.** An implementation of an operation-set on trees is a *simple RC implementation* if it uses an RC-Tree and satisfies the following conditions.

1. The implementation maintains a value at every RC cluster that can be calculated in constant time from the values of the children of the cluster,
2. every query operation is implemented by traversing from a leaf to the root examining values at the visited clusters and their children taking contant time per value examined, and using constant space, and
3. every update operation involves updating the value of a leaf using an associative constant-time operation, and then reevaluating the values on each cluster on the path from the leaf to the root.

Note that every operation has an *associated leaf* (either an edge or vertex). Also note that setting (i.e., overwriting) a value is an associative operation (just return the second of the arguments). For simple RC implementations, all operations take time (work) proportional to the height of the RC-Tree since they only follow a path to the root taking constant time at each cluster. Although the simple RC restriction may seem contrived, most operations on trees studied in previous work [5, 16, 163] can be implemented in this form, including most path and subtree operations. To see why, recall the decomposition properties of RC-Trees, Theorems 9 and 10, which say that any path and any subtree in the underlying tree can be decomposed into a set of disjoint clusters that are all children of a common path in the RC-Tree. Path- and subtree-based operations therefore typically need just update or collect a contribution from each such cluster. We described the sequential algorithms for path and subtree queries that did just this.

In this chapter we will only be concerned with rooted trees, so we will employ rooted RC-Trees (see Section 3.2.5). In our pseudocode, we will use the following notation. For a cluster $x$: $x.v$ is the representative vertex, $x.t$ is the top subcluster, $x.b$ is the bottom subcluster, $x.U$ is a list of unary subclusters, and $x.p$ is the parent cluster. We will also refer to the binary subcluster of a unary cluster as its top cluster as its cluster path is technically above the representative vertex.

## Example: Subtree sums with weight updates

As an example, consider the following two operations on a rooted tree (the first an update, and the second a query):

- **ADDWEIGHT**$(v, w)$ : add weight $w$ to a vertex $v$
- **SUBTREESUM**$(v)$ : return the sum of the weights of all of vertices in the subtree rooted at $v$

We described the technique for implementing such a query in Section 3.4.4. In this section we will provide a more detailed implementation and show that it satisfies the property of being a simple RC implementation.

**Algorithm 15** The SUBTREESUM query.

```
1: procedure SUBTREESUM($v$ : vertex)
2:     $w \leftarrow 0$
3:     $x \leftarrow v$; $p \leftarrow x.p$
4:     while $p$ is a binary cluster do
5:         if $(x = p.t)$ or $(x = p.v)$ then
6:             $w \leftarrow w + p.b.w + p.v.w + \sum_{u \in p.U} u.w$
7:         $x \leftarrow p$; $p \leftarrow x.p$
8:     return $w + p.v.w + \sum_{u \in p.U} u.w$
```

These operations can use a simple RC implementation by keeping as the value of each cluster the sum of values of all its children. This satisfies the first condition since the sums take constant time. Single-edge clusters in the RC-Tree start with the initial weight of the edge, while single-vertex clusters start with zero weight. An ADDWEIGHT$(v, w)$ adds weight $w$ to the vertex $v$ (which is a leaf in the RC-Tree) and updates the sums up to the root cluster. This satisfies the third condition since addition is associative and takes constant time. The query can be implemented as in Algorithm 15, where $x.w$ is the weight stored on the cluster $x$. It starts at the leaf for $v$ and goes up the RC-Tree keeping track of the total weight underneath $v$.

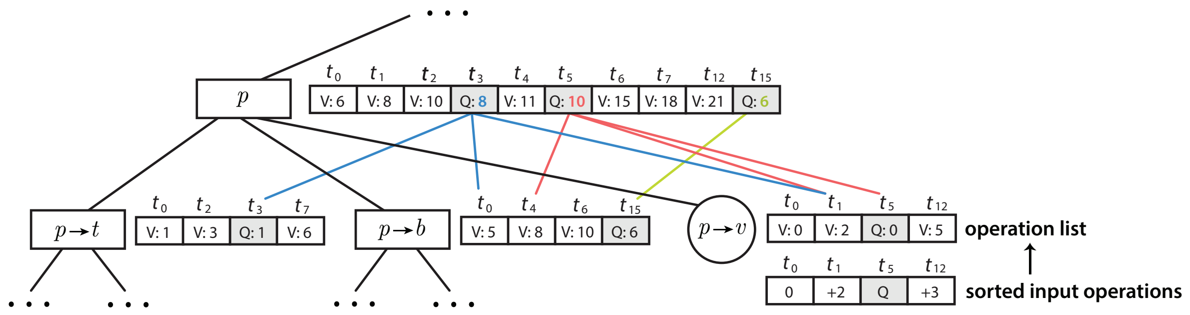

$p$
$t_0$ $t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$ $t_7$ $t_{12}$ $t_{15}$
V: 6 | V: 8 | V: 10 | Q: 8 | V: 11 | Q: 10 | V: 15 | V: 18 | V: 21 | Q: 6
$p \to t$
$t_0$ $t_2$ $t_3$ $t_7$
V: 1 | V: 3 | Q: 1 | V: 6
$p \to b$
$t_0$ $t_4$ $t_6$ $t_{15}$
V: 5 | V: 8 | V: 10 | Q: 6
$p \to v$
$t_0$ $t_1$ $t_5$ $t_{12}$
V: 0 | V: 2 | Q: 0 | V: 5
operation list
0 | +2 | Q | +3
sorted input operations

**Figure 8.1: Merging the operation lists for a binary cluster consisting of ADDWEIGHT and SUBTREESUM operations. Values in the operation sequence, denoted V : $v$, are computed by aggregating the latest values of the children at the given timestamp. For example, at $t_6$ in $p$, the algorithm adds 3 from $p.t$ at $t_2$, 10 from $p.b$ at $t_6$, and 2 from $p.v$ at $t_1$. Queries, denoted Q : $q$, are updated at each level by using the latest values of the children. For example, to update the query at $t_3$, it takes the current value of 1 from $p.t$ at $t_3$, then adds the weight of 5 from $p.b$ at $t_0$, and the weight of 2 from $p.v$ at $t_1$, as per Algorithm 15 since the conditional on Line 5 is true. Similarly, to update the query at $t_5$, the conditional is also true, and the most recent timestamps in $p.b$ and $p.v$ are $t_4$ and $t_1$, so it accumulates those values.**

Note that $x$ will never be a unary cluster, so if not the representative or top subcluster of $p$, it is the bottom subcluster with nothing below it in this cluster. Observe that SUBTREESUM only examines values on a path from the start vertex to the root and the children along that path. Each step takes constant time and requires constant space, satisfying the second condition. The operation-set therefore has a simple RC implementation.

## 8.3 Batched Mixed Operations Algorithm

We are interested in evaluating batches of operations from an operation-set on trees with a simple RC implementation. In particular, we prove the following theorem.

**Theorem 42** (Batched mixed operations)**.** Given a bounded-degree tree of size $n$ and a simple operation set, after $O(n)$ work and $O(\log n)$ span preprocessing, batches of $k$ operations from the operation-set can be evaluated in $O(k \log(kn))$ work and $O(\log n \log k)$ span. The total space required is $O(n + k_{max})$, where $k_{max}$ is the maximum batch size.

*Proof.* The preprocessing just builds an RC-Tree on the input tree, and sets the values for each cluster based on the initial values on the leaves. Note that our algorithm does not perform any structural modifications (links or cuts) to the RC-Tree, only augmented value updates, and hence the batch-dynamic algorithms of Chapters 4–5 are unnecessary. One could simply build a static RC-Tree as a byproduct of a static tree contraction, though the fully dynamic algorithms can of course still be used despite being overkill. Our algorithm for each batch is then implemented as follows:

1. Timestamp the operations by their order in the sequence.

2. Collect all operations by their associated leaf, and sort within each leaf by timestamp. This can be implemented with a single sort on the leaf identifier and timestamp.
3. For each leaf use a prefix sum on the update values to calculate the value of the leaf after each operation, starting from the initial value on the leaf.
4. Initialize each query using the value it received from the prefix sum. We now have a list of operations on each leaf sorted by timestamp. For each update we have its value, and for each query we also have its partial evaluation based on the value. We prepend the initial value to the list, and call this the *operation list.* An operation list is *non-trivial* if it has more than just the initial value.
5. For each level of the RC-Tree starting one above the deepest, and in parallel for every cluster on the level for which at least one child has a non-trivial operation list:
   (a) Merge the operation lists from each child into a single list sorted by timestamp.
   (b) Calculate for each element in the merged operations list, the latest value of each child at or before the timestamp. This can be implemented by prefix sums.
   (c) For each list element, calculate the value at that timestamp from the child values collected in the previous step.
   (d) For queries, use the values and/or child values to update the query.

This algorithm needs to have children with non-trivial operation lists identify parents that need to be processed. This can be implemented by keeping a list of all the clusters at a level with non-trivial operation lists left-to-right in level order. When moving up a level, clusters that share the same parent can be combined. An illustration of the merging process is depicted in Figure 8.1 using the operations from Algorithm 15.

We first consider why the algorithm is correct. We assume by structural induction (over subtrees) that the operation lists contain the correct values for each timestamped operation in the list. This is true at the leaves since we apply a prefix sum across the associative operation to calculate the value at each update. For internal clusters, assuming the child clusters have correct operation lists (values for each timestamp valid until the next timestamp, and partial result of queries), we properly determine the operation lists for the cluster. In particular for all timestamps that appear in children we promote them to the parent, and for each we calculate the value based on the current value, by timestamp, for each child.

We now consider the costs. The cost of the batch before processing the levels is dominated by the sort which takes $O(k \log k)$ work and $O(\log k)$ span. The cost at each level is then dominated by the merging and prefix sums which take $O(k)$ work and $O(\log k)$ span accumulated across all clusters that have a child with a non-trivial operation list. If the RC-Tree has height $O(\log n)$ then across all levels the cost is bounded by $O(k \log n)$ work and $O(\log n \log k)$ span. The total work and span is therefore as stated. The space for each batch of size $k$ is bounded by the size of the RC-Tree which is $O(n)$ and the total space of the operation lists at any two adjacent levels, which is $O(k)$. □

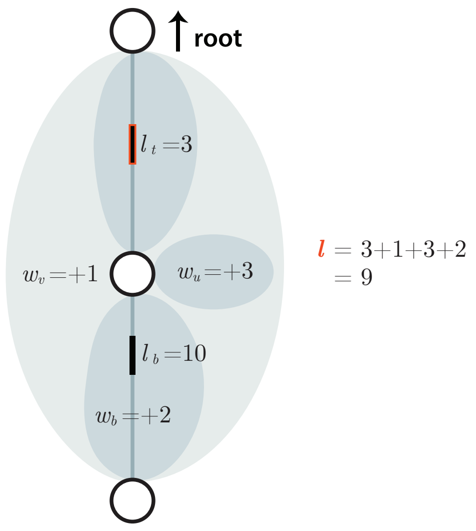


**Figure 8.2: When a binary cluster joins its children, all ADDPATHS' that originated in the vertex, bottom, or unary subclusters will affect all of the edges in the top cluster path. Here, $w' = w_v + w_b + w_u = 6$ weight is added to edges on the top cluster path due to ADDPATH operations from below. The minimum weight edge on the cluster path is therefore $\min(l_t + w', l_b) = \min(3+6, 10) = 9$, which is the edge from the top cluster path, highlighted in red.**

## 8.4 Path Updates and Path/Subtree Queries

We now consider implementing mixed operations consisting of updating paths, and querying both paths and subtrees. We will use these in Chapter 9 to implement an efficient algorithm for minimum cuts. In particular we wish to maintain, given a weighted rooted tree $T = (V, E)$, a data structure that supports the following operations.

- **ADDPATH**$(u, v, w)$: For $u, v \in V$ adds $w$ to the weight of all edges on the $u$ to $v$ path.
- **QUERYSUBTREE**$(v)$: Returns the lightest weight of an edge in the subtree rooted at $v \in V$,
- **QUERYPATH**$(u, v)$: For $u, v \in V$, returns the lightest weight of an edge on the $u$ to $v$ path.
- **QUERYEDGE**$(e)$: Returns $w(e)$

We saw the primary techniques needed to implement these in Sections 3.4.3 and 3.4.4. Here we will provide more detailed implementations and prove that they satisfy the necessary properties of being a simple RC implementation. To implement them, we first implement the simpler operations **ADDPATH**'$(v, w)$, which adds weight $w$ to the path from $v$ to the root; and **QUERYPATH**'$(u, v)$, which requires that $v$ be the representative vertex of an ancestor of $u$ in the RC-Tree. The more general forms can be implemented in terms of these with a constant number of calls given the lowest common ancestor (LCA) in the original tree for **ADDPATH** and in the RC-Tree for **QUERYPATH**.

**Lemma 46.** The **ADDPATH**', **QUERYSUBTREE**, **QUERYPATH**', and **QUERYEDGE** operations on bounded degree trees can be supported with a simple RC implementation.

**Algorithm 16** A simple RC implementation of **AddPath**'.

```
1: using VertexV = int
2: using UnaryV = struct { $m$ : edge, $w$ : int }
3: using BinaryV = struct { $m$ : edge, $l$ : edge, $w$ : int }
4: procedure $f_{\text{UNARY}}(w_v$ : VertexV, $(m_t, l_t, w_t)$ : BinaryV, $U$ : UnaryV list)
5:     $w' \leftarrow w_v + \sum_{u \in U} u.w$
6:     $m_u \leftarrow \min_{u \in U} u.m$
7:     return { $\min(m_t, l_t + w', m_u), w_t + w'$ }
8: procedure $f_{\text{BINARY}}(w_v$ : VertexV, $(m_t, l_t, w_t)$ : BinaryV, $(m_b, l_b, w_b)$ : BinaryV, $U$ : UnaryV list)
9:     $w' \leftarrow w_v + w_b + \sum_{u \in U} u.w$
10:    $m_u \leftarrow \min_{u \in U} u.m$
11:    return { $\min(m_t, m_b, m_u), \min(l_t + w', l_b), w_t + w'$ }
12: procedure AddPath'($v$ : vertex, $w$ : int)
13:    $v$.value $\leftarrow$ $v$.value + $w$
14:    Reevaluate the $f(\cdot)$ on path to root.
```

*Proof.* Our simple RC implementation for combining values and **AddPath**' is given in Algorithm 16. The queries are given in Algorithm 17. The value of each vertex (leaf) in the cluster is the total weight added to that vertex by **AddPath**'. The value for each unary cluster consists of: $m$, the minimum weight edge in the cluster; and $w$, the total weight of **AddPath**s' originating in the cluster. For each binary cluster we separate the minimum weights on and off the cluster path. In particular, the value of each binary cluster consists of: $m$, the minimum weight edge not on the cluster path; $l$, the minimum edge on the cluster path due to all **AddPath**' originating in the cluster; and $w$, the total weight of **AddPath**s' originating in the cluster. The $f_{\text{binary}}$ and $f_{\text{unary}}$ calculate the values for unary and binary clusters from the values of their children. We initialize each vertex with zero, and each edge $e$ with $(m = 0, l = w(e), w = 0)$.

It is a simple RC implementation since (1) the $f(\cdot)$ can be computed in constant time, (2) the queries just traverse from a leaf on a path to the root (possibly ending early) only examining child values, taking constant time per level and constant space, and (3) the update just sets a leaf using an associative addition, and reevaluates the values to the root.

We argue the implementation is correct. Firstly we argue by structural induction on the RC-Tree that the values as described in the previous paragraph are maintained correctly by $f_{\text{binary}}$ and $f_{\text{unary}}$. In particular assuming the children are correct we show the parent is correct. The values are correct for leaves since we increment the value on vertices with **AddPath**', and initialize the edges appropriately. To calculate the minimum edge weight of a unary cluster $f_{\text{unary}}$ takes the minimum of three quantities: the minimum off-path edge of the child binary cluster, the overall minimum edge of any of the child unary clusters, and, importantly, the minimum edge on the cluster path of the child binary cluster plus the **AddPath**' weight contributed by the unary clusters and the representative vertex (i.e., $\min(m_t, l_t + w', m_u)$). This is correct since all paths from those clusters to the root go through the cluster path, so it needs to be adjusted. The off-path edges and child unary clusters do not need to be adjusted since no path from the representative vertex goes through them. The minimum weight is therefore correct. The total **AddPath**' weight is correct since it just adds the contributions.

**Algorithm 17** An implementation of QUERYEDGE, QUERYPATH', and QUERYSUBTREE.

```
procedure QUERYSUBTREE($v$ : vertex)    // Returns the lightest weight of an edge in the subtree rooted at $v$
  $m \leftarrow \infty$; $l \leftarrow \infty$    // $m$: min edge not on the cluster path, $l$: min edge on the cluster path so far
  $x \leftarrow v$; $p \leftarrow x.p$
  while $p$ is a binary cluster do    // Accumulate weights until we reach a unary cluster
    if $(x = p.t)$ or $(x = p.v)$ then    // If $x$ is in the top half of the cluster, then all
      $w' \leftarrow p.b.w + p.v.w + \sum_{u \in p.U} u.w$    // AddPaths originating below it will add to
      $l \leftarrow \min(l + w', p.b.l)$    // the weight of all edges on the cluster path
      $m \leftarrow \min(m, p.b.m, \min_{u \in p.U} u.m)$
    $x \leftarrow p$; $p \leftarrow x.p$
  $w' \leftarrow p.v.w + \sum_{u \in p.U} u.w$
  return $\min(l + w', m, \min_{u \in p.U} u.m)$    // The lightest weight edge is either on the cluster path or not
procedure QUERYEDGE($e$ : edge)    // Returns the weight of the edge e
  $w \leftarrow w(e)$
  $x \leftarrow e$; $p \leftarrow x.p$
  while $p$ is a binary cluster do
    if $x = p.t$ then    // if $e$ is on the cluster path of the top half of the
      $w \leftarrow w + p.b.w + p.v.w + \sum_{u \in p.U} u.w$    // current cluster, then all AddPaths originating
    $x \leftarrow p$; $p \leftarrow x.p$    // below the top cluster will add to the weight of $e$
  return $w + p.v.w + \sum_{u \in p.U} u.w$
procedure QUERYPATH'($u$ : vertex, $v$ : vertex)    // Returns the lightest edge on the path from $u$ to $v$
  $m \leftarrow \infty$; $t \leftarrow \infty$; $b \leftarrow \infty$    // such that $v$ is the representative of an ancestor of $u$ in the RC-Tree
  $x \leftarrow u$; $p \leftarrow x.p$
  while not $p.v = v$ do
    $w' \leftarrow p.v.w + \sum_{u \in p.U} u.w$
    if $p$ is a unary cluster then    // When $p$ is a unary cluster, and $u$ originated in the
      if $x = p.t$ then $m \leftarrow \min(t + w', m)$    // top subcluster, the weight of all AddPaths below
      else $m \leftarrow \min(p.t.l + w', m)$    // is added to the edges between $u$ and the boundary,
      $t \leftarrow \infty$; $b \leftarrow \infty$    // otherwise it is added to all edges on the top cluster path
    else
      $w' \leftarrow w' + p.b.w$    // The weight of all AddPaths below is added to the top cluster path.
      if $x = p.t$ then $t \leftarrow t + w'$; $b \leftarrow \min(b + w', p.b.l)$    // If $u$ originated in the top subcluster, this
      else if $x = p.b$ then $t \leftarrow \min(p.t.l + w', t)$    // affects both $t$ and $b$. If $u$ originated in the
      else $t \leftarrow p.t.l + w'$; $b \leftarrow p.b.l$    // bottom subcluster, it affects only $t$. Otherwise, $u$
    $x \leftarrow p$; $p \leftarrow x.p$    // originated in a unary subcluster so $x$ is not in the cluster path.
  if $x = p.t$ then $l \leftarrow b$
  else if $x = p.b$ then $l \leftarrow t$    // $u$ either connects to $v$ in the direction of the top boundary of $p$ (a
  else return $m$    // weight of $t$), the bottom boundary of $p$ (a weight of $b$) or neither ($m$)
  while $p$ is a binary cluster do
    $w' \leftarrow p.v.w + p.b.w + \sum_{u \in p.U} u.w$    // There might still be more AddPath operations below, so we
    if $(x = p.t)$ then $l \leftarrow l + w'$    // continue up the RC-Tree to accumulate any that remain.
  return $\min(m, l)$
```

For binary clusters we need to separately consider the minimum off- and on-path edges. For the off-path edges the parts that are off the cluster path are the off-path edges from the two binary children, plus all edges from the unary children (i.e., min($m_t, m_b, m_u$)). For the on-path edges both the top and bottom binary clusters contribute their on-path edges. The on-path edges from the bottom binary cluster do not need to be adjusted because no vertices in the cluster are below them. The on-path edges from the top binary cluster need to be adjusted by the **ADDPATH**' weights from all vertices in the bottom cluster, all vertices in unary child clusters, and the representative vertex since they are all below the path (this sum is given by $w'$). See Figure 8.2. The minimum of the resulted adjusted top edge and bottom edge is then returned, which is indeed the minimum edge on the path accounting for **ADDPATH**s' on vertices in the cluster.

**QUERYSUBTREE**($v$) accumulates the appropriate minimum weights within a subtree as it goes up the RC-Tree. It starts at the node for which $v$ is its representative vertex. As with the calculation of values it needs to separate the on-path and off-path minimum weight. Whenever coming as the upper binary cluster to the parent, **QUERYSUBTREE** needs to add all the contributing **ADDPATH**' weights from vertices below it in the parent cluster (the representative vertex, the lower binary cluster and the unary clusters, see Figure 8.2) to the current minimum on-path weight. A minimum is then taken with the lower on-path minimum edge to calculate the new minimum on-path edge weight (Line 7). The off-path minimum is the minimum of the current off-path minimum, the minimum off-path edge of the bottom cluster and the minimums of the unary clusters (Line 8). Once we reach a unary cluster we are done since for a unary cluster all subtrees of vertices within the cluster are fully contained within the cluster. The final line therefore just determines the overal minimum for the subtree rooted at $v$ by considering the on-path edges adjusted by **ADDPATH**' contributions, the off-path edges, and all edges in child unary clusters.

**QUERYEDGE**($e$) simply adds the total weight of all **ADDPATH**' operations that occurred beneath $e$ to the weight of $e$. Specifically, at each iteration of the loop, $w$ contains the $w(e)$ plus the total weight of all **ADDPATH**' operations originating at any vertex below $e$ that is contained in the current cluster $x$. As the query moves up the RC-Tree, if the parent cluster is a binary cluster and $x$ is its top subcluster, then the vertices not yet accounted for are those in the bottom subcluster, the representative vertex, and the unary subclusters. If $x$ is the bottom subcluster of its binary parent, or one of its unary subclusters, then no vertices in $p$ but not $x$ are below $e$. When the while loop terminates, $p$ is a unary cluster and $x$ is its binary subcluster. At this point, the representative of $p$, and all unary subclusters of $p$ are below $e$, and hence their weight is added to the total. Since $p$ is a unary cluster, there exists no additional vertices below $e$ in the tree, and hence the final weight contains the contributions of all **ADDPATH**' operations originating below $e$.

Lastly, **QUERYPATH**' works by maintaining three values, $m, t, b$. To make defining them easier, consider, at each iteration of the main loop (Lines 23–34) in which the current cluster $x$ is a binary cluster, the vertex $c$ which is the closest vertex to $u$ on the cluster path of $x$ (if $u$ is on the cluster path of $x$, say $c = u$). Then, we can define $m$ as the minimum weight edge on the path from $u$ to $c$ (which will be $\infty$ if $u$ is on the cluster path of $x$), $t$ as the minimum weight edge above $c$ on the cluster path of $x$, and $b$ as the minimum weight edge below $c$ on the cluster path of $x$. If $x$ is a unary cluster, then $t$ and $b$ are $\infty$ (undefined), and

$m$ is simply the minimum weight edge on the path from $u$ to the boundary of $x$. Observe that it is important for the algorithm to maintain both $t$ and $b$ because it does not know in advance whether $v$ is above or below the current cluster path. It remains to argue that the implementation correctly maintains these values, and that the postprocessing is correct.

Each time the algorithm moves up to the next highest cluster, it first computes $w'$, the total weight of all **ADDPATH**' operations originating below the representative vertex. If the cluster is a unary cluster, and $u$ originated from the top (binary) subcluster, then the path from $u$ to the boundary of $p$ consists of the previous path from $u$ to $c$ (the lightest edge on which is $m$), and the path from $c$ to the boundary of $p$ (the lightest edge on which is $t$). Since $w'$ weight has been added to all edges on the path from $c$ to the boundary of $p$, the lightest such edge is now $t + w'$ and hence the lightest edge on the path from $u$ to the boundary of $p$ is $\min(t + w', m)$. If $u$ did not originate in the top subcluster of $p$, it came from one of the unary subclusters. In this case, the path from $u$ to the boundary of $p$ consists of the path from $u$ to the boundary of $x$, and the cluster path of the top subcluster (which begins at the boundary of $x$ and ends at the boundary of $p$), and hence the lightest edge is $\min(p.t.l + w', m)$. Since the current cluster is a unary cluster, $t$ and $b$ are undefined (Line 28).

If the next cluster is a binary cluster, we reason as follows. If $u$ originated in the top subcluster, then the path from $c$ to the top boundary remains the same, but $w'$ weight is added to every edge (including $t$). The cluster path below $c$ now consists of the edges previously below $c$ to the bottom boundary of $x$, and additionally those on the cluster path of the bottom subcluster (the edges from the bottom boundary of $x$ to the bottom boundary of $p$). The edges below $c$ on the cluster path of the top subcluster (including $b$) have had their weight increased by $w'$, and hence the lightest edge on the path from $c$ to the bottom boundary of $p$ is now $\min(b + w', p.b.l)$. Similarly, if $u$ originated in the bottom subcluster, then the path from $c$ to the bottom boundary hasn't changed, so $b$ is unchanged, and no weight is added to the edge $t$. However, since the path from $c$ to the top boundary of $p$ now includes the cluster path of the top subcluster, the lightest edge from $c$ to the top boundary is now $\min(p.t.l + w', t)$. Otherwise, $u$ must have originated from a unary subcluster of $p$, and hence the cluster path of $p$ contains no edges from $x$, so $t$ is simply the lightest edge in the top subcluster, and $b$ is the lightest edge in the bottom subcluster.

Once the main loop terminates (Lines 23–34), by the loop condition, it must be because the current cluster $x$ has $v$ as a boundary. If $u$ originated in the top subcluster of the latest $p$, then $v$ must be the bottom boundary of $p$, and hence the path from $u$ to $v$ consists of the path from $u$ to $c$ and the path from $c$ to $v$ which goes towards the bottom boundary of $p$ and hence contains $b$. Conversely, if $u$ originated in the bottom subcluster of $p$, then the path from $u$ to $v$ goes towards the top boundary of $p$ and hence contains $t$. If $u$ originated in a unary subcluster, then the path from $u$ to $v$ just joins $u$ to the boundary of $x$, hence the lightest edge is $m$. If not, the lightest edge is either $m$, or $b$ or $t$ respectively. The weight of $b$ or $t$ might still be affected by **ADDPATH**' operations from below, so the total weight of such operations is accumulated by continuing up the RC-Tree and added to the final weight. □

**Corollary 7.** Given a bounded-degree tree of size $n$, any sequence of $k$ **AddPath**, **QuerySubtree**, **QueryPath**, and **QueryEdge** operations can be evaluated in $O(n + k\log(nk))$ work, $O(\log n \log k)$ span and $O(n+k)$ space.

*Proof.* The LCAs required to convert **AddPath** to **AddPath**' and **QueryPath** to **QueryPath**' can be computed in $O(n+m)$ work, $O(\log n)$ span, and $O(n)$ space [157]. The rest follows from Theorem 42 and Lemma 46. □

## 8.5 Discussion

In this chapter, we designed a fast and general algorithm for evaluating mixed batches of updates and queries on rooted trees. It is more work efficient and more general than a previous algorithm of Geissmann and Gianinazzi which evaluates batches of path weight updates and queries. The motivation for solving this problem is to implement efficient parallel algorithms for the minimum cut problem. In Chapter 9, we will see several algorithms where this framework is a critical ingredient. We will use this framework to implement an algorithm for simulating Karger's random edge contraction algorithm [113] which results in a new faster parallel approximation algorithm for minimum cuts. Then we will use it to solve the 2-*respecting cut* problem, which, combined with the rest of Chapter 9 solves the minimum cut problem work efficiently in parallel.

Given the generality of our framework, it would be interesting to explore whether there are other sequential algorithms that it might be used to parallelize.

# Chapter 9
# Parallel Minimum Cuts

## 9.1 Introduction

The minimum cut problem has been studied by computer scientists and mathematicians for decades and is one of the core problems in the study of graph theory. Karger gave a number of algorithms for the problem based on the technique of random edge contractions [113, 116], including approximate algorithms, exact algorithms, sequential algorithms, and parallel algorithms. To contract an edge in this context means to merge the two adjacent vertices together to create a "supervertex" with the union of their neighbors. Karger observed that by randomly contracting edges of the graph until only two supervertices remain, you obtain a random cut of the graph, and by repeating this process sufficiently many times, you are likely to find the minimum cut.

The big breakthrough in minimum cuts however was Karger's near-linear time algorithm based on random sampling, tree packings, and 2-respecting cuts [114]. Given a weighted, undirected graph $G$ and a spanning tree $T$, a cut of $G$ $k$-respects $T$ if at most $k$ edges of $T$ cross the cut. Karger's near-linear time algorithm runs in $O(m \log^3 n)$ time and consists in: (1) Find $O(\log n)$ spanning trees of $G$ such that w.h.p., the minimum cut 2-respects at least one of them, then (2) find, for each of the aforementioned spanning trees, the minimum 2-respecting cut in $G$.

Karger solves the first step using a combination of random sampling and *tree packing*. Given a weighted graph $G$, a tree packing of $G$ is a set of weighted spanning trees of $G$ such that for each edge in $G$, its total weight in all of the spanning trees is no more than its weight in $G$. The underlying tree packing algorithms used by Karger have running time proportional to the size of the minimum cut, so random sampling is first used to produce a sparsified graph, or *skeleton*, where the minimum cut has size $\Theta(\log n)$ w.h.p. The sampling process is crafted such that the tree packing still has the desired 2-respecting property w.h.p.

To sample a skeleton graph such that its minimum cut is precisely size $\Theta(\log n)$, one must randomly sample edges with probability $\Theta(\log n / c)$, where $c$ is the size of the minimum cut. Of course, paradoxically, $c$ is precisely what we are trying to compute in the first place! Karger circumvents this issue by using an approximation algorithm to obtain an $O(1)$-approximation to $c$, which suffices for the desired skeleton. One can use Matula's algorithm [129] which runs in linear time on unweighted graphs, and can be extended to run in $O(m \log n \log W)$ time on weighted graphs, where $W$ is the total weight in the graph. By employing a transformation that removes high-weight edges from the graph without affecting the minimum cut too much, one can reduce $W$ to polynomially large in $n$ to obtain a runtime of $O(m \log^2 n)$ for this step.

Given the skeleton graph, Karger gives two algorithms for producing tree packings such that sampling $\Theta(\log n)$ trees from them guarantees that, w.h.p., the minimum cut 2-respects one of them. The first approach uses a tree packing algorithm of Gabow [67]. The second is based on the packing algorithm of Plotkin et al. [146], and is much more amenable to parallelism. It works by performing $O(\log^2 n)$ minimum spanning tree computations. In total, Step 1 of the algorithm takes $O(m + n\log^3 n)$ time.

For the second step, Karger develops an algorithm to find, given a graph $G$ and a spanning tree $T$, the minimum cut of $G$ that 2-respects $T$. The algorithm works by arbitrarily rooting the tree, and considering two cases: when the two cut edges are on the same root-to-leaf path, and when they are not. Both cases use a similar technique; They consider each edge $e$ in the tree and try to find the best matching $e'$ to minimize the weight of the cut induced by the edges $\{e, e'\}$. This is achieved by using a dynamic tree data structure to maintain, for each candidate $e'$, the value that the cut would have if $e'$ were selected as the second cutting edge, while iterating over the possibilities of $e$ and updating the dynamic tree. Karger shows that this step can be implemented sequentially in $O(m\log^2 n)$ time, which results in a total runtime of $O(m\log^3 n)$ when applied to the $O(\log n)$ spanning trees.

Geissmann and Gianinazzi [71] parallelize Karger's algorithm by implementing an algorithm for batched mixed path updates and path queries (see Chapter 8 and using it to evaluate the sequence of dynamic tree operations in parallel. Their resulting algorithms runs in $O(m\log^4 n)$ work and $O(\log^3 n)$ span, just a factor of $\Theta(\log n)$ slower than Karger's sequential algorithm while being highly parallel. Gawrychowski, Mozes, and Weimann [68] soon after give a breakthrough algorithm that is the first sequential improvement to Karger's algorithm in over 20 years by speeding up the algorithm for computing minimum 2-respecting cuts to just $O(m\log n)$, resulting in an overall runtime of $O(m\log^2 n)$.

**Our contribution** Our goal is to obtain the best of both worlds and design a parallel algorithm that is work efficient, i.e., matches the $O(m\log^2 n)$ work bound of Gawrychowski, Mozes, and Weimann, while also being highly parallel and running in polylogarithmic span. Our results are built upon numerous components that are novel, pre-existing, or combinations thereof. The core component is RC-Trees (Chapter 3), and the framework of simple RC operation sets and batched-mixed operations on trees (Chapter 8). Since we are going to use RC-Trees, we require that $G$ have bounded degree. Note that any arbitrary degree graph can easily be *ternarized* by replacing high-degree vertices with cycles of infinite weight edges, resulting in a graph of maximum degree three with the same minimum cut, and only a constant-factor larger size in terms of edges, which our bounds depend on.

The first order of business is to determine an approximate minimum cut work-efficiently in parallel. Our first tool is a $(\log n)$-approximate minimum cut which we develop by taking a sequential approximation algorithm of Karger [113] based on random contractions and parallelizing it by turning it into a sequence of mixed operations on trees and applying our algorithm from Chapter 8. We generalize the parallel sparse certificate algorithm of Cheriyan, Kao, and Thurimella for unweighted graphs to work on weighted graphs, and then use it to generalize Matula's $O(1)$-approximate algorithm for unweighted minimum cuts to graphs with low total weight. We then show how to sample a low-weight skeleton graph efficiently in parallel by taking advantage of the $(\log n)$-approximate minimum cut and find

| **Component** | **Work** | **Comments** |
|---|---|---|
| Simple RC Operation Sets (Chapter 8) | $O(k \log(kn))$ | A framework for evaluating queries on weighted trees |
| | | Minimum 2-respecting Cuts |
| Descendant edges case (Section 9.6.1) | $O(m \log n)$ | Improves on $O(m \log^3 n)$ |
| Independent edges case (Section 9.6.2) | $O(m \log n)$ | Improves on $O(m \log^3 n)$ |
| | | Generating the 2-constraining spanning trees |
| $(\log n)$-approximate min cut (Section 9.3) | $O(m \log^2 n)$ | Parallelizes Karger's random contraction algorithm using simple RC ops |
| Bounded edge weights (Section 9.4) | $O(m)$ | Bounds weights by $O(m \log n)$ preserving an $O(1)$-min-cut |
| Subsampling the skeleton (Section 9.4) | $O(m \log^2 n)$ | Samples $\log n$ skeleton graphs faster than $O(m \log^3 n)$ |
| Parallel $k$-certificate (Section 9.4) | $O(km)$ | Extends the algorithm of Cheriyan et al. for weighted graphs |
| Parallel Matula's algorithm (Section 9.4) | $O(dm \log(W/m))$ | Extends the algorithm of Karger and Motwani for weighted graphs |

**Table 9.1: A summary of the work bounds of the various components of the algorithm and comments comparing to existing work. All components have $O(\text{polylog } n)$ span.**

an $O(1)$-approximate minimum cut using our generalized parallel Matula's algorithm. This approximate cut is finally used to produce the tree packing.

Second, we describe an algorithm for solving the 2-respecting cut problem. Like Karger's algorithm, we separately solve the so-called *descendant edges* case (when the two cut edges are on the same root-to-leaf path) and the independent edges case (when they are not). To solve the descendant edges case, we compute an offline sequence of dynamic tree operations that search for the best pair of descendant edges and parallelize it using our algorithm for batched-mixed operations from Chapter 8. Lastly, we solve the independent edges case by using RC-Trees to perform a parallel divine-and-conquer search of the 2-constraining trees, and using compressed path trees from Chapter 7 to make the search space of relevant edges small and efficient.

Table 9.1 shows the work bound for each of these components of our algorithm and compares them to existing work where relevant.

## 9.2 Producing the Tree Packing

We follow the general approach used by Karger to produce a set of $O(\log n)$ spanning trees such that w.h.p., the minimum cut 2 respects at least one of them. We have to make several improvements to achieve our desired work and span bounds. At a high level, Karger's algorithm works as follows.

1. Compute an $O(1)$-approximate minimum cut $c$
2. Sample edges from the unweighted multigraph corresponding to the weighted graph $G$, where an edge with weight $w$ is represented as $w$ parallel edges, with probability $\Theta(\log n / c)$
3. Use the tree packing algorithm of Plotkin [146] to generate a packing of $O(\log n)$ trees

In this section, we describe the tools required to parallelise this algorithm. Step 2 is trivial to parallelize, as the sampling can be done independently in parallel. The sampling procedure produces an unweighted multigraph with $O(m \log n)$ edges, and takes $O(m \log^2 n)$ work and $O(\log n)$ span.

In Step 3, Plotkin's algorithm consists of $O(\log^2 n)$ minimum spanning tree (MST) computations on a weighting of the sampled graph, which has $O(m \log n)$ edges. Naively this would require $O(m \log^3 n)$ work, but we can use a trick of Gawrychowski et al. [68]. Since the sampled graph is a multigraph sampled from $m$ edges, each invocation of the MST algorithm only cares about the current lightest of each parallel edge, which can be maintained in $O(1)$ time since the weights of the selected edges change by a constant each iteration. Using Cole, Klein, and Tarjan's linear-work MST algorithm [42] results in a total of $O(m \log^2 n)$ work in $O(\log^3 n)$ span w.h.p.

The only nontrivial part of parallelizing the tree production is actually Step 1, computing an $O(1)$-approximate minimum cut. In the sequential setting, Matula's algorithm [129] can be used, which runs in linear time on unweighted graphs, and on weighted graphs in $O(m \log^2 n)$ time. Karger and Motwani [115] give a parallel version of Matula's algorithm, but it takes $O(m^2/n)$ work. Ghaffari and Kuhn [72] present a distributed version of Matula's algorithm in the CONGEST model that runs in $\tilde{O}((D + \sqrt{n})/\epsilon^5)$ rounds. We show how to compute an approximate minimum cut in $O(m \log^2 n)$ work and $O(\log^3 n)$ span, which allows us to prove the following.

**Theorem 43.** Given a weighted graph, in $O(m \log^2 n)$ work and $O(\log^3 n)$ span, a set of $O(\log n)$ spanning trees can be produced such that the minimum cut 2-respects at least one of them w.h.p.

We achieve our bounds by improving Karger's algorithms and speeding up several of the components. We use the following combination of ideas, new and old.

1. We extend a $k$-approximation algorithm of Karger [113] to work in parallel, allowing us to produce a $\log n$-approximate minimum cut in low work and span.
2. We use a faster sampling technique for producing Karger's skeletons for weighted graphs. This is done by transforming the graph into a graph that maintains an approximate minimum cut but has edge weights each bounded by $O(m \log n)$, and then using binomial random variables to sample all of the multiedges of a particular edge at the same time, instead of separately. Subsampling is then used to sample the same graph with decreasing probabilities.
3. We show that Cheriyan, Kao, and Thurimella's parallel sparse $k$-certificate algorithm for unweighted graphs [37] can be modified to run on weighted graphs.
4. We show that Karger and Motwani's parallelization of Matula's algorithm can be generalized to weighted graphs.
5. We use the $\log n$-approximate minimum cut to allow the algorithm to make $O(\log \log n)$ guesses of the minimum cut such that at least one of them is an $O(1)$ approximation.

## 9.3 Parallel $\log n$-Approximate Minimum Cut

If only computing an $O(1)$-approximate minimum cut was simple. As this section will show, it's not so simple, but we will however make use of simple RC operation-sets to compute one. To compute the $O(1)$-approximate minimum cut, we will begin by computing a $(\log n)$-approximate minimum cut. We do so by parallelizing an algorithm of Karger for computing $k$-approximate minimum cuts that is efficient when $k = \Omega(\log n)$ [113].

### 9.3.1 Mixed Connectivity and Component Weight

The following ingredient is useful in parallelizing Karger's $k$-approximate minimum cut algorithm. We show that that the following operations have a simple RC implementation, and hence can be efficiently solved in mixed batches. Given a vertex-weighted, undirected graph with given initial vertex weights, we wish to support:

- **SubtractWeight**$(v, w)$: Subtract weight $w$ from vertex $v$
- **JoinEdge**$(e)$: Mark the edge $e$ as "joined"
- **QueryWeight**$(v)$: Return the weight of the connected component containing the vertex $v$, where the components are induced by the joined edges

**Lemma 47.** The **SubtractWeight**, **JoinEdge**, and **QueryWeight** operations can be supported with a simple RC implementation.

*Proof.* The values stored in the RC clusters are as follows. Vertices store their weight, and unary clusters store the weight of the component reachable via joined edges from the boundary vertex. A binary cluster is either *joined*, meaning that its boundary vertices are connected by joined edges, in which case it stores a single value, the weight of the component reachable via joined edges from the boundaries, otherwise it is *split*, in which case it stores a pair: the weight of the component reachable via joined edges from the top boundary, and the weight of the component reachable via joined edges from the bottom boundary. We provide pseudocode for the update operations for Illustration in Algorithm 18.
The initial value of a vertex is its starting weight. The initial value of an edge is $(0, 0)$, indicating that it is split at the beginning. Note that $f_{\text{unary}}$ and $f_{\text{binary}}$ can be evaluated in constant time, and the structure of the updates involves setting the value at a leaf using an associative operation and re-evaluating the values of the ancestor clusters.

We argue that the values are correctly maintained by structural induction. First consider unary clusters. If the top subcluster is split, then the representative vertex and unary subclusters are not reachable via joined edges, and hence the only reachable component is the component reachable inside the top subcluster from its top boundary, whose weight is $t_v$. If the top subcluster is joined, then the representative vertex is reachable, which is by definition the boundary vertex of the unary subclusters, and hence the reachable component is the union of the reachable components of all of the subclusters, whose weight is as given.

For binary clusters, there are four possible cases, depending on whether the top and bottom subclusters are joined or not. If both are joined, then the representative and hence the

**Algorithm 18** A simple RC implementation of **SubtractWeight** and **JoinEdge**.

```
 1: procedure $f_{\text{UNARY}}(v_v, t, U)$
 2:     if t $= (t_v, b_v)$ then return $t_v$
 3:     else return $v_v + t + \sum_{u_v \in U} u_v$
 4: procedure $f_{\text{BINARY}}(v_v, t, b, U)$
 5:     if $t = t_v$ and $b = b_v$ then
 6:         return $t_v + b_v + v_v + \sum_{u_v \in U} u_v$
 7:     else if $t = (t_{t_v}, t_{b_v})$ and $b = b_v$ then
 8:         return $(t_{t_v}, t_{b_v} + v_v + b_v + \sum_{u_v \in U} u_v)$
 9:     else if $t = t_v$ and $b = (b_{t_v}, b_{b_v})$ then
10:         return $(t_v + v_v + b_{t_v} + \sum_{u_v \in U} u_v)$
11:     else if $t = (t_{t_v}, t_{b_v})$ and $b = (b_{t_v}, b_{b_v})$ then
12:         return $(t_{t_v}, b_{b_v})$
13: procedure SubtractWeight($v, w$)
14:     $v$.value $\leftarrow$ $v$.value - $w$
15:     Reevaluate the $f(\cdot)$ on path to root.
16: procedure JoinEdge($e$)
17:     $e$.value $\leftarrow$ 0
18:     Reevaluate the $f(\cdot)$ on path to root.
```

boundary of all subclusters is reachable from both boundaries, and hence the cluster is joined and the reachable component is the union of the reachable components of the subclusters. If either subcluster is split, then the reachable component at the corresponding boundary is just the reachable component of the subcluster, whose weight is as given. Lastly, if one of the subclusters is not split, then the corresponding boundary can reach the representative vertex, and hence the reachable components of the unary subclusters, whose weights are as given.

It remains to argue that **QueryWeight** has a simple RC implementation. Consider a vertex $v$ whose component weight is desired and consider the parent cluster $P$ of $v$, i.e., the cluster of which $v$ is the representative. If $P$ has no binary subclusters that are joined, observe that $P$ must contain the entire component of $v$ induced by joined edges, since the only way for a component to exit a cluster is via a boundary which would have to be joined. Answering the query in this situation is therefore easy; the result is the sum of the weights of $v$, the unary subclusters of $P$, the bottom boundary weight of the top subcluster (if it exists), and the top bounary weight of the bottom subcluster (if it exists). Suppose instead that $P$ contains a binary subcluster that is joined to some boundary vertex $u \neq v$. Since the subcluster is joined, $u$ is in the same induced component as $v$, and hence **QueryWeight**$(v)$ has the same answer as **QueryWeight**$(u)$. Since $u$ is a boundary of $P$, we also know that the leaf cluster $u$ is the child of some ancestor of $P$. Since the root cluster has no binary subclusters, this process of jumping to joined boundaries must eventually discover a vertex that falls into the easy case, and since such a vertex $u$ is always the child of some ancestor is $P$, the algorithm only examines clusters that are on or are children of the root-to-$v$ path in the RC-Tree, and hence the algorithm is a simple RC implementation. □

Invoking Theorem 42, we obtain the following useful corollary.

**Corollary 8.** Given a vertex-weighted undirected graph, a batch of $k$ **SubtractWeight**, **JoinEdge**, and **QueryWeight** operations can be evaluated in $O(k \log(kn))$ work and $O(\log n \log k)$ span.

### 9.3.2 Parallel $k$-Approximate Minimum Cut

Karger describes an $O(mn^{2/k} \log n)$ time sequential algorithm for finding a cut in a weighted graph within a factor of $k$ of the optimal cut [113]. It works by randomly selecting edges to contract with probability proportional to their weight until a single vertex remains, and keeping track of the component with smallest incident weight (not including internal edges) during the contraction.

His analysis shows that in a weighted graph with minimum cut $c$, with probability $n^{-2/k}$, the component with minimum incident weight encountered during a single trial of the contraction algorithm corresponds to a cut of weight at most $kc$, and therefore by running $O(n^{2/k} \log n)$ trials, we find a cut of size at most $kc$ w.h.p.

Although Karger's contraction algorithm is easy to parallelize using a parallel minimum spanning tree algorithm, keeping track of the incident component weights is trickier. To overcome this problem, we show that we can use our batch component weight algorithm to simulate the sequential contraction process efficiently. With this tool, we can determine the minimum incident weight of a component as follows:

1. Compute an MST with respect to the weighted random edge ordering, where a heavier weight indicates that an edge contracts later
2. For each edge $(u, v) \in G$, determine the heaviest edge in the MST on the $(u, v)$ path
3. Construct a vertex-weighted tree from the MST, where the weights are the total incident weight on each vertex in $G$. For each edge $(u, v)$ in the MST in contraction order:
    - Determine the set of edges in $G$ such that $(u, v)$ is the heaviest edge on its MST path. For each such edge identified, **SubtractWeight** from each of its endpoints by the weight of the edge
    - Perform **JoinEdge** on the edge $(u, v)$
    - Perform **QueryWeight** on the vertex $u$

Observe that the weight of a component at the point in time when it is queried is precisely the total weight of incident edges (again, not including internal edges). Taking the minimum over the initial degrees and all query results therefore yields the desired answer.

Karger shows how to parallelize picking the (weighted) random permutation of the edges with $O(m \log^2 n)$ work. It can easily slightly modified to improve the bounds by a logarithmic factor as follows. The algorithm selects the edges by running a prefix sum over the edge weights. Assuming a total weight of $W$, it then picks $m$ random integers up to $W$, and for each uses binary search on the result of the prefix sum to pick an edge. This process, however, might end up picking only the heaviest edges. Karger shows that by removing those edges the total weight $W$ decreases by a constant factor, w.h.p. To make this efficient, we must first preprocess the edge weights to make them polynomial in $n$. Gawrychowski et al. [68] describe a transformation that affects the value of the minimum cut by no more than a constant factor and bounds all edge weights by $O(n^5)$. Therefore, repeating for $\log n$ rounds the algorithm will select all edges in the appropriate weighted random order. Each round takes $O(m \log n)$ work for a total of $O(m \log^2 n)$ work.

Replacing the binary search in Karger's algorithm with a sort of the random integers and merge into the the result of the prefix sum yields an $O(m \log n)$ work randomized algorithm. In particular $m$ random numbers uniformly distributed over a range can be sorted in $O(m)$ work and $O(\log n)$ span by first determining for each number which of $m$ evenly distributed buckets within the range it is in, then sorting by bucket using an integer sort [152] and finally sorting within buckets.

Step 1 therefore takes $O(m \log n)$ work and $O(\log^2 n)$ span to compute the random edge permutation, and $O(m)$ work and $O(\log n)$ span to run a parallel MST algorithm [117]. Step 2 takes $O(m \log n)$ work and $O(\log n)$ span using RC-Trees, and Step 3 takes $O(m \log n)$ work and $O(\log^2 n)$ span by Corollary 8 and the fact that the algorithm performs a batch of $O(n)$ operations. By Karger's analysis, trying $O(n^{2/k} \log n)$ random contractions yields the following lemma, and setting $k = \log n$ gives our desired corollary.

**Lemma 48.** For a weighted graph, a cut within a factor of $k$ of the minimum cut can be found w.h.p. in $O(mn^{2/k} \log^2 n)$ work and $O(\log^2 n)$ span.

**Corollary 9.** For a weighted graph, a cut within a factor of $\log n$ of the minimum cut can be found w.h.p. in $O(m \log^2 n)$ work and $O(\log^2 n)$ span.

## 9.4 Sampling, Certificates, and Low-Weight Cuts

### 9.4.1 Transformation to Bounded Edge Weights

For our algorithm to be efficient, we require that the input graph has small integer weights. Gawrychowski et al. [68] give a transformation that ensures all edge weights of a graph are bounded by $O(n^5)$ without affecting the minimum cut by more than a a constant factor. For our algorithm $O(n^5)$ would be too big, so we design a different transformation that guarantees all edge weights are bounded by $O(m \log n)$, and only affects the weight of the minimum cut by a constant factor.

**Lemma 49.** There exists a transformation that, given an integer-weighted graph $G$, produces an integer-weighted graph $G'$ no larger than $G$, such that $G'$ has edge weights bounded by $O(m \log n)$, and the minimum cut of $G'$ corresponds to an $O(1)$-approximate minimum cut in $G$.

*Proof.* Let $G$ be the input graph and suppose that the true value of the minimum cut is $c$. First, we use Corollary 9 to obtain a $O(\log n)$-approximate minimum cut, whose value we denote by $\tilde{c}$ $(c \leq \tilde{c} \leq c \log n)$. We can contract all edges of the graph with weight greater than $\tilde{c}$ since they can not appear in the minimum cut. Let $s = \tilde{c}/(2m \log n)$. We delete (not contract) all edges with weight less than $s$. Since there are at most $m$ edges in any cut, this at most affects the value of a cut by $sm = \tilde{c}/(2\log n) \leq c/2$. Therefore the minimum cut in this graph is still a constant factor approximation to the minimum cut in $G$.

Next, scale all remaining edge weights down by the factor $s$, rounding down. All edge weights are now integers in the range $[1, 2m \log n]$. This is the transformed graph $G'$. It remains to argue that the value of the minimum cut is a constant-factor approximation. First, note that the scaling process preserves the order of cut values, and hence the true minimum cut in $G$ has the same value in $G'$ as the minimum cut in $G'$. Consider any cut in $G'$, and scale the weights of the edges back up by a factor $s$. This introduces a rounding error of at most $s$ per edge. Since any cut has at most $m$ edges, the total rounding error is at most $sm \leq c/2$. Therefore the value of the minimum cut in $G'$ is a constant factor approximation to the value of the minimum cut in $G$. □

Lastly, observe that this transformation can easily be performed in parallel by using a work-efficient connected components algorithm to perform the edge contractions, as is standard (see e.g. [116]).

## 9.4.2 Sampling Binomial Random Variables

It will be helpful in the next step to be able to efficiently sample binomial random variables. We will use the following results due to Farach-Colton et al. [57].

**Lemma 50** (Farach-Colton et al. [57], Theorem 1)**.** Given a positive integer $n$, one can sample a random variate from the binomial distribution $B(n, 1/2)$ in $O(1)$ time with probability $1 - 1/n^{\Omega(1)}$ and in expectation after $O(n^{1/2+\varepsilon})$-time preprocessing for any constant $\varepsilon > 0$, assuming that $O(\log n)$ bits can be operated on in $O(1)$ time. The preprocessing can be reused for any $n' = O(n)$

We can also use the following reduction to sample $B(n, p)$ for arbitrary $0 \leq p \leq 1$.

**Lemma 51** (Farach-Colton et al. [57], Theorem 2)**.** Given an algorithm that can draw a sample from $B(n', 1/2)$ in $O(f(n))$ time with probability $1 - 1/n^{\Omega(1)}$ and in expectation for any $n' \leq n$, then drawing a sample from $B(n', p)$ for any real p can be done in $O(f(n) \log n)$ time with probability $1 - 1/n^{\Omega(n)}$ and in expectation, assuming each bit of p can be obtained in $O(1)$ time

We note, importantly, that the model used by Farach-Colton et al. assumes that random $\Theta(\log n)$-size words can be generated in constant time. Since we only assume that we can generate random bits in constant time, we will have to account for this with an extra $O(\log n)$ factor in the work where appropriate. Note that this does not negatively affect the span since we can pre-generate as many random words as we anticipate needing, all in parallel at the beginning of our algorithm. Also, although it might not be clear in their definition, the constants in the algorithm can be configured to control the constant in the $\Omega(1)$ term in the probability, and therefore their algorithms take $O(1)$ time and $O(\log n)$ time w.h.p.

To make use of these results, we need to show that the preprocessing of Lemma 52 can be parallelized. Thankfully, it is easy. The preprocessing phase consists of generating $n^\varepsilon$ alias tables of size $O(\sqrt{n \log n})$. Hübschle-Schneider and Sanders [104] give a linear work, $O(\log n)$ span parallel algorithm for building alias tables. Building all of them in parallel means we can perform the alias table preprocessing in $O(n^{1/2+\varepsilon})$ work and $O(\log n)$ span. The last piece of preprocessing information that needs to be generated is a lookup table for decomposing any integer $n' = O(n)$ into a sum of a constant number of square numbers. This table construction is trivial to parallelize, and hence all preprocessing runs in $O(n^{1/2+\varepsilon})$ work and $O(\log n)$ span.

**Lemma 52.** Given a positive integer $n$, after $O(n^{1/2+\varepsilon})$ work and $O(\log n)$ span preprocessing, one can sample random variables from $B(n, 1/2)$ in $O(\log n)$ work w.h.p., and from $B(n, p)$ in $O(\log^2 n)$ work w.h.p. The preprocessing can be reused for any $n' = O(n)$.

### 9.4.3 Subsampling $p$-Skeletons

Karger defines the $p$-skeleton $G(p)$ of an unweighted graph $G$ as a copy of $G$ where each edge appears with probability $p$. A $p$-skeleton therefore has $O(pm)$ edges in expectation. For a weighted graph, the $p$-skeleton is defined as the $p$-skeleton of the corresponding unweighted multigraph in which an edge of weight $w$ is replaced by $w$ parallel multiedges. The $p$-skeleton of a weighted graph therefore has $O(pW)$ edges in expectation, where $W$ is the total weight in the graph. Karger gives an algorithm for generating a $p$-skeleton in $O(pW\log(m))$ work, which relies on performing $O(pW)$ independent random samples with probabilities proportional to the weight of each edge, each of which takes $O(\log(m))$ amortized time. In Karger's algorithm, given a guess of the minimum cut $c$, he computes $p$-skeletons for $p = \Theta(\log n/c)$. Since no edge of weight greater than $c$ can be contained in the minimum cut, all such edges can be contracted, leaving us with $W \le mc$, so the skeleton has $O(m \log n)$ edges and takes $O(m\log^2 n)$ work to compute. Since our algorithm does not know the minimum cut $c$ yet, it uses guessing and doubling on $p$, and hence has to compute several $p$-skeletons, so $O(m\log^2 n)$ work is too slow. We overcome this problem using binomial random variables and subsampling.

**Lemma 53.** Given a weighted graph $G$ with edge weights bounded by $m^{2-\varepsilon}$, an initial sampling probability $p$ and an integer $k$, there exists an algorithm that can produce the skeleton graphs $G(p), G(p/2), \ldots, G(p/2^k)$ in $O(m\log^2 n + km\log n)$ work w.h.p. and $O(k\log n)$ span.

*Proof.* Begin by using Lemma 52 and performing the required preprocessing for sampling binomial random variables from $B(m^{2-\varepsilon}, 1/2)$, which takes $O(m)$ work and $O(\log n)$ span. To construct $G(p)$, for each edge $e$ in the graph, sample a binomial random variable $x \sim B(w(e), p)$. The skeleton then contains the edge $e$ with weight $x$ (conceptually, $x$ unweighted copies of the multiedge $e$). This results in the same distribution of graphs as if sampled using Karger's technique, and takes $O(m\log^2 n)$ work w.h.p. and $O(\log n)$ span. For each additional skeleton $G(p')$ requested, subsample from the previous skeleton by drawing binomial random variables from $B(w_{G(2p')}(e), 1/2)$, which takes $O(m \log n)$ work w.h.p. and $O(\log n)$ span. In total, to perform $k$ rounds of sampling, this takes $O(m\log^2 n + km\log n)$ work w.h.p. and $O(k \log n)$ span. □

Using subsampling here is important, since otherwise it would cost $O(km\log^2 n)$ work to sample all of the desired skeleton graphs. Additionally, note that Lemma 49 makes it easy to satisfy the requirement that all edge weights be bounded by $m^{2-\varepsilon}$.

## 9.4.4 Parallel Weighted Sparse Certificates

A *sparse $k$-connectivity certificate* of an unweighted graph $G = (V, E)$ is a graph $G' = (V, E' \subset E)$ with at most $O(kn)$ edges, such that every cut in $G$ of weight at most $k$ has the same weight in $G'$. Cheriyan, Kao, and Thurimella [37] introduce a parallel graph search called *scan-first search*, which they show can be used to generate $k$-connectivity certificates of unweighted graphs. Here, we briefly note that the algorithm can easily be extended to handle weighted graphs. The scan-first search algorithm is implemented as follows.

**Algorithm 19** Scan-first search [37]

```
1: procedure SFS($G = (V, E)$ : Graph, $r$ : Vertex)
2:     Find a spanning tree $T'$ rooted at $r$
3:     Find a preorder numbering to the vertices in $T'$
4:     For each vertex $v \in T'$ with $v \neq r$, let $b(v)$ denote the least neighbor of $v$ in preorder
5:     Let $T$ be the tree formed by $\{v, b(v)\}$ for all $v \neq r$
```

Note that the scan-first search tree of a connected graph is always a tree. If the graph is disconnected, the result is a scan-first search tree of each component. Using a linear work, low span spanning tree algorithm, scan-first search can easily be implemented in $O(m)$ work and $O(\log n)$ span. Cheriyan, Kao, and Thurimella show that if $E_i$ are the edges in a scan-first search forest of the graph $G_{i-1} = (V, E \setminus (E_1 \cup ... E_{i-1}))$, then $E_1 \cup ... E_k$ is a sparse $k$-connectivity certificate. A sparse $k$-connectivity certificate can therefore be found in $O(km)$ work and $O(k \log n)$ span by running scan-first search $k$ times.

In the weighted setting, we treat an edge of weight $w$ as $w$ parallel unweighted multiedges. As always, this is only conceptual, the multigraph is never actually generated. To compute certificates in weighted graphs, we therefore use the following simple modification. After computing each scan-first search tree, instead of removing the edges present from $G$, simply lower their weight by one, and remove them only if their weight becomes zero. It is easy to see that this is equivalent to running the ordinary algorithm on the unweighted multigraph. We therefore have the following.

**Lemma 54.** A sparse $k$-connectivity certificate for a weighted, undirected graph can be found in $O(km)$ work and $O(k \log n)$ span.

### 9.4.5 Parallelizing Matula's Algorithm

Matula [129] gave a linear time sequential algorithm for $(2+\varepsilon)$-approximate edge connectivity (unweighted minimum cut). It is easy to extend to weighted graphs so that it runs in $O(m \log n \log W)$ time, where $W$ is the total weight of the graph. Using standard transformations to obtain polynomially bounded edge weights, this gives an $O(m \log^2 n)$ algorithm. Karger and Motwani [115] gave a parallel version of Matula's unweighted algorithm that runs in $O(m^2/n)$ work. Essentially, their version of Matula's algorithm does the following steps as indicated in Algorithm 20.

**Algorithm 20** Approximate minimum cut

```
1: procedure MATULA($G = (V, E)$ : Graph)
2:     if $|V| = 1$ then return $\infty$
3:     local $d \leftarrow$ minimum degree in $G$
4:     local $k \leftarrow d/(2+\varepsilon)$
5:     local $C \leftarrow$ Compute a sparse $k$-certificate of $G$
6:     local $G' \leftarrow$ Contract all non-certificate edges of $E$
7:     return min($d$, MATULA($G'$))
```

It can be shown that at each iteration, the size of the graph is reduced by a constant factor, and hence there are at most $O(\log n)$ iterations. Furthermore, the work performed at each step is geometrically decreasing, so the total work, using the sparse certificate algorithm of Cheriyan, Kao, and Thurimella [37] is $O(dm)$ and the span is $O(d \log^2 n)$, where $d$ is the minimum degree of $G$.

Here, we give a slight modification to this algorithm that makes it work on weighted graphs in $O(dm \log(W/m))$ work and $O(d \log n \log W)$ span, where $d$ is the minimum weighted degree of the graph. To extend the algorithm to weighted graphs, we can replace the sparse certificate routine with our modified version for weighted graphs, and replace the computation of $d$ with the equivalent weighted degree. By interpreting an edge-weighted graph as a multigraph where each edge of weight $w$ corresponds to $w$ parallel multiedges, we can see that the algorithm is equivalent. To argue the cost bounds, note that like in the original algorithm where the size of the graph decreases by a constant factor each iteration, the total weight of the graph must decrease by a constant factor in each iteration. Because of this, it is no longer true that the work of each iteration is geometrically decreasing. Naively, this gives a work bound of $O(dm \log(W))$, but we can tighten this slightly as follows. Observe that after performing $\log(W/m)$ iterations, the total weight of the graph will have been reduced to $O(m)$, and hence, like in the sequential algorithm, the work must subsequently begin to decrease geometrically. Hence the total work can actually be bounded by $O(dm \log(W/m) + dm) = O(dm \log(W/m))$. We therefore have the following.

**Lemma 55.** Given a weighted graph with minimum weighted-degree $d$ and total weight $W$, an $O(1)$-approximate minimum cut can be found in $O(dm \log(W/m))$ work and $O(d \log n \log W)$ span.

## 9.5 Parallel $O(1)$-Approximate Minimum Cut

We have finally amassed the ingredients needed to produce a parallel $O(1)$-approximate minimum cut algorithm. Well, we need one more trick, unsurprisingly due to Karger. To produce the sampled skeleton graph, Karger's algorithm chooses the sampling probability inversely proportional to the weight of the minimum cut, which paradoxically is what we are trying to compute. This issue is solved by using guessing and doubling. The algorithm guesses the minimum cut and computes the resulting approximation. It can then use Karger's sampling theorem (Theorem 6.3.1 and Lemma 6.3.2 of [112]) to verify whether the guess was too high.

**Lemma 56** (Karger [112])**.** Let $G$ be a graph with minimum cut $c$ and let $p = \Theta((\log n)/\varepsilon^2 c)$. Then w.h.p. the minimum cut in $G(p)$ has value in $(1 \pm \varepsilon)pc$.

**Lemma 57** (Karger [112])**.** W.h.p., if $G(p)$ has minimum cut $\hat{c} = \Theta((\log n)\varepsilon^2)$ for $\varepsilon \leq 1$, then the minimum cut $c$ in $G$ has value in $(1 \pm \varepsilon)\hat{c}/p$.

If the true minimum cut is $c$, then the correct sampling probability for Karger's algorithm is $p = \Theta((\log n)\varepsilon^2 c)$, which produces a skeleton cut of size $\hat{c} = \Theta((\log n)/\varepsilon^2)$ w.h.p. If the algorithm makes a guess $C > 2c$ with corresponding probability $P = \Theta((\log n)/\varepsilon^2 C)$, then Lemma 57 says that the minimum cut in the skeleton graph is less than $\hat{c}$ w.h.p. The algorithm can therefore double the guess for $P$ and try again, until the minimum cut in the skeleton is larger than $\hat{c}$, at which point we know that the $P$-skeleton approximates the minimum cut within a factor $\varepsilon$. To perform these steps efficiently, our algorithm does the following:

1. Use Corollary 9 to compute a $\log n$-approximate minimum cut value $C$ in $O(m \log^2 n)$ work and $O(\log^2 n)$ span.
2. Transform the graph using Lemma 49 to ensure the weights are bounded by $O(m \log n)$ and retaining an $O(1)$-approximate minimum cut in $O(m \log^2 n)$ work and $O(\log^2 n)$ span.
3. Sample the skeleton graphs $G(\log^2 n/C), G(\log^2 n/(2C)), \ldots, G(\log n/C)$ using Lemma 53. This is $\log\log n$ skeletons, and hence takes $O(m \log^2 n)$ work w.h.p. and $O(\log^2 n)$ span.
4. For each skeleton graph:
   - Compute a sparse $\Theta(\log n)$ certificate of the skeleton graph. This takes $O(m \log n)$ work and $O(\log^2 n)$ span by Lemma 54.
   - Compute an $O(1)$-approximate minimum cut in the $\Theta(\log n)$ certificate using Matula's algorithm (Lemma 55). Since the certificate guarantees that the total weight is at most $O(n \log n)$ and hence that the minimum weighted degree is at most $O(\log n)$, this takes $O(m \log n \log\log n)$ work and $O(\log^2 n \log\log n)$ span.

Since there are $O(\log\log n)$ skeleton graphs, the total work done by the final step is at most $O(m \log n (\log\log n)^2)$, which is at most $O(m \log^2 n)$, and the span is $O(\log^3 n)$. The correctness of the algorithm follows from the sampling theorem (Lemma 57) and Karger's discussion [112]. Finally, we can conclude the following result.

**Lemma 58.** Given a weighted, undirected graph, the weight of an $O(1)$-approximate minimum cut can be computed w.h.p. in $O(m\log^2 n)$ work and $O(\log^3 n)$ span

## 9.6 Finding Minimum 2-respecting Cuts

We are given a connected, weighted, undirected graph $G = (V, E)$ and a spanning tree $T$. In this section, we will give an algorithm that finds the minimum 2-respecting cut of $G$ with respect to $T$ in $O(m\log n)$ work and $O(\log^3 n)$ span.

Our algorithm, like those that came before it, finds the minimum 2-respecting cut by considering two cases. We assume that the tree $T$ is rooted arbitrarily. In the first case, we assume that the two tree edges of the cut occur along the same root-to-leaf path, i.e. one is a descendant of the other. This is called the *descendant edges* case. In the second case, we assume that the two edges do not occur along the same root-to-leaf path. This is the *independent edges* case.

### 9.6.1 Descendant Edges

We present our minimum 2-respecting cut algorithm for the descendant edges case. Let $T$ be a spanning tree of a connected graph $G = (V, E)$ of degree at most three, and root $T$ at an arbitrary vertex of degree at most two. The rooted tree is therefore a binary tree.

We use the following fact. For any tree edge $e \in T$, let $F_e$ denote the set of edges $(u, v) \in E$ (tree and non-tree) such that the $u$ to $v$ path in $T$ contains the edge $e$. Then the weight of the cut induced by a pair of edges $\{e, e'\}$ in $T$ is given by

$$w(F_e \Delta F_{e'}) = w(F_e) + w(F_{e'}) - 2w(F_e \cap F_{e'}),$$

where $\Delta$ denotes the symmetric difference between two sets. For each tree edge $e$, our algorithm seeks the tree edge $e'$ that minimizes $w(F_e \Delta F_{e'})$, which is equivalent to minimizing

$$w(F_{e'}) - 2w(F_e \cap F_{e'}).$$

To do so, it traverses $T$ from the root while maintaining weights on a tree data structure that satisfies the following invariant:

**Invariant 3** (Current subtree invariant)**.** When visiting $e = (u, v)$, for every edge $e' \in \text{Subtree}(v)$, the weight of $e'$ in the dynamic tree is $w(F_{e'}) - 2w(F_e \cap F_{e'})$

The initial weight of each edge $e$ is therefore $w(F_e)$. Maintaining this invariant as the algorithm traverses the tree can then be achieved with the following observation. When the traversal descends from an edge $p = (w, u)$ to a neighboring child edge $e = (u, v)$, the following hold for all $e' \in \text{Subtree}(v)$:

1. $(F_e \cap F_{e'}) \supseteq (F_p \cap F_{e'})$, since any path that goes through $p$ and $e'$ must pass through $e$.

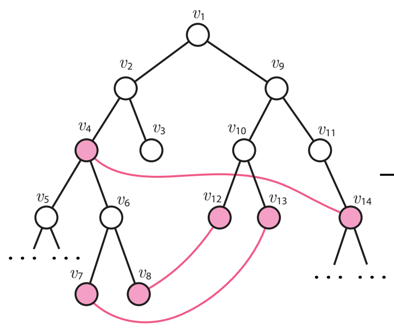
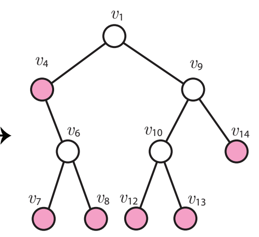
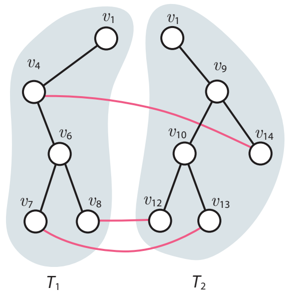




**Figure 9.1: The bipartite problems are generated by compressing the tree with respect to the endpoints of the edges whose endpoints share an LCA, then splitting the tree into left and right halves.**

2. $(F_e \cap F_{e'}) \setminus (F_p \cap F_{e'})$ are the edges $(x, y) \in F_{e'}$ such that $e$ is a *top edge* of the path $x - y$ in $T$ (i.e., $e$ is on the path from $x$ to $y$ in $T$, but the parent edge of $e$ is not).

Therefore, to maintain the current subtree invariant, when the algorithm visits the edge $e$, it need only subtract twice the weight of all $x - y$ paths that contain $e$ as a top edge. This can be done efficiently by precomputing the sets of top edges. There are at most two top edges for each path $x - y$, and they can be found from the LCA of $x$ and $y$ in $T$. We need not consider tree edges since they will never appear in $F_{e'}$. By maintaining the aforementioned invariant, the solution follows by taking the minimum value of $w(F_e) + \text{QUERYSUBTREE}(v)$ for all edges $e = (u, v)$ during the traversal. This algorithm sounds entirely sequential, but it can be parallelized using our batched mixed operations algorithm (Corollary 7 from Chapter 8).

The operation sequence can be generated as follows. First, the weights $w(F_e)$ for each edge can be computed using the batched mixed operations algorithm (Corollary 7) where each edge $(u, v)$ of weight $w$ creates an $\text{ADDPATH}(u, v, w)$, followed by $\text{QUERYEDGE}(e)$ for every edge $e \in T$. This takes $O(m \log n)$ work and $O(\log^2 n)$ span. The LCAs required to compute the sets of top edges can be computed using the parallel LCA algorithm of Schieber and Vishkin [157] in $O(m)$ work and $O(\log n)$ span in total. By computing an Euler tour of the tree $T$ (an ordered sequence of visited edges) beginning at the root, the order in which to perform the tree operations can be deduced in $O(n)$ work and $O(\log n)$ span. Each edge in the Euler tour generates an ADDPATH operation for each of its top edges, followed by a QUERYSUBTREE operation. Note that each edge is visited twice during the Euler tour. The second visit corresponds to negating the ADDPATH operations from the first visit. The solution is then the minimum result of all of the QUERYSUBTREE operations. Since there are a constant number of top edges per path, and $O(m)$ paths in total, the operation sequence has length $O(m)$. Using Corollary 7, we arrive at the following.

**Theorem 44.** Given a weighted, undirected graph $G$ and a rooted spanning tree $T$, the minimum 2-respecting cut of $G$ with respect to $T$ such that one of the cut edges is a descendant of the other can be computed in in $O(m \log n)$ work and $O(\log^2 n)$ span w.h.p.

### 9.6.2 Independent Edges

The independent edge case is where the two cutting edges do not fall on the same root-to-leaf path. To solve the independent edges problem, we use the framework of Gawrychowski et al. [68], which is to decompose the problem into a set of subproblems, which they call *bipartite problems*. The key challenge in parallelizing the solution to the bipartite problem is dealing with the fact that the resulting trees might not be balanced. The algorithm of Gawrychowski et al. relies on performing a biased divide-and-conquer search guided by a heavy-light decomposition [93], and then propagating results up the trees bottom up. Since the trees may be unbalanced, this can not be easily parallelized. Our solution is to use the recursive clustering of RC-Trees to guide a divide and conquer search in which we can maintain all of the needed information on the clusters.

**Definition 14** (The bipartite problem)**.** Given two weighted rooted trees $T_1$ and $T_2$ and a set of weighted edges that cross from one to the other, $L = \{(u,v) : u \in T_1, v \in T_2\}$, the bipartite problem is to select $e_1 \in T_1$ and $e_2 \in T_2$ with the goal of minimizing the sum of the weight of $e_1$ and $e_2$ plus the weights of all edges $(v_1, v_2) \in L$ such that $v_1$ is in the subtree rooted at the bottom endpoint of $e_1$ and $v_2$ is in the subtree rooted at the bottom endpoint of $e_2$. The size of a bipartite problem is the size of $L$ plus the size of $T_1$ and $T_2$.

Gawrychowski et al. observe that if $T_1$ and $T_2$ are edge-disjoint subtrees of $T$, then, assigning weights of $w(F_e)$ to each tree edge and weights of $-2w(e)$ to each edge non-tree, the solution to the bipartite problem is the minimum 2-respecting cut such that $e_1 \in T_1$ and $e_2 \in T_2$. The independent edges problem is then solved by reducing it to several instances of the bipartite problem, and taking the minimum answer among all of them. We will show how to generate the bipartite problems efficiently, and how to solve them efficiently, both in parallel.

#### Generating the bipartite problems

The following parallel algorithm generates $O(n)$ instances of the bipartite problem with total size at most $O(m)$. For each edge $e$ in $T$, the algorithm first assigns them a weight equal to $w(F_e)$. Now consider all non-tree edges, i.e. all edges $e \in E, e \notin T$, group them by the LCA of their endpoints in $T$, and assign them a weight of $-2w(e)$. This forms a partition of the $O(m)$ edges of $G$, each group identified by a vertex. Each vertex in $T$ conversely has an associated (possibly empty) list of non-tree edges.

For each vertex $v$ in $T$ with a non-empty associated list of edges, create a compressed path tree (Chapter 7) of $T$ with respect to the endpoints of the associated edges and $v$. Finally, for each such compressed path tree, root it at $v$ (the common LCA of the edge endpoints). The bipartite problems are generated as follows. For each vertex $v$ with a non-empty list of non-tree edges, and the corresponding compressed path tree $T_v$, consider the children $x, y$ of $v$ in $T_v$. The bipartite problem consists of $T_1$, which contains the edge $(v, x)$ and the subtree of $T_v$ rooted at $x$, and likewise, $T_2$, which contains the edge $(v, y)$ and the subtree of $T_v$ rooted at $y$, and $L$, the list of non-tree edges. See Figure 9.1 for an illustration.

**Lemma 59.** Given a tree and a set of non-tree edges, the corresponding bipartite problems can be generated in $O(m \log n)$ work and $O(\log^2 n)$ span w.h.p.

*Proof.* The edge weight values can be computed in the same way as before using our batched mixed operations on trees algorithm in $O(m \log n)$ work and $O(\log^2 n)$ span. LCAs can be computed using the parallel LCA algorithm of Schieber and Vishkin [157] in $O(m)$ work and $O(\log n)$ span. Grouping the edges by LCA can be achieved using a parallel sorting algorithm in $O(m \log n)$ work and $O(\log n)$ span. Together, these steps take $O(m \log n)$ work and $O(\log^2 n)$ span. For each group, computing the compressed path tree takes $O(m_i \log(1 + n/m_i)) \le O(m_i \log n)$ work and $O(\log^2 n)$ span w.h.p., where $m_i$ is the number of edges in the group. Performing all compressed path tree computations in parallel and observing that the edge lists of each vertex are a disjoint partition of the edges of $G$, this takes at most $O(m \log n)$ work and $O(\log^2 n)$ span in total w.h.p. □

It remains only for us to show that the bipartite problems can be efficiently solved in parallel.

## Solving the bipartite problems

Our solution is a recursive algorithm that utilizes the recursive cluster structure of RC-Trees. Recall that RC-Trees consist of unary and binary clusters (and the nullary cluster at the root, but this is not needed by our algorithm).

Since the bipartite problems are constructed such that trees $T_1$ and $T_2$ always have a root with a single child, the root cluster of their RC-Trees consists of exactly one unary cluster.

**High-level idea** Recall that the goal is to select an edge $e_1 \in T_1$ and an edge $e_2 \in T_2$ that minimizes their costs plus the cost of all edges $(u, v) \in L$ such that $u$ is a descendant of $e_1$ and $v$ is a descendant of $e_2$. Our algorithm first constructs an RC-Tree of $T_1$, and weights the edges in $T_1$ and $T_2$ by their cost. At a high level, the algorithm then works as follows. Given a binary cluster $c_1$ of $T_1$, the algorithm maintains weights on $T_2$ such that for each edge $e_2 \in T_2$, its weight is the weight of $e_2$ in the original tree plus the sum of the weights of all edges $(u, v) \in L$ such that $u$ is a descendant of the bottom boundary vertex of $c_1$, and $v$ is a descendant of $e_2$. This implies that for a binary cluster of $T_1$ consisting of an isolated edge $e_1 \in T_1$, the weights of each $e_2 \in T_2$ are precisely such that $w(e_1) + w(e_2)$ is the value of selecting $\{e_1, e_2\}$ as the solution. This idea leads to a very natural recursive algorithm. We start with the topmost unary cluster of $T_1$ and proceed recursively down the clusters of $T_1$, maintaining $T_2$ with weights as described. When the algorithm recurses into the top binary child of a cluster, it must add the weights of all $(u, v) \in L$ that are descendants of that cluster to the corresponding paths in $T_2$. If recursing on the bottom binary subcluster of a binary cluster, the weights on $T_2$ are unchanged. When recursing on a unary cluster, since it has no descendants, the algorithm uses the original weights of $T_2$. Once the recursion hits a binary cluster that consists of a single edge $e_1$, it can return the solution $w(e_1) + w(e_2)$, where $e_2$ is the lightest edge with respect to the current weights on $T_2$. Lastly, to perform this process efficiently, the algorithm *compresses*, using the compressed path tree algorithm (Chapter 7), the tree $T_2$ every time it recurses, keeping only the vertices that are endpoints of the crossing edges that touch the current cluster of $T_1$.

**Implementation** We provide pseudocode for our algorithm in Algorithm 21. Given a bipartite problem $(T_1, T_2, L)$, we use the notation $L(C)$ to denote the edges of $L$ limited to those that are incident on some vertex in the cluster $C$. Furthermore, we use $V_{T_2}(L(C))$ to denote the set of vertices given by the endpoints of the edges in $L(C)$ that are in $T_2$. The pseudocode does not make the parallelism explicit, but all that is required is to run the recursive calls in parallel. The procedure takes as input a cluster $C$ of $T_1$, a compressed version of $T_2$ with its original weights, and $T_2'$, the compressed version of $T_2$ with updated weights. At the top level, it takes the cluster representing all of $T_1$ for the first argument, and the cluster for all of $T_2$ for the second and third argument. The COMPRESS function compresses the given tree with respect to the given vertex set and its root, and returns the compressed tree still rooted at the same root. ADDPATHS($S$) takes a set $S \subset L$ of edges and for each one, adds $w(u, v)$ to the root-to-$v$ path, where $v \in T_2$, returning a new tree.

**Algorithm 21** Parallel bipartite problem algorithm
```
 1: procedure BIPARTITE($C$, $T_2$, $T_2'$, $L$)
 2:     if $C = \{e\}$ then
 3:         return $w(e) +$ LIGHTESTEDGE($T_2'$)
 4:     else
 5:         $T_{\text{cmp}} \leftarrow T_2$.COMPRESS($V_{T_2}(L(C.t))$)
 6:         $T_2'' \leftarrow T_2'$.ADDPATHS($L(C) \setminus L(C.t)$)
 7:         $T_{\text{cmp}}'' \leftarrow T_2''$.COMPRESS($V_{T_2}(L(C.t))$)
 8:         ans $\leftarrow$ BIPARTITE($C.t$, $T_{\text{cmp}}$, $T_{\text{cmp}}''$, $L(C.t)$)
 9:         for each cluster $C'$ in $C.U$ do
10:             $T_{\text{cmp}} \leftarrow T_2$.COMPRESS($V_{T_2}(L(C'))$)
11:             ans $\leftarrow$ min(ans, BIPARTITE($C'$, $T_{\text{cmp}}$, $T_{\text{cmp}}$, $L(C')$))
12:         if $C$ is a binary cluster then
13:             $T_{\text{cmp}} \leftarrow T_2$.COMPRESS($V_{T_2}(L(C.b))$)
14:             $T_{\text{cmp}}' \leftarrow T_2'$.COMPRESS($V_{T_2}(L(C.b))$)
15:             ans $\leftarrow$ min(ans, BIPARTITE($T_{\text{cmp}}$, $T_{\text{cmp}}'$, $L(C.b)$))
16:         return ans
```

Since this algorithm creates copies of $T_2$, we must ensure that we can still identify a desired vertex given its label. One simple way to achieve this is to build a static hashtable alongside each copy of $T_2$ that maps vertex labels to the instance of that vertex in that copy.

An ingredient that we need to achieve low span is an efficient way to update the weights in $T_2$ when adding weights to a collection of paths. This is easy to achieve in linear work and $O(\log n)$ span by propagating the total weight of all updates up the clusters, and then propagating back down the tree, the weight of all updates that are descendants of the current cluster. It remains to analyze the cost of the BIPARTITE procedure.

**Lemma 60.** Solving a bipartite problem of size $m$ takes $O(m \log m)$ work and $O(\log^3 m)$ span w.h.p.

*Proof.* First, since all recursive calls are made in parallel and the recursion is on the clusters of $T_1$, the number of levels of recursion is $O(\log m)$ w.h.p. We will show that the algorithm performs $O(m)$ work in total at each level, in $O(\log^2 m)$ span w.h.p. Observe first that at each level of recursion, the edges $L$ for each call are a disjoint partition of the non-tree edges, since each recursive call takes a disjoint subset. We will now argue that each call does work proportional to $|L|$. Since $T_2$ and $T_2'$ are both compressed with respect to $L$, their size is proportional to $|L|$. AddPaths takes linear work in the size of $T_2$ and $O(\log m)$ span, and hence takes $O(|L|)$ work and $O(\log m)$ span. Compress$(K)$ takes $O(|K| \log(1+|T_2|/|K|)) \leq O(|K|+|T_2|)$ work and $O(\log^2 m)$ span w.h.p.. Since compression is with respect to some subset of $L$, all of the compress operations take $O(|L|)$ work and $O(\log^2 m)$ span w.h.p. In total, this is $O(|L|)$ work in $O(\log^2 m)$ span w.h.p. at each level for each call. Since the $L$s at each level are a disjoint partition of the non-tree edges, the total work per level is $O(m)$ w.h.p., and hence the desired bounds follow. □

Since there are $O(n)$ bipartite problems of total size $O(m)$, solving them all in parallel yields the following.

**Theorem 45.** Given a weighted, undirected graph $G$ and a rooted spanning tree $T$, the minimum 2-respecting cut of $G$ with respect to $T$ such that the cut edges are independent can be computed in $O(m \log n)$ work and $O(\log^3 n)$ span w.h.p.

Combining Theorems 43, 44, and 45 on each of the $O(\log n)$ trees proves our main result.

## 9.7 Discussion

We presented the first work-efficient algorithm for minimum cuts that runs in low span. That is, the first highly parallel algorithm that performs no more work than the best sequential algorithm. Since our algorithm is work efficient, finding a faster parallel algorithm would entail finding a faster sequential algorithm. Our algorithm is Monte Carlo and it runs in $O(m \log^2 n)$ work and $O(\log^3 n)$ span. It remains an open problem to find a deterministic algorithm, even a sequential one, that runs in $O(m \operatorname{polylog} n)$ time.

Our algorithm is a strong example application of dynamic data structures, such as RC-Trees, to solving non-dynamic (i.e. static) problems. A similar result was recently obtained by Ghaffari et al. [74] who also used our RC-Tree data structure to implement a nearly work-efficient parallel depth-first search. It would be interesting to explore further what results we can obtain by using RC-Trees to speed up and parallelise existing static algorithms, possibly using our batched mixed operations framework.

# Part IV

# Parallel Self-Adjusting Computation

# Chapter 10
# Parallel Self-Adjusting Computation

## 10.1 Introduction

Self-adjusting computation is an approach to automatically, or semi automatically, convert a (suitable) static algorithm to a dynamic one [1, 2, 3, 4, 36, 50, 90, 150]. Most often, self-adjusting computation is implemented in the form of a *change propagation* algorithm. The idea, roughly, is to run a static algorithm while keeping track of data dependencies. Then when an input changes (e.g. adding an edge to a graph), the change can be propagated through the computation, updating intermediate values, creating new dependencies, and updating the final output. Not all algorithms are suitable for the approach—for some, updating a single input value could propagate changes through most of the computation. To account for how much computation needs to be rerun, researchers have studied the notion of "stability" [1, 4] over classes of changes. The goal is to bound the "distance" between executions of a program on different inputs based on the distance between the inputs. For example, for an appropriate sorting algorithm adding an element to the unsorted input list ideally would cause at most $O(\log n)$ recomputation, and that recomputation could be propagated with a constant factor overhead. Achieving this would lead to the performance of a dynamic binary search tree. On the other hand, an unstable sorting algorithm might result in $\Omega(n)$ recomputation for a single update, which would be inefficient to try to propagate.

In the sequential setting this approach has been applied to a wide variety of algorithms, with various bounds on the stability, and also cost of change propagation as a function of the computational distance. Applications includes dynamic trees [4], kinetic data structures [6, 8, 13], computational geometry [9, 11], Huffman coding [10], and Bayesian inference [7]. Self-adjusting computation has also been extended in several directions. Notable works include work on "on-demand" updates with Adapton [91], the CEAL language [89, 90], and automatic derivation of self-adjusting programs via information-flow type systems [34, 35].

More recent work [12, 22, 24, 32, 88] has studied applying change propagation in parallel, allowing for batch dynamic updates—e.g., adding a set of edges to an existing graph and then propagating those changes in parallel. Batch updates are particularly important in practice due to the rapid rate of modifications to very large data sets such as the web graph or social networks. Furthermore, in principle, parallelism and change propagation should work well together since algorithms with shallow dependence chains tend both to be good for parallelism (since fewer dependencies means more task can run in parallel) and for dynamic updates (since changes will not have to propagate as deeply). Indeed, several researchers have studied the approach and developed systems in the applied setting, which show good performance improvements [22, 24, 32] on tasks such as map-reduce.

In Chapter 4, we studied bounds on the cost of change propagation for the class of so-called "round-synchronous" computations. We used this to generate efficient algorithms for batch-dynamic tree contraction and batch-dynamic trees supporting batches of links and cuts among other operations. However, the round-synchronous model limits the applicability to a small set of algorithms that fit the definition.

In this chapter we develop a more general framework for supporting self-adjusting computation for arbitrary nested-parallel algorithms. We prove bounds on the cost of change propagation in the framework based on an appropriately defined distance metric. We have also implemented the framework and run experiments on a variety of benchmarks. A nested parallel program is one that is built from arbitrary sequential and parallel composition. A computation is defined recursively as either two computations that are composed in parallel (a fork), two that are composed sequentially, or the base case which is a sequential *strand*. Multiway forking can easily be implemented by nesting parallel compositions.

Our technique is to represent a computation by a dependency graph that is based on a *Series-Parallel tree* [58], (*SP tree*). An SP tree corresponds to the sequential and parallel composition of binary nested parallel programs—i.e., parallel composition consists of a $P$ node with two children (the left-right order does not matter), and sequential composition consists of an $S$ node with two children (here the order does matter). The leaves are sequential computations, and can just be modeled as leaf $S$ nodes. The SP tree represents the control dependencies in the program—i.e., that a particular strand needs to executed before another strand. We introduce R nodes to indicate data reads, which are used to track data dependencies between writes and reads—i.e., that a particular read depends on a particular write. Together we refer to the trees as *RSP trees.* The RSP tree of a computation allows propagating a change in a way that respects sequential control dependencies while allowing parallelism where there is no dependence. We prove that a parallel change propagation algorithm can propagate changes through the computation in a manner that is both efficient and scalable.

Programs written in our framework write their inputs and any nonlocal values that depend on them into "modifiable references", or *modifiables* for short, which track all reads to them and facilitate change propagation. Like previous work on sequential change propagation [4], we achieve our efficiency by restricting input programs to those which write to each modifiable exactly once. All race-free functional programs satisfy this restriction, but since local variables do not need to be tracked they are not bound by this restriction so the scope of programs amenable to our framework is wider than those which are purely functional.

Roughly speaking, given two executions of the same algorithm on different inputs, we define the *computation distance* to be the work that is performed by one but not the other (see Definition 16 for the full definition). We then show the following theorem that bounds the runtime of the change propagation algorithm as a function of computation distance.

**Theorem 46** (Efficiency)**.** Consider an algorithm $A$, two input states $I$ and $I'$, and their corresponding RSP trees $T$ and $T'$. Let $W_\Delta = \delta(T, T')$ denote the computation distance, $R_\Delta$ denote the number of affected reads, $s$ denote the span of $A$, and $h$ denote the maximum heights of $T$ and $T'$. Then, change propagation on $T$ with the dynamic update $(I, I')$ runs in $O(W_\Delta + R_\Delta \cdot h)$ work in expectation and $O(s \cdot h)$ span w.h.p.

We have implemented our techniques in a library for C++, which we call PSAC++[1] (Parallel Self-Adjusting Computation in C++). The library allows writing parallel self-adjusting programs by using several small annotations in a style similar to writing conventional parallel programs. Self-adjusting programs can respond to changes to their data by updating their output via the built-in change propagation. Our experiments with several applications show that parallel change propagation can handle a broad variety of batch changes to input data efficiently and in a scalable fashion. For small changes, parallel change propagation can yield very significant savings in work; such savings can amount to orders of magnitude of improvement. For larger changes, parallel change propagation may save some work, and still exploit parallelism, yielding improvements due to both reduction in work and an increase in scalability. We summarize the contributions of this chapter as follows:

- A general approach for parallel change propagation based on using RSP trees to safely propagate changes in the correct order while allowing parallelism in the propagation.
- Theoretical bounds on the work (sequential time) and span (parallel time) of our algorithms.
- An implementation as C++ library, with six example applications to study as benchmarks.
- Experimental results that confirm what is backed up by our theoretical analysis, that parallel change propagation is efficient for a range of applications.

## 10.1.1 Technical Overview

The idea of the change propagation algorithm is first to run an algorithm on some initial input while keeping a trace of reads and writes to "nonlocal" locations. This trace can be thought of as a write-read dependence graph, indicating what reads depend on what writes (also called a data dependence graph). Along with each read the trace also stores the code that was run on the value, and maintains some form of control ordering of the execution. When an input is updated at particular locations, the change propagation algorithm knows what read those locations and reruns them. This can cause new reads and writes that both update the trace, and create changes that have to propagated to their readers. Importantly, and one of the biggest challenges in change propagation, is that the reads that rerun have to do so in control order, otherwise they could use stale information. For example, if a read A, and a later a read B in program control order both need to be rerun, running B first might miss updates by A. Since A could do something different when rerun, the trace might not even know there will be a data dependence between them (A is now going to write to something B reads). This means that the topological order on the trace's data dependence graph in insufficient for safety, and that control dependencies also need to be considered.

In the sequential setting, the total order of all instructions is typically maintained using a dynamic list-maintenance data structure [53] keeping all reads in time order. The structure needs to be dynamic since during propagation new computation can be added, and old deleted, at arbitrary points in the ordering. During change-propagation, all reads that are affected by a write are placed in a priority queue prioritized by this order, and always processing the earliest first.

[1]Our code is publicly available at `https://github.com/cmuparlay/psac`

For our work on parallel change propagation, the broad idea is to organize the control dependencies of the program around the RSP tree. Unlike the sequential case, instead of having a total order of execution time, the RSP tree effectively keeps track of the parallel partial order of control dependencies among the strands. As with the sequential case, we also keep track of all write-read data dependencies. Unlike the sequential algorithm which uses a priority queue of time order, our algorithm instead uses the RSP tree itself to maintain the partial order among strands—and this allows running multiple tasks in parallel during the propagation.

The initial run builds the RSP tree. It stores on each read (R) node a closure to rerun if the value it read changes[2]. When the input is modified, the change-propagation algorithm identifies all readers of the changed values. We refer to these as the *affected readers*. Now, instead of adding them to a priority queue by sequential time order, the algorithm makes some markings in the RSP tree. In particular it starts at each affected reader and marks all ancestors in the RSP tree. It then traverses the RSP tree and using the marks finds readers that require and are safe to rerun, i.e., only descending if a node is marked. Whenever it gets to a $P$ node, the algorithm traverses down whichever children are marked (either left, right, or both in parallel), and whenever it gets to an internal $S$ node, it traverses down the left branch if it is marked, and then the right branch if it is marked. At an $S$ node the algorithm never goes down both branches simultaneously since that would be unsafe

When the traversal meets an affected reader, change propagation runs the closure associated with the reader and updates the resulting computation and its corresponding subtree of the RSP tree, possibly cascading new reads and writes and marking additional regions of the RSP tree for additional propagation. Once the marked regions of the tree have all been traversed, change propagation is complete and the computation will be fully up to date.

## 10.2 Framework

Our framework for parallel self-adjusting computation is built around a set of core primitives that are easy to integrate into existing algorithms. In this section, we describe these primitives and give an example algorithm for illustration.

**write**(dest: $\alpha$ **mod**, value : $\alpha$)
**alloc_mod**(T: **type**) : T **mod**
**read**(m : ($\alpha_1$ **mod**, ..., $\alpha_k$ **mod**), r : $\alpha_1 \times \ldots \times \alpha_k \mapsto ()$)
**par**(left_f : $() \mapsto ()$ , right_f : $() \mapsto ()$)
**run**($f : () \mapsto ()$) : $S$
**propagate**(root : $S$)

**Figure 10.1: Interface for Parallel Self-Adjusting Computation**

[2]A closure is a code pointer along with needed local variables.

## Modifiables

The primary mechanism by which computations are dynamised is through the use of *modifiable* variables. A modifiable variable, or modifiable for short, is either a value that is part of the input to the algorithm, or a nonlocal variable whose value depends on the value of another modifiable variable. Algorithms are dynamised by placing their inputs in modifiables, and ensuring that all nonlocal variables whose values depend on a modifiable are also placed in modifiables. When a modifiable is updated, our framework determines which values are affected by the resulting changes and propagates the appropriate updates.

Modifiables can be allocated either statically, i.e. before the computation is run, or dynamically, in which case their lifetime will be tied to the scope of the computation that allocated them. Writing to modifiables is achieved using the **write** operation. We require that each modifiable is written to at most once during each run of the computation, and that modifiables are not read before they are written. We also require that modifiables are only read from and written to by computations that are in the dynamic (nested) scope of the computation that allocated it.

## Read operations

To ensure that dependencies are tracked, modifiables must be read using the **read** operation. **read** reads the values of the given modifiables and invokes the given reader function with their current values as arguments.

## Parallelism

We support fork-join parallelism through a binary fork operation **par**, which takes two thunks (functions that take no arguments and return nothing) and executes them in parallel.

## Control

Computations are initiated with the **run** operation, which returns a handle to the computation (represented by the root of the RSP tree). After making changes to the input, changes are propagated using the **propagate** operation.

## Additional primitives

For performance, our practical implementation also supports an **alloc_array** operation and a corresponding **read_array** operation for allocating and reading arrays of modifiables. We also support a **parallel_for** primitive, which executes a given function over a range of values in parallel. We omit the details of these primitives.

## A note on randomness

We require that all algorithms implemented in our framework be deterministic. That is, given some input, if re-executed they must produce exactly the same output. It is still possible, and indeed we have several in our application examples, to implement randomized algorithms. To do so, the randomness must simply be pre-generated before executing the computation to ensure that, when re-executed, the same results will be obtained.

### Example

To illustrate our framework, we give an implementation of a parallel divide-and-conquer sum function. See Algorithm 22. Note that in our pseudocode, for readability, we denote reads using the syntax:

```
with read(mods...) as args... do
    f(args...)
```

As typical with self-adjusting computation, the code uses "destination passing", where SUM takes the modifiable in which the result should be written as an argument.

**Algorithm 22** Parallel self-adjusting sum

```
procedure SUM(A[lo...hi] : int mod array, res : int mod)
    if lo = hi - 1 then
        with read(A[lo]) as x do
            write(res, x)
    else
        local mid ← lo + (hi - lo) / 2
        local left_res ← alloc_mod(int)
        local right_res ← alloc_mod(int)
        par(function ⇒ SUM(A[lo...mid], left_res),
            function ⇒ SUM(A[mid...hi], right_res))
        with read(left_res, right_res) as x, y do
            write(res, x + y)
```

## 10.3 Change Propagation Algorithm

We use a variant of SP trees (see introduction) extended with read (R) nodes, which we call RSP trees. A read is tracked in the RSP tree by creating an R node as the left child of the current S node whenever a reader is executed. The reader code then executes in the subtree of the R node and the continuation (the code that executes directly after the read completes) proceeds in the sibling node. The full semantics for RSP trees is defined by the algorithms in this section.

Figure 10.2 shows an example RSP tree for the divide-and-conquer sum computation of Algorithm 22 on an input of size four. The four $R$ nodes lowest in the tree correspond to the reads of the input, which occur at the base case of the algorithm. The $R$ nodes higher in the tree are the reads of the results of the recursive calls. Although not depicted in this simple algorithm, reads may be nested, in which case read nodes may appear as descendants of other read nodes.

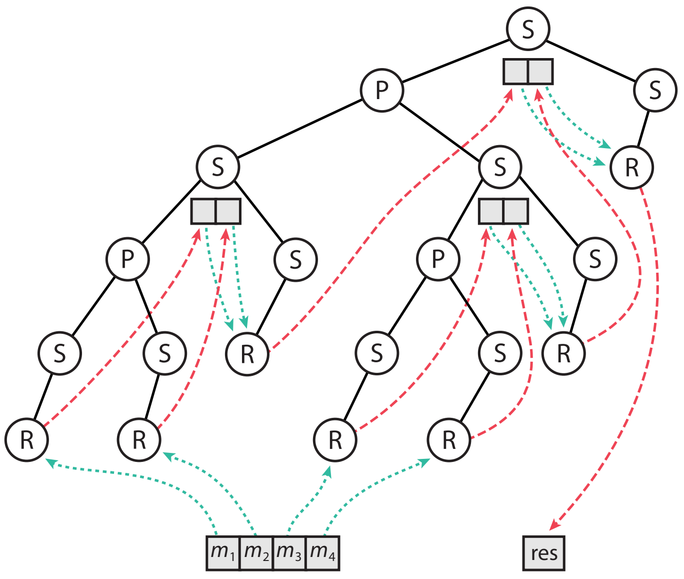


**Figure 10.2: The RSP tree of Algorithm 22 on an input of size four. Dynamically allocated modifiables are shown underneath the *S* node that allocated them. Writes and reads to/from modifiables are shown as red (long-dashed) and green (short-dashed) arrows respectively.**

Our framework facilitates self-adjusting computation by first building the RSP tree during the initial run of the static algorithm. To execute dynamic updates, when a modifiable is written to, all of the read nodes that read from it, and all of their ancestors in the RSP tree are marked as pending re-execution. Change propagation then simply consists in traversing the RSP tree, ignoring subtrees that are not marked, since no changes are present, and re-executing the marked readers. Note that this re-execution destroys the old portion of the RSP tree corresponding to the read and generates a new one, meaning that the old and new computations can be entirely structurally different. Additionally, such re-execution may also write to subsequent modifiables that were also read during the computation, so this process may mark additional nodes in the tree as pending re-execution, which will cause further propagation. The remainder of this section discusses the high-level implementation of these operations and the framework's primitives.

Pseudocode for the key components of our algorithm is shown in Algorithms 23–26. In the code, *current_scope* is a thread-local variable pointing to the current S node of the RSP tree that the code is running in.

Maintaining this notion of scope is important for two reasons. Most obviously, it ensures that the RSP tree can be constructed while the algorithm is ran. Less obviously, it also allows us to more efficiently allocate and collect dynamically allocated modifiables.

## Writing to modifiables

Pseudocode for writing to modifiables is given in Algorithm 23. If the new value differs from the old, all of the readers of that modifiable must be marked to trigger change propagation.

**Algorithm 23** Writing modifiables

```
1: procedure WRITE(dest : $\alpha$ mod, value: $\alpha$)
2:     if dest is unwritten or value $\neq$ dest.val then
3:         dest.val $\leftarrow$ value
4:         for each reader in dest.readers do in parallel
5:             reader.affected $\leftarrow$ true
6:             reader.mark()                                  // defined in Algorithm 26
```

## Reading modifiables

Pseudocode for the read operation and for handling R nodes is shown in Algorithm 24. R nodes consist of two specific fields, the list of modifiables that were read (mods), and the reader function that executes on the values of the modifiables (reader_f). When an R node is created or destroyed, it adds or removes itself from the corresponding lists of readers. When an R node executes its reader, the R node is used as the scope of the computation. This means that R nodes count as S nodes for the purpose of determining sequential dependencies.

**Algorithm 24** Reading modifiables

```
1: procedure READ(m : ($\alpha_1$ mod, ..., $\alpha_k$ mod), r : $\alpha_1 \times ... \times \alpha_k \mapsto ()$)
2:     local cur $\leftarrow$ current_scope
3:     cur.left $\leftarrow$ new $R$ node (m, r)                  // Calls R::create
4:     current_scope $\leftarrow$ cur.left
5:     cur.left.DO_READ()
6:     cur.right $\leftarrow$ new S node
7:     current_scope $\leftarrow$ cur.right
8: procedure $R$::CREATE(m, r)                           // Called on creating a new node
9:     this.mods $\leftarrow$ m
10:    this.reader_f $\leftarrow$ r
11:    for each mod v in mods do in parallel
12:        v.readers $\leftarrow$ v.readers $\cup$ {this}              // must be atomic!
13: procedure $R$::DO_READ
14:    local $m_1, ..., m_k \leftarrow$ this.mods
15:    local $v_1, ..., v_k \leftarrow m_1$.val, ..., $m_k$.val
16:    this.reader_f($v_1, ..., v_k$)
17: procedure $R$::DESTROY
18:    for each mod m in this.mods do
19:        m.readers $\leftarrow$ m.readers \ {this}              // must be atomic!
```

## Parallelism

The **par** function creates a *P* node as the left child of the current scope. The *P* node has two *S* nodes as children, which will correspond to the scope of the two computations that run in parallel. After completing the parallel computation, an *S* node is created as the right child of the current node to be the scope of any subsequent computation. The algorithm is shown in Algorithm 25.

**Algorithm 25** Parallelism

```
procedure PAR(left_f: $() \mapsto ()$ , right_f: $() \mapsto ()$)
    local cur ← current_scope
    cur.left ← new $P$ node
    cur.left.left ← new $S$ node
    cur.left.right ← new $S$ node
    in parallel do:
        { current_scope ← cur.left.left;  left_f() }
        { current_scope ← cur.left.right;  right_f() }
    cur.right ← new $S$ node
    current_scope ← cur.right
```

### Control operations

These are depicted in Algorithm 26. Run creates the root S node of the RSP tree and runs the computation from scratch. The propagate functions perform change propagation for each of the kind of RSP tree nodes. Note that the P version propagates in parallel, and the S version sequentially. When reaching a read node, the algorithm reruns the associated reader.

## 10.4 Analysis

In this section, we provide an analysis of our model to establish its correctness and prove bounds on the runtimes of our algorithms. Bounds in our analysis will depend on the work and span of the underlying algorithm, as well as the height of the generated RSP tree, which we note is at most the span of the algorithm, but can be much less. For all of our examples, it is at most $O(\log(n))$, even when the span of the algorithm is larger.

### 10.4.1 Setting

For our analysis, we will consider algorithms $A$ in our parallel self-adjusting framework, which can be thought of as functions which act on given inputs $I = \{(m_i, v_i)\}_i$, a set of modifiable-value pairs consisting of modifiables that $A$ will read, and their values. Executing $A(I)$ results in an output $(\tau, T)$, where $\tau$ is a set of modifiable-value pairs consisting of every modifiable written to by the execution of the algorithm, and the corresponding value. $T$ is the RSP tree of the computation, where each read node is annotated with the reader function and the values that were read. We define the domain of a set of pairs $X$ by $dom(X) = \{m : (m, v) \in X\}$. Due to the write-once restriction, note that in a valid execution, we must have $dom(I) \cap dom(\tau) = \emptyset$.

We can then define a *dynamic update* $\Delta = (I, I')$ to be a pair of input states with $I \neq I'$, denoting that the input is changed from $I$ to $I'$, which may involve changing the values of

**Algorithm 26** Control operations

```
procedure RUN($f : () \mapsto ()$) : $S$
    local root ← new $S$ node
    current_scope ← root
    $f$()
    return root
procedure PROPAGATE(root : $S$)
    if root.marked then root.propagate()
procedure NODE::MARK
    this.marked ← true
    if this.parent $\neq \bot \wedge \neg$ this.parent.marked then
        this.parent.mark()
procedure $S$::PROPAGATE
    if this.left $\neq \bot \wedge$ this.left.marked then
        this.left.propagate()
    if this.right $\neq \bot \wedge$ this.right.marked then
        this.right.propagate()
    this.marked ← false
procedure $P$::PROPAGATE
    if this.left.marked $\wedge$ this.right.marked then
        in parallel do:
            this.left.propagate()
            this.right.propagate()
    else if this.left.marked then this.left.propagate()
    else this.right.propagate()
    this.marked ← false
procedure $R$::PROPAGATE
    if this.affected then
        this.left ← $\bot$
        this.right ← $\bot$
        this.DO_READ()
        this.affected ← false
    else
        if this.left $\neq \bot \wedge$ this.left.marked then
            this.left.propagate()
        if this.right $\neq \bot \wedge$ this.right.marked then
            this.right.propagate()
    this.marked ← false
```

modifiables in $I$, adding new modifiables that were not read the first time, and removing modifiables that are no longer read. We can then think of change propagation as taking an RSP tree $T$ and a dynamic update $(I, I')$, and outputting a set of writes $\tau$ and an updated RSP tree $T'$. We can now define the notion of *affected readers*, which, intuitively, when applying an algorithm to two different inputs, are readers that exist in both versions of the computation but read different values, i.e. they are the frontiers at which the computations diverge.

**Definition 15** (Affected readers)**.** Consider an algorithm $A$, two input states $I$ and $I'$, and their corresponding RSP trees $T$ and $T'$, i.e., $A(I) = (\tau, T)$ and $A(I') = (\tau', T')$ for some $\tau$ and $\tau'$. We say that a read node is *subsumed* by another read node in the same tree if the first one is a descendant of the second one, i.e., the first one was created while executing the second one's computation. Given two read nodes $v \in T$ and $v' \in T'$, we say that they are *cognates* if the paths in $T$ and $T'$ to $v$ and $v'$ are the same, that is, the path branches left or right at the same time and have the same labels. We call a pair of cognate read nodes *affected* if they read different values, and are not subsumed by another such node.

Note that this definition of affected node makes sense because of the fact that computations in our framework are deterministic, and hence, the only place at which a computation can begin to differ is at a read node that reads different values than last time. We now introduce the notion of *computation distance*. The computation distance models the amount of work required to re-execute the affected readers. In essence, it is the minimum amount of work required to update the computation assuming absolutely no overhead.

**Definition 16.** Consider an algorithm $A$, two input states $I$ and $I'$, and their corresponding RSP trees $T$ and $T'$. Define the cost of a read node to be the work performed by its reader function[a]. The computation distance between the executions of $A$ on the inputs $I$ and $I'$ is defined as the sum of the costs of the affected read nodes in $T$ and $T'$. More formally, if we denote by $l(T)$, $v(T)$, $w(T)$, the RSP label of the root node of $T$, the values read by the read node at the root of $T$, and the work performed by the reader function of the read node at the root of $T$, we can define the computation distance recursively starting at the root of the trees as follows.

$$\delta(T, T') = \begin{cases} w(T) + w(T') & \text{if } l(T) = R \wedge v(T) \neq v(T'), \\ \sum_{i=1}^{k} \delta(T_i, T_i') & \text{otherwise,} \end{cases}$$

where $T_i$ denotes the $i^{\text{th}}$ subtree of $T$.

[a]The work performed by the reader function is considered to be the work that it would perform when executed without self-adjusting computation, i.e., assuming that reads and writes take constant time.

Observe that due to determinism, the definition of computation distance will only consider cognate nodes $T, T'$ which must have the same number of children/subtrees. We are now ready to state the correctness theorem of our framework.

**Theorem 47** (Correctness)**.** Consider an algorithm $A$ and an input state $I$ where $A(I) = (\tau, T)$. Let $\Delta = (I, I')$, where $A(I') = (\tau', T')$, denote a dynamic update to the input. Then, applying change propagation to the RSP tree $T$ with dynamic update $\Delta$ yields

1. writes $\tau''$ such that $\tau' \subseteq \tau'' \cup \{(m, v) \in \tau \mid m \notin dom(\tau'')\}$,
2. the RSP tree $T'$.

*Proof.* Proving the correctness of change propagation essentially relies on establishing two facts: that it visits and re-executes all affected read nodes, and that re-executing just the affected read nodes is sufficient.

The fact that all affected read nodes are re-executed can be established inductively on the sequential dependencies of the affected readers. The earliest affected reader must read a modifiable that exists in $I$ and $I'$ but has a different value, and hence will be marked in the RSP tree and will be re-executed. An affected reader that has had all of its sequential dependencies re-executed must be marked since it either reads a modifiable that exists in $I$ and $I'$ but has a different value, or it reads a modifiable that is written earlier in the computation. In the second case, since computations are deterministic, the modifiable must be written by a reader whose input has changed, and hence is an affected reader which has been re-executed.

Establishing that re-executing all affected readers writes to all modifiables whose values in $\tau'$ are different than in $\tau$ follows from determinism and the write-once restriction. Determinism implies that all differing writes must occur inside an affected reader, and the write-once restriction ensures that these writes exist in $\tau'$. Lastly, the fact that the RSP tree is updated to $T'$ follows from determinism. □

We now prove our efficiency theorem that bounds the cost of change propagation in terms of the computation distance.

*Proof of Theorem 46.* Since there are $O(R_\Delta)$ affected readers, it costs at most $O(R_\Delta \cdot h)$ work to traverse the RSP tree to reach each of them. The work required to re-execute all affected readers and destroy their old RSP subtrees is $O(W_\Delta)$ by definition, plus any overhead encountered from maintaining modifiables' reader sets and marking ancestors when performing the **write** primitive. We can argue that these overheads can be reduced to constant or amortized. To reduce the maintenance of reader sets to constant overhead, the algorithm can maintain each modifiable's reader set as a hashtable. To avoid issues of concurrency and resizing, insertions and deletions (Lines 12 and 19 in Algorithm 24) can be deferred and performed in batch after change propagation is complete. The overhead of **write** can be amortized by noticing that for all marked nodes, they will either be traversed by change propagation, or destroyed by a re-execution.

Lastly, we consider how these overheads affect the span of the algorithm. Each **write** operation takes up to $h$ time, and each **read** may require a hashtable operation that takes up to $\log(r)$ time w.h.p., where $r$ is the size of the reader set. However, $h \geq \log(r)$ and hence the overhead is at most $h$ per operation w.h.p., leading to a total span of $O(s \cdot h)$ w.h.p. □

It is worth noting that the randomness in our bounds comes purely from the use of hashtables to store the reader sets. For algorithms in which each modifiable has only a constant number of readers, which is very often the case, the bounds can be made deterministic.

## 10.4.2 Analyzing the Computation Distance of Algorithms

To obtain bounds for dynamic updates on particular algorithms implemented in our framework, it suffices to analyze the number of affected reads and the computation distance for the desired class of updates (and the span of the algorithm which is usually already known). Here, we will sketch an analysis of the sum algorithm from Algorithm 22.

**Theorem 48.** Consider Algorithm 22 on an input $A$ of $n$ modifiables, and a dynamic update in which the values of $k$ modifiables are changed. The number of affected reads and the computation distance induced by such an update is $O\left(k\log\left(1+\frac{n}{k}\right)\right)$.

*Proof.* Note that the algorithm performs $\log(n)$ levels of recursion. We count separately the number of affected reads that occur during the first $\log(k)$ levels and those that occur after. During the first $\log(k)$ levels, since the algorithm performs binary recursion, there can be no more than $O(2^{\log(k)}) = O(k)$ reads in total, affected or not. The $k$ updated modifiables will affect $k$ of the base-case reads on Line 3. The corresponding writes on Line 4 then affect up to $k$ reads on Line 10 from the calling functions. The writes on Line 11 then affect up to $k$ reads from their callers, and so on. The final $\log\left(\frac{n}{k}\right)$ levels of recursion therefore account for at most $k\log\left(\frac{n}{k}\right)$ affected readers. Therefore, in total, there can be at most $O\left(k+k\log\left(1+\frac{n}{k}\right)\right) = O\left(k\log\left(1+\frac{n}{k}\right)\right)$ affected readers, each of which performs $O(1)$ work. $\square$

In Chapter 4, we analyzed the computation distance of tree contraction, which also appears in our benchmarks, in the round-synchronous model. The round-synchronous model can be implemented in the framework of this chapter, and hence the bounds also apply to the computation distance here.

### Overhead of self-adjusting computation

In addition to the cost of dynamic updates, we can also discuss the overhead of the initial computation. Note that each node in the RSP tree corresponds to at least one primitive operation, and hence the cost of the building the tree and later destroying it can be charged to the computation. Then, just as in change propagation, the overhead of the **read** and **write** primitives are either constant, or can be amortized (see the Proof of Theorem 46), leading to constant amortized overhead.

Lastly, we remark on the memory usage. The two sources of memory overhead come from the RSP tree and modifiables. In the worst case, the size of the RSP tree is proportional to the work of the algorithm. However, for any sensible algorithm, both strands of any parallel fork will contain at least one read (if they do not, the parallel fork was unnecessary). Therefore, under this assumption, the size of the SP tree is proportional to the number of reads in the algorithm. Since the memory overhead of modifiables (their reader sets) is also proportional to the number of reads, the additional memory overhead is just proportional to the number of reads.

### Work-efficiency of change propagation

By definition, re-executing a set of affected readers of computation distance $W_\Delta$ must take $O(W_\Delta)$ work. Based on Theorem 46, we therefore consider the work overhead of change propagation to be $R_\Delta \cdot h$. This means that if $W_\Delta \geq R_\Delta \cdot h$, i.e, each affected reader performs at least $h$ work on average, then change propagation essentially has just constant-time overhead. In practice, this suggests that good granularity control is important for writing efficient self-adjusting algorithms.

### Comparison to sequential self-adjusting computation

The best sequential algorithms for self-adjusting computation [4] can propagate an update of computation distance $W_\Delta$ in $O(W_\Delta \log(W_\Delta))$ work. Compared to our bounds, which are at most $O(W_\Delta \cdot h)$, the difference is a $\log(W_\Delta)$ versus $h$. Given a parallel algorithm on input size $n$ with $\text{polylog}(n)$ span, we have $h \leq \text{polylog}(n)$. However, often, and for every example we studied, $h$ is just $\log(n)$, even for algorithms with larger span. Therefore at worst, our algorithm is $O(\text{polylog}(n))$ slower than the best sequential algorithm, but in the common case, just $O(\log(n)/\log(W_\Delta))$ slower.

## 10.5 Implementation

To study its practical performance, we implemented our framework as a library for C++. For parallelism, we use the work-stealing scheduler from the Parlay library [28]. For memory allocation, we use jemalloc [56] in addition to Parlay's pool-based memory allocator. In this section, we discuss some of the interesting aspects of the implementation of the system, and note some useful optimizations.

### 10.5.1 Reader Set Implementation

One interesting part of the system is handling the reader sets of modifiables. Since multiple concurrently executing threads may read the same modifiable, it is important that modifications to this set are thread-safe. To obtain our theoretical bounds, we describe the algorithm using a hashtable. In practice, however, we observe that the majority of modifiables in self-adjusting algorithms have just a small constant number of readers, often just one. We therefore implement the reader sets with a hybrid data structure that stores a single reader inline with no heap allocation when there is only one reader. When the number of readers becomes more than one, the reader set atomically converts itself into a linked data structure. We used a linked list for algorithms with small reader sets, and a randomized binary search tree for algorithms with larger reader sets.

Our binary search tree uses the hashes of the addresses of the reader nodes as the random keys. To insert a new reader into the tree, our algorithms attempts to insert it into the appropriate leaf of the tree using an atomic compare-and-swap (CAS) operation. If the CAS succeeds, the insertion is successful. Otherwise, note that the correct location for the key must be a child of the node that instead won the CAS, so our algorithm proceeds down and tries again. To simplify deletions, rather than deleting from the tree eagerly, nodes that need to be removed are simply marked as dead, and removed during the next traversal.

To ensure thread safety, we have to make sure that operations on the reader sets that might race are safe. Note that insertions correspond to reads, traversals correspond to writes, and deletions correspond to the cleanup of destroyed subtrees after a computation is re-executed. Insertions will therefore never race with traversals since reads and writes to the same modifiable can not race in a valid self-adjusting program. Deletions may however race with traversals or insertions since the cleanup of an RSP subtree may take place while another re-computation is occurring. One way to mitigate any potential problems is to defer all destructions of RSP subtrees until a later garbage-collection phase, rather than performing them during change propagation. Lastly, multiple traversals can not race due to the write-once condition, but multiple insertions or deletions can. Our algorithm is safe with respect to concurrent insertions, and our lazy deletion strategy makes concurrent deletions safe.

### 10.5.2 Garbage Collection

Rather than eagerly deleting subtrees of the RSP tree when a reader is re-executed, we instead move such subtrees off to a garbage pile which we collect after performing change propagation. This simplifies the destruction of subtrees since doing so naively can very easily lead to race conditions, such as those discussed in the reader set implementation. Performing delayed garbage collection also has the benefit of improving the responsiveness of change propagation, as the result of the update can be made visible to the user before the garbage collector is run.

### 10.5.3 Supporting Dynamically-Sized Inputs

Modifiables give us the ability to easily write algorithms that support updating the *values* in the input and propagating the results. In many situations, we also want to support the ability to add/remove elements to/from the input. In sequential self-adjusting computation, this is achieved by using linked lists to represent the input. In the parallel setting, we can achieve similar results by representing the input as a balanced binary tree. The trick is to use modifiables to represent the parent/children relationships in the tree so that if a new element is inserted, this will cause an update of a child modifiable and allow change propagation to update the computation with the new element.

## 10.6 Benchmarks and Evaluation

In this section, we evaluate the practical performance of our system. We implemented six benchmarks, exhibiting a range of different characteristics and providing different insights into the quality of the proposed algorithms.

### Experimental setup

We ran our experiments on a 4-socket AMD machine with 32 physical cores in total, each running at 2.4 GHz, with 2-way hyperthreading, a 6MB L3 cache per socket, and 200 GB of main memory. All of our code was compiled using Clang 9 with optimization level `-O3`. Each experiment was run using 1 – 64 worker threads in increasing powers of two. We used the Google Benchmark C++ library to measure the speed (in real/wall time) of each benchmark. We run each benchmark ten times and take the average running time.

### Benchmark setup

Each benchmark consists of four parts. First, we run a static sequential program and a static parallel program that implement the same algorithm to the self-adjusting one but without any overhead from self-adjusting computation. We then benchmark the parallel self-adjusting program, both on its initial computation, and on performing dynamic updates with change propagation. For each of the examples, we use varying batch update sizes to measure the effect that batch sizes have on the amount of parallelism exhibited by the update, and the amount of work required to propagate it. We do not include the time taken to perform garbage collection in the measurements.

### Reporting of results

For each benchmark, we provide numerical results in Tables 10.1–10.6, which show the running times of the static sequential algorithm (Seq), parallel static algorithm (Parallel Static), the initial computation of the self-adjusting algorithm (PSAC Compute), and the dynamic updates (PSAC Update), for 1 processor (1), 32 processors (32), and 32 processors with hyperthreading (32ht). For each of the parallel algorithms, we compute the self-speedup (SU), which is the relative improvement of the 32 or 32ht performance (whichever is better) compared to the 1 processor performance. For each example, we measure the performance for some fixed input size $n$ and varying batch update sizes $k$.

For the dynamic updates, we measure work savings (WS), which is the relative improvement of their 1 processor performance compared to the static sequential algorithm. Finally, we report the total speedup, which is the relative performance of the dynamic updates with 32 or 32ht processors compared to the static sequential algorithm (equivalently, the product of the speedup and the work savings). This allows us to measure separately, the benefits due to parallelism (the SU), the benefits due to dynamism (the WS), and their total combined benefit (Total).

| Static Algorithm | | | | | | |
|---|---|---|---|---|---|---|
| **n** | **1t** | **32t** | **32ht** | **SU** | **Seq Baseline** | |
| $10^6$ | 36.21s | 1.19s | 908ms | 39.84 | 36.72s | |
| **PSAC Initial Run** | | | | | | |
| **n** | **1t** | **32t** | **32ht** | **SU** | | |
| $10^6$ | 36.89s | 1.20s | 750ms | 49.16 | | |
| **PSAC Dynamic Update** | | | | | | |
| **k** | **1t** | **32t** | **32ht** | **SU** | **WS** | **T** |
| $10^0$ | 44us | 45us | 47us | 0.98 | 819.6k | 800.4k |
| $10^1$ | 445us | 90us | 112us | 4.93 | 82.42k | 406.7k |
| $10^2$ | 4.36ms | 265us | 208us | 20.92 | 8.41k | 176.0k |
| $10^3$ | 44ms | 2.03ms | 1.20ms | 36.97 | 827.9 | 30.61k |
| $10^4$ | 436ms | 19ms | 11ms | 37.46 | 84.18 | 3.15k |
| $10^5$ | 4.04s | 167ms | 118ms | 34.08 | 9.10 | 310.2 |
| $10^6$ | 37.67s | 1.81s | 1.21s | 31.04 | 0.97 | 30.26 |

**Table 10.1: Benchmark results for Spellcheck.**

### Applications

We implemented the following benchmarks.

- **Spellcheck:** Computes the minimum edit distance of a set of a million strings to a target.
- **Raytracer:** Renders a $2000 \times 2000$ pixel scene consisting of three reflective balls using a simple ray tracing method.
- **String Hash:** Computes the Rabin-Karp fingerprint (hash) of a one-hundred-million character string.
- **Dynamic Sequence:** Computes a list contraction of a linked list of length one million.
- **Dynamic Trees:** Computes a tree contraction of a tree on one million nodes.
- **Filter:** Filters the elements of a BST with respect to a given predicate, returning a new BST.

## 10.6.1 Results

The results of our main experiments are depicted in Tables 10.1–10.6, and some further experiments are shown in Tables 10.7–10.8.

### The initial run

We are interested in the overhead of the initial run. This is the ratio of the runtime of the self-adjusting algorithm compared to the sequential baseline. Prior work on sequential self-adjusting computation [9] observed overheads ranging from 1.9 to 29.

The overhead of the initial run varies with the problem and the granularity of the work performed by the readers. For the spellcheck benchmark, the overhead is negligible since each of the readers performs a relatively expensive edit distance computation, completely hiding the overhead of the framework. For algorithms with smaller granularity, such as the

| Static Algorithm | | | | | | |
|---|---|---|---|---|---|---|
| **n** | **1t** | **32t** | **32ht** | **SU** | **Seq Baseline** | |
| - | 2.46s | 80ms | 49ms | 49.32 | 2.54s | |
| **PSAC Initial Run** | | | | | | |
| **n** | **1t** | **32t** | **32ht** | **SU** | | |
| - | 13.65s | 520ms | 323ms | 42.24 | | |
| **PSAC Dynamic Update** | | | | | | |
| **k** | **1t** | **32t** | **32ht** | **SU** | **WS** | **T** |
| - | 112ms | 8.66ms | 7.47ms | 15.08 | 26.44 | 398.8 |

**Table 10.2: Benchmark results for Raytracer.**

| Static Algorithm | | | | | | |
|---|---|---|---|---|---|---|
| **n** | **1t** | **32t** | **32ht** | **SU** | **Seq Baseline** | |
| $10^8$ | 1.86s | 58ms | 31ms | 58.34 | 1.72s | |
| **PSAC Initial Run** | | | | | | |
| **n** | **1t** | **32t** | **32ht** | **SU** | | |
| $10^8$ | 3.05s | 96ms | 61ms | 49.81 | | |
| **PSAC Dynamic Update** | | | | | | |
| **k** | **1t** | **32t** | **32ht** | **SU** | **WS** | **T** |
| $10^0$ | 14us | 16us | 16us | 0.90 | 118.8k | 107.1k |
| $10^1$ | 134us | 66us | 74us | 2.02 | 12.81k | 25.86k |
| $10^2$ | 1.26ms | 162us | 122us | 10.35 | 1.36k | 14.10k |
| $10^3$ | 12ms | 717us | 512us | 25.17 | 133.3 | 3.36k |
| $10^4$ | 103ms | 4.63ms | 3.11ms | 33.16 | 16.68 | 553.0 |
| $10^5$ | 621ms | 26ms | 18ms | 34.20 | 2.77 | 94.76 |
| $10^6$ | 2.49s | 108ms | 66ms | 37.27 | 0.69 | 25.73 |
| $10^7$ | 5.21s | 213ms | 132ms | 39.30 | 0.33 | 12.99 |
| $10^8$ | 19.47s | 709ms | 472ms | 41.24 | 0.09 | 3.65 |

**Table 10.3: Benchmark results for String Hash.**

| Static Algorithm | | | | | | |
|---|---|---|---|---|---|---|
| **n** | **1t** | **32t** | **32ht** | **SU** | **Seq Baseline** | |
| $10^6$ | 647ms | 70ms | 73ms | 8.77 | 586ms | |
| **PSAC Initial Run** | | | | | | |
| **n** | **1t** | **32t** | **32ht** | **SU** | | |
| $10^6$ | 4.32s | 219ms | 464ms | 19.66 | | |
| **PSAC Dynamic Update** | | | | | | |
| **k** | **1t** | **32t** | **32ht** | **SU** | **WS** | **T** |
| $10^0$ | 761us | 629us | 736us | 1.21 | 770.2 | 931.0 |
| $10^1$ | 5.58ms | 1.65ms | 1.97ms | 3.38 | 105.2 | 355.2 |
| $10^2$ | 31ms | 3.31ms | 2.95ms | 10.69 | 18.62 | 199.0 |
| $10^3$ | 201ms | 14ms | 10ms | 18.53 | 2.91 | 53.94 |
| $10^4$ | 1.26s | 74ms | 53ms | 23.33 | 0.47 | 10.89 |
| $10^5$ | 5.11s | 263ms | 195ms | 26.15 | 0.11 | 3.00 |
| $10^6$ | 8.45s | 624ms | 492ms | 17.14 | 0.07 | 1.19 |

**Table 10.4: Benchmark results for Dynamic Sequence.**

| Static Algorithm | | | | | | |
|---|---|---|---|---|---|---|
| **n** | **1t** | **32t** | **32ht** | **SU** | **Seq Baseline** | |
| $10^6$ | 915ms | 85ms | 66ms | 13.85 | 824ms | |
| **PSAC Initial Run** | | | | | | |
| **n** | **1t** | **32t** | **32ht** | **SU** | | |
| $10^6$ | 4.85s | 242ms | 689ms | 20.02 | | |
| **PSAC Dynamic Update** | | | | | | |
| **k** | **1t** | **32t** | **32ht** | **SU** | **WS** | **T** |
| $10^0$ | 698us | 584us | 672us | 1.19 | 1.18k | 1.41k |
| $10^1$ | 3.28ms | 1.04ms | 1.23ms | 3.14 | 251.7 | 789.4 |
| $10^2$ | 24ms | 2.29ms | 2.18ms | 11.03 | 34.23 | 377.7 |
| $10^3$ | 210ms | 12ms | 10ms | 20.46 | 3.93 | 80.33 |
| $10^4$ | 1.47s | 79ms | 60ms | 24.33 | 0.56 | 13.68 |
| $10^5$ | 5.19s | 254ms | 173ms | 29.85 | 0.16 | 4.74 |
| $10^6$ | 8.59s | 428ms | 306ms | 28.00 | 0.10 | 2.69 |

**Table 10.5: Benchmark results for Dynamic Trees.**

| Static Algorithm | | | | | | |
|---|---|---|---|---|---|---|
| **n** | **1t** | **32t** | **32ht** | **SU** | **Seq Baseline** | |
| $10^7$ | 361ms | 17ms | 15ms | 23.09 | 262ms | |
| **PSAC Initial Run** | | | | | | |
| **n** | **1t** | **32t** | **32ht** | **SU** | | |
| $10^7$ | 630ms | 35ms | 31ms | 20.27 | | |
| **PSAC Dynamic Update** | | | | | | |
| **k** | **1t** | **32t** | **32ht** | **SU** | **WS** | **T** |
| $10^0$ | 36us | 109us | 128us | 0.33 | 13.20k | 4.36k |
| $10^1$ | 275us | 242us | 298us | 1.14 | 1.73k | 1.96k |
| $10^2$ | 2.74ms | 1.39ms | 1.25ms | 2.19 | 173.6 | 380.9 |
| $10^3$ | 25ms | 3.73ms | 3.69ms | 6.95 | 18.56 | 129.1 |
| $10^4$ | 143ms | 9.96ms | 8.18ms | 17.50 | 3.32 | 58.19 |
| $10^5$ | 543ms | 31ms | 23ms | 22.97 | 0.88 | 20.13 |
| $10^6$ | 1.04s | 80ms | 55ms | 18.76 | 0.46 | 8.61 |
| $10^7$ | 2.12s | 198ms | 159ms | 13.28 | 0.22 | 2.98 |

**Table 10.6: Benchmark results for Filter.**

Rabin-Karp benchmark, we observe work overheads of around 1.7. The filter algorithm also uses a similar granularity, and hence experiences similarly low overhead.

On the other hand, the raytracing algorithm involves modifiables with a large number of readers, so the work overhead is higher, at around a factor of 4.6. The list contraction and tree contraction benchmarks both perform $O(\log(n))$ rounds of computation, with dependency chains spanning across them, and hence have larger overheads of 5.8 and 7.3.

## Work savings

Work savings measure the relative improvement in runtime from using self-adjusting computation to perform an update compared to running the algorithm from scratch. As was the case for the work overhead, the work savings are dependent on the granularity of the work performed by the readers. Of course, the work savings are also heavily dependent on the size of the update relative to the size of the entire input. For small updates, the work savings range from 770 when updating one element of one million elements (list contraction) to 819k when updating one of one million strings (edit distance). The work savings for the raytracer benchmark are also very encouraging. For the given dynamic update, a total of 6.25% of the image needed to be updated, but the change propagation algorithm performed approximately 4% of the work required to recompute from scratch. For benchmarks with varying update sizes, the work savings gradually decrease as the update size increases.

It is interesting to look at the crossover point where from-scratch execution becomes more efficient than change propagation. For benchmarks like spellcheck, which perform heavy work at the reads, from-scratch execution does not outperform change propagation until updating the entire input. For modest-granularity benchmarks like hashing, from-scratch execution wins when the update size reaches $k = 10^6$ for sequential execution, or $k = 10^5$ for parallel execution, on an initial input of size $n = 10^8$. For tree contraction and list contraction, the crossover points occur at $k = 10^4$ out of $n = 10^6$ elements for both sequential and parallel execution. For filter, the crossover point occurs at roughly $k = 10^5$ out of $n = 10^7$ elements. In general, crossovers tend to occur a couple of orders of magnitude before the input size.

## Speedup

The initial runs of our algorithms all benefit, often substantially, from parallelism. On 32 hyperthreaded cores (64 threads), spellcheck and hashing experience parallel speedups of 49-50x. Raytracing achieves 42x, and list contraction, tree contraction, and filter 19-20x.

In addition to the initial run, updates also benefit from parallelism, particularly as the update sizes increases. Although there is little potential for parallelism for $k = 1$ updates, each benchmark exhibits speedups ranging from 22-39x for larger update sizes. At the crossover points, where change propagation is still competitive with from-scratch execution, speedups range from 22-34x. This further supports the notion that parallelism and self-adjusting computation are highly complementary methods. Self-adjusting computation leads to substantial savings for small update sizes, and parallelism provides strong speedups for larger update sizes. For moderate update sizes, both are effective and their benefits combine to yield good total performance improvements.

| Benchmark | Problem size | Input memory | Tree size | Memory |
|---|---|---|---|---|
| Spellcheck | $10^6$ strings | 80MB | 6M | 312MB |
| Raytracer | 8M pixels | 192MB | 24M | 1.3GB |
| String Hash | $10^8$ chars | 100MB | 9.4M | 462.5MB |
| Sequence | $10^6$ elems | 20MB | 33M | 1.96GB |
| Tree | $10^6$ nodes | 20MB | 18.8M | 1.3GB |
| Filter | $10^7$ elems | 200MB | 2.48M | 193MB |

**Table 10.7: RSP tree sizes and the amount of memory consumed by the RSP tree for each benchmark problem.**

### Tree size and memory usage

The RSP tree size, and hence the memory overhead of a self-adjusting algorithm depends heavily on the *granularity* at which the data is stored and processed. Table 10.7 shows the RSP tree sizes and memory usage of each of our benchmarks at their default granularity. For most algorithms, the memory overhead ranges between 1-7x the input size, which is consistent with prior work on sequential self-adjusting computation [9]. The outliers are our list contraction and tree contraction benchmarks, which use substantially more memory because they perform $O(n\log(n))$ work over $O(\log(n))$ rounds of computation, all of which is represented in the RSP tree, essentially leading to the tree size being an additional factor of $\log(n)$ larger than the input. A more sophisticated implementation of these algorithms could achieve $O(n)$ work by using *compaction* on the set of live nodes at each round. This could reduce their memory footprint, and would be interesting to explore in future work.

### The cost of garbage collection

When a self-adjusting computation is discarded, the resulting RSP tree must be destroyed, which also entails removing its read nodes from the reader sets of any modifiables that they read. Table 10.8 shows the runtime of garbage collection for each of the RSP trees for our six benchmark problems compared to the performance of the initial run. Note that for all of the problems other than Raytracer, garbage collection time is at least a factor of 500 less than the actual computation. For Raytracer, garbage collection is slightly more costly since it has many readers per modifiable and hence has to pay the cost of deletion from the reader sets. Even then, garbage collection takes less than 1% of the time of the initial run.

## 10.6.2 Additional Experiments

Finally, we perform two small experiments that measure the effect that data granularity and the sizes of reader sets have on the overall performance of self-adjusting computation.

| **Benchmark** | **Initial Run** | **Garbage Collection (32ht)** |
|---|---|---|
| Spellcheck | 750ms | 158us |
| Raytracer | 323ms | 1.99ms |
| String Hash | 61ms | 101us |
| Sequence | 464ms | 437us |
| Tree | 242ms | 493us |
| Filter | 31ms | 31us |

**Table 10.8: The cost of garbage collection for each benchmark problem. The initial run is the performance on 32 threads or 32 hyperthreads, whichever is better.**

| **Granularity** | **Memory** | **Run** | **Update** | **Run** | **Update** |
|---|---|---|---|---|---|
| | | $p = 1$ | $k = 1$ | $p = 32\text{ht}$ | $k = 10^4$ |
| 16 | 1.85GB | 6.8s | 17us | 158ms | 4ms |
| 32 | 925MB | 4.11s | 15us | 95ms | 3.64ms |
| 64 | 462.5MB | 3.03s | 14us | 62ms | 3.12ms |
| 128 | 231.25MB | 2.5s | **14us** | 44ms | 3.04ms |
| 256 | 115.63MB | 2.3s | 15us | 34ms | 2.88ms |
| 512 | 57.81MB | 2.1s | 18us | 31ms | **2.83ms** |
| 1024 | 28.79MB | 2.0s | 24us | 29ms | 3.68ms |
| 2048 | 14.45MB | 2.0s | 39us | 28ms | 5.79ms |

**Table 10.9: Memory usage, initial run speed and update speed for various granularities in the hashing benchmark with size $n = 10^8$. Memory denotes memory used by the RSP tree.**

## Granularity tradeoffs

An important consideration when implementing parallel algorithms is careful control of granularity. This is perhaps even more true when implementing self-adjusting algorithms, since the granularity of the data and the functions executed by readers will directly influence the size of the RSP tree and the overhead of modifiables. A larger granularity will lead to lower work and memory overheads. The tradeoff, however, is that if the granularity is too large, updates will slow down, since more irrelevant information will be recomputed when a small piece of the input is updated. Here, we will explore the performance implications and tradeoffs that come from tuning the granularity of our string hashing benchmark. Results are shown in Table 10.9.

As expected, the memory usage and work overhead decreases monotonically as the granularity is increased. The more interesting aspect to look at is the update performance. Note that it is not necessarily the case that the smallest granularity will lead to the fastest updates. Although a smaller granularity means less redundant data is read and recomputed, it also leads to larger RSP trees, which might negate the benefit. The optimal granularity for update speed will therefore be one that balances the tradeoff between reading data and reducing the RSP tree size. For our string hashing benchmark, we observe that the optimal tradeoff occurs at a granularity of 128 characters for single character ($k = 1$) updates, and at 512 characters for larger ($k = 10^4$) updates. This phenomena is explainable by cache line

| # Mods | Readers/Mod | Run | Update |
|---|---|---|---|
| 1 | $10^6$ | 55.1ms | 191ms |
| 10 | $10^5$ | 48.9ms | 183ms |
| $10^2$ | $10^4$ | 47.2ms | 163ms |
| $10^3$ | $10^3$ | 46.5ms | 130ms |
| $10^4$ | $10^2$ | 45.1ms | 57.5ms |
| $10^5$ | 10 | 38.4ms | 57.0ms |
| $10^6$ | 1 | 27.8ms | 44.3ms |

**Table 10.10: Runtime of the reader-set size microbenchmark for varying numbers of input modifiables. Run denotes the runtime of the initial run, and Update denotes the runtime of a making a dynamic update to every modifiable.**

reads. Using a granularity of 512 will reduce the depth of recursion, and hence the number of cache misses by about 9, while reading a chunk of 512 characters corresponds to 8 cache lines, which balance out.

### Impact of reader-set size

Most self-adjusting computations, including all but one of our benchmarks, only have a constant number of readers (often just one) per modifiable. The raytracer benchmark illustrates the effect of having a large number of readers per modifiable, exhibiting a lesser speedup compared to most of the others. Here, we present a small microbenchmark that examines the performance impact of varying the number of readers of a modifiable. In Table 10.10, we depict the results of experiment in which $10^6$ workers in parallel each read from a random modifiable and write its value to a unique output destination. We vary the number of modifiables to observe the effect on performance.

We observe that for the initial run (the **Run** column), the performance is only marginally impacted as the number of readers per mod varies from 10 to $10^6$. The exception is when there is only one expected reader per mod, in which case the performance is up to twice as fast as the $10^6$ reader case. This is because of the optimization we perform in which modifiables with a single reader store that reader inline instead of allocating a linked data structure. We measure the effect on updates (the **Update** column) by changing the value of all of the modifiables and propagating the result. We observe that when varying from 10 to $10^6$ reads per modifiable, performance is at most a factor of four slower, or a factor of five slower compared to the one reader case.

## 10.7 Discussion

In this work, we designed, analyzed, and implemented a system for parallel self-adjusting computation. We showed that a small set of primitives is sufficient to express self-adjusting programs that can exploit arbitrary nested parallelism. Compared to previous work, this is the first such system with theoretical bounds on the runtime of the updates. Our experiments show that the system is capable of producing dynamic algorithms that both vastly outperform their static counterparts when performing small to moderately sized updates, and scale well on multiprocessor machines. There are several interesting avenues down which future work could lead.

Our implementation of self-adjusting computation comes as a library for `C++`, which boasts the advantage that it is very portable and can be compiled with many compilers on many different systems. We believe that implementing direct compiler support for our primitives could, however, decrease the overhead introduced by dependency tracking and SP tree construction, yielding more efficient algorithms. Direct compiler support for our primitives would also make it easier to develop static analysis tools for detecting programmer bugs in self-adjusting applications.

Sequential self-adjusting computation has been extended to support imperative computations, i.e. computations in which a variable can be written to multiple times sequentially. Although our algorithms only support write-once computations, we believe that they can be extended to efficiently support multi-write computations, as the SP tree structure should contain sufficient information to correctly order successive writes. This direction has begun being explored in Baweja's masters thesis [19].

# Chapter 11
# Conclusion

## Conclusion

With the scale and dynamism of today's massive datasets and the fast-accelerating development of parallel hardware, the study of algorithms that can process these datasets effectively has seen a huge increase in investment from both the theoreticians and practitioners of parallel algorithms in recent years. In this thesis, we argued that *work-efficient parallel batch-dynamic algorithms* are a powerful tool for designing algorithms and systems for processing the datasets of today and beyond. We have contributed to the foundation of work-efficient parallel batch-dynamic algorithms by designing algorithms for key graph problems, such as dynamic trees (Chapters 3—5), connectivity (Chapter 6), and minimum spanning trees (Chapter 7). These algorithms have already proved to be useful and applicable, both by us and by other researchers. Our work on parallel RC-Trees was shown to be a useful ingredient in our incremental MST algorithm (Chapter 7), and in obtaining the first ever work-efficient parallel algorithm for minimum cuts (Chapter 9). They have also been applied by other researchers as an ingredient to implement the first near-linear work parallel algorithm for depth-first search [74]. Our batch-dynamic connectivity algorithm similarly was the starting point for the first polylogarithmic-work batch-dynamic MST algorithm of Tseng et al. [175].

Although not a batch-dynamic algorithm itself, our work-efficient parallel minimum cut algorithm (Chapter 9) demonstrates how parallel batch-dynamic algorithms can be broadly applied as subroutines in other non-dynamic algorithms with great success. Obtaining a work-efficient algorithm for minimum cut was an open problem for decades, with all known parallel algorithms incurring some work overhead in their processing of the 2-respecting cut problem. It was parallel batch-dynamic algorithms that broke this barrier and allowed a parallel algorithm to catch up to the efficiency of the best sequential algorithm. The key ingredient was our framework for parallel evaluation of batched mixed operations on trees, which is itself a general enough tool that it might find other applications too.

Finally, we have shown that there is hope for non-theoreticians to reap the benefits of parallel batch-dynamic algorithms by designing the first theoretically efficient system for parallel self-adjusting computation (Chapter 10). In this system, programmers implement simple static parallel algorithms which are automatically converted into dynamic algorithms which automatically support batching if updates are made in batches. We show experimentally that our system enables static algorithms to produce updated results over large datasets significantly faster than from-scratch execution, saving both work and parallel time.

# Bibliography

[1] Umut A. Acar. *Self-Adjusting Computation.* PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005. 1.2.4, 10.1

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002. 1.2.4, 10.1

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003. 1.2.4, 10.1

[4] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004. 1.2.1, 1.2.4, 3.1.1, 4.1, 10.1, 10.4.2

[5] Umut A Acar, Guy E Blelloch, and Jorge L Vittes. An experimental analysis of change propagation in dynamic trees. In *Algorithm Engineering and Experiments (ALENEX)*, 2005. 1.2.1, 1.2.1, 3.1.1, 3.2, 3.3, 3.4.4, 3.5, 3.5.3, 3.5.6, 4.1, 8.2

[6] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic algorithms via self-adjusting computation. In *European Symposium on Algorithms (ESA)*, 2006. 10.1

[7] Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007. 10.1

[8] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust kinetic convex hulls in 3D. In *European Symposium on Algorithms (ESA)*, 2008. 10.1

[9] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):1–53, 2009. 10.1, 10.6.1

[10] Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Türkoğlu. Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*, 2010. 10.1

[11] Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry (SoCG)*, 2010. 10.1

[12] Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Parallelism in dynamic well-spaced point sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011. 10.1

[13] Umut A. Acar, Benoît Hudson, and Duru Türkoğlu. Kinetic mesh-refinement in 2D. In *Symposium on Computational Geometry (SoCG)*, 2011. 10.1

[14] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012. 1.2.2, 7.4.2, 7.4.3, 7.4.4, 7.4.6, 7.4.6

[15] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. 7.1

[16] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms (TALG)*, 1(2):243–264, 2005. 1.2.1, 1.2.1, 3.1.1, 3.1.1, 3.2.2, 3.5, 8.2

[17] Alexandr Andoni, Clifford Stein, Zhao Song, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2018. 1.2.2

[18] Baruch Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *International Conference on Parallel Processing (ICPP)*, 1983. 6.1

[19] Anubhav Baweja. *Applications and Extensions of Parallel Self-adjusting Computation.* PhD thesis, Carnegie Mellon University, 2021. 10.7

[20] András A. Benczúr and David R. Karger. Approximating $s$-$t$ minimum cuts in $\tilde{O}(n^2)$ time. In *ACM Symposium on Theory of Computing (STOC)*, 1996. 7.4.6

[21] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994. 4, 3.5.3

[22] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing (SoCC)*, 2011. 10.1

[23] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011. 1.1

[24] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B Brandenburg, and Rodrigo Rodrigues. iThreads: A threading library for parallel incremental computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. 10.1

[25] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3), March 1996. 1.2.2

[26] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1998. 7.4.1, 7.4.4

[27] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016. 7.4.1, 7.4.4

[28] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib-a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020. 10.5

[29] Guy E. Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020. 2.1.1

[30] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. 2.1.1

[31] Otakar Boruvka. O jistém problému minimálním. *Práce Mor. Prırodved. Spol. v Brne (Acta Societ. Scienc. Natur. Moravicae)*, 3(3):37–58, 1926. 1.2.2

[32] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. 1.2.4, 10.1

[33] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, 2005. 7.4.3

[34] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *International Conference on Functional Programming (ICFP)*, 2011. 10.1

[35] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. In *Programming Language Design and Implementation (PLDI)*, 2012. 10.1

[36] Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *International Conference on Functional Programming (ICFP)*, Sep 2014. 1.2.4, 10.1

[37] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. Scan-first search and sparse certificates: an improved parallel algorithm for $k$-vertex connectivity. *SIAM J. Comput.*, 22(1):157–174, 1993. 3, 9.4.4, 19, 9.4.5

[38] Richard Cole. Parallel merge sort. *SIAM J. on Computing*, 17(4):770–785, 1988. 2.2

[39] Richard Cole and Uzi Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986. 3.3.2, 5.1

[40] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. 5.1, 18, 5.2, 5.6

[41] Richard Cole and Uzi Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and computation*, 92(1):1–47, 1991. 5.1, 6.1

[42] Richard Cole, Philip N Klein, and Robert E Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1996. 1.2.2, 6.1, 7.3.1, 9.2

[43] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010. 1.1

[44] Michael S Crouch, Andrew McGregor, and Daniel Stubbs. Dynamic graphs in the sliding-window model. In *European Symposium on Algorithms (ESA)*, 2013. 7.1, 7.4.1, 43, 7.4.2, 7.4.3, 7.4.4, 7.4.4, 7.4.6

[45] Sajal K Das and Paolo Ferragina. An $o(n)$ work EREW parallel algorithm for updating MST. In *European Symposium on Algorithms (ESA)*, 1994. 1.2.2, 7.1

[46] Sajal K Das and Paolo Ferragina. *Parallel Dynamic Algorithms for Minimum Spanning Trees.* Citeseer, 1995. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.136&rep=rep1&type=pdf`. 7.1

[47] Sajal K Das and Paolo Ferragina. An EREW PRAM algorithm for updating minimum spanning trees. *Parallel Process. Lett.*, 9(01):111–122, 1999. 7.1

[48] Mayur. Datar, Aristides. Gionis, Piotr. Indyk, and Rajeev. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. on Computing*, 31(6):1794–1813, 2002. 7.1, 7.4

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `https://doi.org/10.1145/1327452.1327492`. 1.1

[50] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1981. 1.2.4, 10.1

[51] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020. 1.4

[52] Laxman Dhulipala, Quanquan C Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021. 1.4, 5.1

[53] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *ACM Symposium on Theory of Computing (STOC)*, 1987. 10.1.1

[54] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1): 269–271, 1959. 1.2.2

[55] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification–a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. 1.2.2, 6.1, 7.1, 7.4.4

[56] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan conference, Ottawa, Canada*, 2006. 10.5

[57] Martín Farach-Colton and Meng-Tsung Tsai. Exact sublinear binomial sampling. *Algorithmica*, 73(4): 637–651, 2015. 2.3, 9.4.2, 50, 51

[58] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999. 1.2.4, 10.1

[59] Paolo Ferragina. An EREW PRAM fully-dynamic algorithm for MST. In *International Parallel Processing Symposium (IPPS)*, 1995. 7.1

[60] Paolo Ferragina. A technique to speed up parallel fully dynamic algorithms for MST. *J. Parallel Distrib. Comput.*, 31(2):181–189, 1995. 7.1

[61] Paolo Ferragina and Fabrizio Luccio. Batch dynamic algorithms for two graph problems. In *International Conference on Parallel Architectures and Languages Europe*, 1994. 1.1, 1.2.2, 7.1

[62] Paolo Ferragina and Fabrizio Luccio. Three techniques for parallel maintenance of a minimum spanning tree under batch of updates. *Parallel Process. Lett.*, 6(02):213–222, 1996. 1.1, 7.1

[63] Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. on Computing*, 14(4):781–798, 1985. 1.2.1, 1.2.1, 3.1.1, 3.1.1, 3.3, 3.2.1, 6.1, 7.1

[64] Greg N Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and $k$ smallest spanning trees. *SIAM J. on Computing*, 26(2):484–538, 1997. 1.2.1, 1.2.1, 3.1.1, 3.1.1

[65] Greg N Frederickson. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24(1): 37–65, 1997. 1.2.1, 1.2.1, 3.1.1, 3.1.1

[66] Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. *SIAM J. Comput.*, 48(4):1196–1223, 2019. 40, 7.4.6

[67] Harold N Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995. 1.2.3, 9.1

[68] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $O(m \log^2 n)$ time. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2020. 1.2.3, 7.2, 9.1, 9.2, 9.3.2, 9.4.1, 9.6.2

[69] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. on Computing*, 20(6):1046–1067, 1991. 6.1, 6.4.2, 7.4.7

[70] Hillel Gazit, Gary L Miller, and Shang-Hua Teng. Optimal tree contraction in the EREW model. In *Concurrent Computations*, pages 139–156. Springer, 1988. 3.2.1, 5.5.1

[71] Barbara Geissmann and Lukas Gianinazzi. Parallel minimum cuts in near-linear work and low depth. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 1.2.3, 7.4.4, 8.1, 9.1

[72] Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *International Symposium on Distributed Computing (DISC)*, 2013. 9.2

[73] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020. 7.4.4

[74] Mohsen Ghaffari, Christoph Grunau, and Jiahao Qu. Nearly work-efficient parallel DFS in undirected graphs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2023. 1.4, 3.6, 9.7, 11

[75] David Gibb, Bruce Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *arXiv preprint arXiv:1509.06464 [cs.DS]*, 2015. 1.2.2, 6.1

[76] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991. 2.2

[77] Ashish Goel, Michael Kapralov, and Ian Post. Single pass sparsification in the streaming model with edge deletions. *arXiv preprint*, arXiv:1203.4900 [cs.DS], 2012. 7.4.6, 44

[78] Andrew Goldberg, Serge Plotkin, and Gregory Shannon. Parallel symmetry-breaking in sparse graphs. In *ACM Symposium on Theory of Computing (STOC)*, 1987. 5.1

[79] Andrew V Goldberg and Serge A Plotkin. Parallel $(\delta + 1)$-coloring of constant-degree graphs. *Inf. Process. Lett.*, 25(4):241–245, 1987. 5.1, 18, 5.2

[80] Andrew V Goldberg, Michael D Grigoriadis, and Robert E Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Mathematical Programming*, 50(1-3):277–290, 1991. 3.1.1

[81] Tal Goldberg and Uri Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Israel Symposium on the Theory of Computing and Systems*, 1995. 2.2, 5.2, 5.3, 5.5.1

[82] Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial*

*and Applied Mathematics*, 9(4):551–570, 1961. 1.2.3

[83] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in polylogarithmic amortized update time. *ACM Trans. Algorithms*, 14(2):17:1–17:21, 2018. 7.4.4

[84] Yan Gu, Julian Shun, Yihan Sun, and Guy E Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015. 2.2, 3.3.2, 7.3.1

[85] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010. 1.1

[86] Shay Halperin and Uri Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM (extended abstract). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1994. 6.1

[87] Shay Halperin and Uri Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. In *J. Algorithms*, 2000. 6.1

[88] Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A proposal for parallel self-adjusting computation. In *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP)*. ACM, 2007. 1.2.4, 10.1

[89] Matthew Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. 10.1

[90] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*, 2009. 1.2.4, 10.1

[91] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Programming Language Design and Implementation (PLDI)*, 2014. 10.1

[92] JX Hao and James B Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994. 1.2.3

[93] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–355, 1984. 9.6.2

[94] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. 3.1.1

[95] Monika R Henzinger and Valerie King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. on Computing*, 31(2):364–374, 2001. 1.2.2

[96] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 1995. 1.2.1, 1.2.1, 1.2.2, 3.5.3

[97] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, August 1979. 6.1

[98] Jacob Holm and Kristian de Lichtenberg. Top-trees and dynamic graph algorithms. Master's thesis, University of Copenhagen, 1998. 1.2.1, 1.2.1, 3.1.1, 3.1.1

[99] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. 7.1

[100] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. 1.2.2, 6.1, 6.2

[101] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest.

In *European Symposium on Algorithms (ESA)*, 2015. 7.1

[102] Jacob Holm, Eva Rotenberg, and Alice Ryhl. Splay top trees. In *SIAM Symposium on Simplicity in Algorithms (SOSA)*, 2023. 3.1.1

[103] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 510–520, 2017. 1.2.2, 6.1, 6.7

[104] Lorenz Hübschle-Schneider and Peter Sanders. Parallel weighted random sampling. *ACM Transactions on Mathematical Software (TOMS)*, 48(3):1–40, 2022. 9.4.2

[105] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007. 1.1

[106] Anand Iyer, Li Erran Li, and Ion Stoica. CellIQ : Real-time cellular network analytics at scale. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. 6.1

[107] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *International Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2016. 6.1

[108] Vojtěch Jarník. O jistém problému minimálním. *Práca Moravské Pírodovedecké Spolecnosti*, 6:57–63, 1930. 1.2.2

[109] Donald B Johnson and Panagiotis Metaxas. Optimal algorithms for the vertex updating problem of a minimum spanning tree. In *International Parallel Processing Symposium (IPPS)*, 1992. 1.1, 3.3.2, 7.1

[110] Hermann Jung and Kurt Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Inf. Process. Lett.*, 27(5):227–236, 1988. 5.1, 7.1

[111] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013. 1.2.2, 6.1, 6.7

[112] David Karger. *Random sampling in graph optimization problems*. PhD thesis, Stanford University, 1995. 2.3, 9.5, 56, 57, 9.5

[113] David R Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993. 1.2.3, 8.5, 9.1, 1, 9.3, 9.3.2

[114] David R Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000. 1.2.1, 1.2.3, 8.1, 9.1

[115] David R Karger and Rajeev Motwani. Derandomization through approximation: An NC algorithm for minimum cuts. In *ACM Symposium on Theory of Computing (STOC)*, 1994. 9.2, 9.4.5

[116] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996. 1.2.3, 9.1, 9.4.1

[117] David R Karger, Philip N Klein, and Robert E Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995. 1.2.2, 3.5.5, 7.1, 9.3.2

[118] David R. Karger, Noam Nisan, and Michal Parnas. Fast connected components algorithms for the EREW PRAM. *SIAM J. on Computing*, 28(3):1021–1034, February 1999. 6.1

[119] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010. 1.1

[120] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster worst case deterministic dynamic connectivity. In *European Symposium on Algorithms (ESA)*, 2016. 1.2.2, 6.1

[121] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997. 3.5.5, 7.2

[122] Valerie King, Chung Keung Poon, Vijaya Ramachandran, and Santanu Sinha. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Inf. Process. Letters*, 62(3):153–159, 1997. 3.5.5, 3.5.5

[123] János Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985. 7.2

[124] Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. Improved worst-case deterministic parallel dynamic minimum spanning forest. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 7.1

[125] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956. 1.2.2

[126] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022. 1.4, 5.1

[127] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *ACM Symposium on Theory of Computing (STOC)*, 1985. 5.1

[128] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*, 2010. 1.1

[129] David W Matula. A linear time $2+\varepsilon$ approximation algorithm for edge connectivity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993. 9.1, 9.2, 9.4.5

[130] Robert McColl, Oded Green, and David A Bader. A new parallel algorithm for connected components in dynamic graphs. In *IEEE International Conference on High Performance Computing (HiPC)*, 2013. 6.1

[131] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011. 1.1

[132] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, October 1985. 1.2.1, 3.2.1, 4.4

[133] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5. 4.1, 4.5.1

[134] Gary L. Miller and John H. Reif. Parallel tree contraction part 2: Further applications. *SIAM J. on Computing*, 20(6):1128–1147, 1991. 4.1

[135] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science (TCS)*, 130(1), 1994. 1.2.2

[136] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *ACM Symposium on Operating Systems Principles*, 2013. 1.1

[137] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discrete Math.*, 5(1):54–66, 1992. 1.2.3

[138] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse $k$-connected spanning subgraph of a $k$-connected graph. *Algorithmica*, 7(1-6):583–596, 1992. 1.2.3

[139] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $O(n^{1/2-\varepsilon})$-time. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 2017. 1.2.2

[140] C St JA Nash-Williams. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, 1(1):445–450, 1961. 1.2.3

[141] Shaunak Pawagi. A parallel algorithm for multiple updates of minimum spanning trees. In *International Conference on Parallel Processing (ICPP)*, 1989. 1.1, 7.1

[142] Shaunak Pawagi and Owen Kaser. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica*, 9(4):357–381, 1993. 1.1, 7.1

[143] Shaunak Pawagi and IV Ramakrishnan. An $O(\log n)$ algorithm for parallel update of minimum spanning trees. *Information Processing Letters*, 22(5):223–229, 1986. 7.1

[144] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a

minimum spanning forest. *SIAM J. on Computing*, 31(6):1879–1895, 2002. 5.1, 6.1

[145] C. A. Phillips. Parallel graph contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1989. 6.1

[146] Serge A Plotkin, David B Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995. 9.1, 3

[147] Chung Keung Poon and Vijaya Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *International Symposium on Algorithms and Computation (ISAAC)*, 1997. 6.1

[148] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957. 1.2.2

[149] Mihai Pătrașcu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. on Computing*, 35(4):932–963, 2006. 6.1

[150] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1989. 1.2.4, 10.1

[151] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990. 6.3.2

[152] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. on Computing*, 18(3), 1989. 9.3.2

[153] John H. Reif and Sandeep Sen. Parallel computational geometry: An approach using randomization. In *Handbook of Computational Geometry*, chapter 18, pages 765–828. Elsevier Science, 1999. 2.2

[154] John H Reif and Stephen R Tate. Dynamic parallel tree contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1994. 4.1, 4.6

[155] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, September 2008. ISSN 0004-5411. doi: 10.1145/1391289.1391291. URL `http://doi.acm.org/10.1145/1391289.1391291`. 1.2.2

[156] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment (PVLDB)*, 11 (4):420–431, 2017. 6.1

[157] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988. 3.5.3, 3, 3.5.3, 8.4, 9.6.1, 9.6.2

[158] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on GPUs. *Proceedings of the VLDB Endowment (PVLDB)*, 11(1):107–120, September 2017. 6.1

[159] Xiaojun Shen and Weifa Liang. A parallel algorithm for multiple edge updates of minimum spanning trees. In *International Parallel Processing Symposium (IPPS)*, 1993. 1.1, 7.1

[160] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982. 1.2.2, 6.1

[161] Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014. 6.1

[162] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing (Euro-Par)*, 2016. 6.1, 7.1, 7.4.7

[163] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3): 362–391, 1983. 1.2.1, 1.2.1, 3.1.1, 3.1.1, 3.2, 3.4.3, 3.5.3, 8.1, 8.2

[164] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3): 652–686, 1985. 1.2.1, 1.2.1, 3.1.1, 3.1.1, 3.5.3

[165] Philip M. Spira and A Pan. On finding and updating spanning trees and shortest paths. *SIAM J. on Computing*, 4(3):375–380, 1975. 7.1

[166] Xiaoming Sun and David P. Woodruff. Tight bounds for graph problems in insertion streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, 2015. 7.4

[167] Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005. 1.2.1, 1.2.1, 3.1.1, 3.1.1

[168] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. 1.2.2

[169] Robert Endre Tarjan. Complexity of monotone networks for computing conjunctions. In *Annals of Discrete Mathematics*, volume 2, pages 121–133. Elsevier, 1978. 3.5.4

[170] Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979. 3.5.5

[171] Robert Endre Tarjan. *Data structures and network algorithms*. SIAM, 1983. 1.2.1, 1.2.2, 3.5.5, 7.1, 7.3, 42

[172] Mikkel Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33(2):229–243, 1999. 1.2.2

[173] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 2000. 1.2.2, 6.1

[174] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-parallel euler tour trees. In *Algorithm Engineering and Experiments (ALENEX)*, 2019. 1.2.1, 1.2.1, 1.2.1, 3.1.1, 4.1, 4.6, 6.1, 6.3, 6.3.2, 6.3.2, 6.4.1, 6.4.1, 6.4.2, 6.4.4, 6.4.4, 6.5.2

[175] Tom Tseng, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic minimum spanning forest and the efficiency of dynamic agglomerative graph clustering. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022. 1.4, 5.1, 6.7, 11

[176] Yung Hyang Tsin. On handling vertex deletion in updating spanning trees. *Information Processing Letters*, 27(4):167–168, 1988. 7.1

[177] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, 1990. 4.2

[178] Peter Varman and Kshitij Doshi. A parallel vertex insertion algorithm for minimum spanning trees. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 1986. 7.1

[179] Peter Varman and Kshitij Doshi. An efficient parallel algorithm for updating minimum spanning trees. *Theoretical Computer Science (TCS)*, 58(1-3):379–397, 1988. 7.1

[180] Uzi Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984. 6.1

[181] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017. 6.1

[182] Zhengyu Wang. An improved randomized data structure for dynamic graph connectivity. *arXiv preprint arXiv:1510.04590 [cs.DS]*, 2015. 1.2.2, 6.1

[183] Renato F Werneck. *Design and analysis of data structures for dynamic trees*. PhD thesis, Princeton University, 2006. 5.4.1

[184] C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Chelmis, and V. K. Prasanna. Real-time analytics for fast evolving social graphs. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015. 6.1

[185] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013. 1.2.2, 6.1

[186] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 2017. 1.2.2