Work-Efficient Batch-Incremental Minimum Spanning Trees with Applications to the Sliding-Window Model

Daniel Anderson
Carnegie Mellon University
dlanders@cs.cmu.edu

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

Kanat Tangwongsan*
Mahidol University
International College
kanat.tan@mahidol.edu

ABSTRACT

Algorithms for dynamically maintaining minimum spanning trees (MSTs) have received much attention in both the parallel and sequential settings. While previous work has given optimal algorithms for dense graphs, all existing parallel batch-dynamic algorithms perform polynomial work per update in the worst case for sparse graphs. In this paper, we present the first work-efficient parallel batch-dynamic algorithm for incremental MST, which can insert ℓ edges in $O(\ell \lg(1 + n/\ell))$ work in expectation and $O(\text{polylog}(n))$ span w.h.p. The key ingredient of our algorithm is an algorithm for constructing a *compressed path tree* of an edge-weighted tree, which is a smaller tree that contains all pairwise heaviest edges between a given set of marked vertices. Using our batch-incremental MST algorithm, we demonstrate a range of applications that become efficiently solvable in parallel in the sliding-window model, such as graph connectivity, approximate MSTs, testing bipartiteness, k-certificates, cycle-freeness, and maintaining sparsifiers.

ACM Reference Format:
Daniel Anderson, Guy E. Blelloch, and Kanat Tangwongsan. 2020. Work-Efficient Batch-Incremental Minimum Spanning Trees with Applications to the Sliding-Window Model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3350755.3400241

1 INTRODUCTION

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 [10], and Jarník [35] (later rediscovered by Prim [45] and Dijkstra [18]), and later, the perhaps more well-known algorithm of Kruskal [40]. 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. While most dynamic data structures handle only a single update at a time, there has also been work on *batch-dynamic* data structures, which process a batch of updates. Typically, it is assumed that the size of a batch can vary from round to round. Batch-dynamic data structures have two potential advantages—they can allow for more parallelism, and they can, in some situations, perform less work than processing updates one at a time.

There has been significant interest in parallelizing incremental and dynamic MSF. Some of this work studies how to implement single updates in parallel [14, 16, 20, 21, 37, 39, 44, 52–54], and some studies batch updates [22, 23, 36, 42, 43, 46]. The most recent and best result [39] requires $O(\sqrt{n} \lg n)$ work per update on n vertices, and only allows single edge updates. All previous results that support batches of edge updates in polylogarithmic time require $\Omega(n \min(\ell, n))$ work, where ℓ is the size of the batch. This is very far from the work performed by the best sequential data structures, which is $O(\lg n)$ worst-case time for incremental edge insertions [5, 48], and $O\left(\frac{\lg^4 n}{\lg \lg n}\right)$ amortized expected time for fully dynamic edge insertions and deletions [33].

In this paper, 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 [38] when inserting all edges as a batch. The size of a batch can vary from round to round. Our main contribution is summarized by the following theorem:

Theorem 1.1. *There exists a data structure that maintains the MSF of a weighted undirected graph that can insert a batch of ℓ edges into a graph with n vertices in $O\left(\ell \lg\left(1 + \frac{n}{\ell}\right)\right)$ work in expectation and $O(\lg^2(n))$ span w.h.p.[1] in the arbitrary-CRCW PRAM.*

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 [17], 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 newer

*Corresponding Author

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400241

[1] We say that $g(n) \in O(f(n))$ *with high probability* (w.h.p.) if $g(n) \in O(kf(n))$ with probability at least $1 - O(1/n^k)$ for all $k \geq 1$

ones. There have been many dozens, perhaps hundreds, of papers using the model in general. Crouch et. al. [13] 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. [13], 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. Our results are summarized by the following theorem:

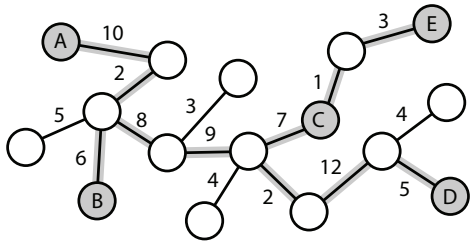
Theorem 1.2. *There exist data structures for the batch sliding-window model (batch edge insertions on the new side and deletions on the old size) for the problems of maintaining connectivity, k-certificate, bipartiteness, $(1+\epsilon)$-MSF, cycle-freeness, and ε-sparsifiers, that all require $\tilde{O}(n)$ memory, support batch updates of size ℓ in $\tilde{O}(\ell)$ work and polylogarithmic span, and queries (except for sparsifiers) in polylogarithmic work, where n denotes the number of vertices.*

Finally, we note that we can also apply these techniques to the incremental setting, and, using existing results on batch-incremental graph connectivity [47], obtain fast algorithms there as well. Table 1 gives more specifics on the individual results and compares them to the existing bounds for parallel dynamic graphs in the incremental and fully dynamic settings.

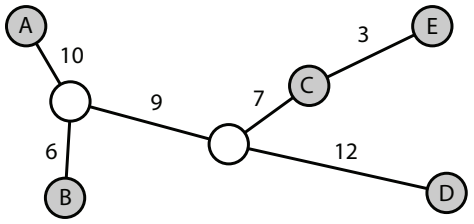
1.1 Technical Overview

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. 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 some recent results on parallel batch-dynamic tree contraction and parallel rake-compress trees (RC trees) [2].

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 $O(\lg(n))$ time solution to the sequential incremental MSF problem [48]. To handle a batch of edge insertions, our algorithm computes the compressed path tree



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

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. [12], which is linear work in expectation and logarithmic span w.h.p., which in turn is based on the linear time sequential algorithm [38]. Since the compressed path tree has size $O(\ell)$, 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.

Lastly, using a mix of known reductions and several new ideas, we show how our batch-incremental MSF algorithm can be used to solve problems in a parallel version of the sliding-window graph streaming model. A useful ingredient in this step is the *recent edge* property [13], which says that by weighting the edges of a graph stream with successively lower weights over time, connectivity between a pair of vertices in the window can be tested by inspecting the heaviest (i.e. oldest) edge on the path from u to v in an MSF of the graph so far. Combining this idea with the use of several work-efficient parallel batch-dynamic data structures, we show how to maintain graph connectivity, bipartiteness, approximate MSFs, k-certificates, cycle-freeness, and sparsifiers, subject to batch updates in $O(\text{polylog}(n))$ work and span per edge update, and $O(n\,\text{polylog}(n))$ space.

1.2 Related Work

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 [49] 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

	Incremental (This paper)	Sliding window (This paper)	Fully dynamic (Previous work)
Connectivity	$O(\ell\alpha(n))^*$ (Previous work [47])	$O(\ell \lg(1+n/\ell))^*$	$O(\ell \lg(n) \lg(1+n/\ell))^{*,\dagger}$ [1]
k-certificate	$O(k\ell\alpha(n))^*$	$O(k\ell \lg(1+n/\ell))^*$	-
Bipartiteness	$O(\ell\alpha(n))^*$	$O(\ell \lg(1+n/\ell))^*$	-
Cycle-freeness	$O(\ell\alpha(n))^*$	$O(\ell \lg(1+n/\ell))^*$	-
MSF	$O(\ell \lg(1+n/\ell))^*$	$O(\varepsilon^{-1}\ell \lg(n) \lg(1+n/\ell))^{*,\ddagger}$	$O(\ell n \lg\lg\lg(n) \lg(m/n))$ [22]
ε-sparsifier	$O(\varepsilon^{-2}\ell \lg^4(n)\alpha(n))^*$	$O(\varepsilon^{-2}\ell \lg^4(n) \lg(1+n/\ell))^*$	-

Table 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. ℓ 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 ℓ, these algorithms sometimes achieve better bounds. Some bounds given are randomized ($*$), amortized ($\dagger$), or give $(1+\varepsilon)$-approximate solutions ($\ddagger$)

first sublinear time algorithm for edge updates was given by Frederickson [24], who gave an $O(\sqrt{m})$ algorithm. A well-celebrated improvement to Frederickson's algorithm was given by Eppstein et. al [19], 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 $O(\lg(n))$ time per update using dynamic trees [5, 48] 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 [32], supporting updates in $O(\lg^4(n))$ amortized time per operation, later improved by a $\lg\lg n$ factor [33] in expectation. No worst-case polylogarithmic time algorithm is known for the fully dynamic case. This paper is concerned with algorithms for MSTs that are both parallel and dynamic.

Parallel single-update algorithms. Work by Pawagi and Ramakrishnan [44] gives a parallel algorithm for vertex insertion (with an arbitrary number of adjacent edges) and edge-weight updates in $O(\lg(n))$ span but $O(n^2 \lg(n))$ work. Varman and Doshi [53, 54] improve this to $O(n \lg(n))$ work. Jung and Mehlhorn [37] give an algorithm for vertex insertion in $O(\lg(n))$ 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 [52] extended the work of Pawagi and Ramakrishnan [44] to handle vertex deletions in the same time bounds, thus giving a fully vertex-dynamic parallel algorithm in $O(\lg(n))$ span and $O(n^2 \lg(n))$ work. Das and Ferragina [14] give algorithms for inserting and deleting edges in $O(\lg(\frac{m}{n}) \lg(n))$ span and $O(n^{2/3} \lg(\frac{m}{n}))$ work. Subsequent improvements by Ferragina [20, 21], and Das and Ferragina [16] improve the span bound to $O(\lg(n))$ with the same work bound. A recent result by Kopelowitz et al. [39] gives an algorithm that takes $O(\sqrt{n} \lg(n))$ work and $O(\lg(n))$ 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 [42] gives an algorithm for batch vertex insertion that inserts ℓ vertices in $O(\lg(n) \lg(\ell))$ span and $O(n\ell \lg(n) \lg(\ell))$ work. Johnson and Metaxas [36] give an algorithm for the same problem with $O(\lg(n) \lg(\ell))$ span and $O(n\ell)$ work.

Pawagi and Kaser [43] were the first to give parallel batch-dynamic algorithms for fully-dynamic MSTs. For inserting ℓ independent vertices, inserting ℓ edges, or decreasing the cost of ℓ edges, their algorithms takes $O(\lg(n) \lg(\ell))$ span and $O(n\ell)$ work. Their algorithms for increasing the cost of or deleting ℓ edges, or deleting a set of vertices with total degree ℓ take $O(\lg(n) + \lg^2(\ell))$ span and $O\left(n^2\left(1 + \frac{\lg^2(\ell)}{\lg(n)}\right)\right)$ work. Shen and Liang [46] give an algorithm that can insert ℓ edges, modify ℓ edges, or delete a vertex of degree ℓ in $O(\lg(n) \lg(\ell))$ span and $O(n^2)$ work. Ferragina and Luccio [22, 23] give algorithms for handling $\ell = O(n)$ edge insertions in $O(\lg(n))$ span and $O(n \lg\lg\lg(n) \lg(m/n))$ work, and ℓ edge updates in $O(\lg(n) \lg(m/n))$ span and $O(\ell n \lg\lg\lg(n) \lg(m/n))$ work. Lastly, Das and Ferragina's algorithm [14] can be extended to the batch case to handle ℓ edge insertions in $O(\ell + \lg(m/n) \lg(n))$ span and $O(n^{2/3}(\ell + \lg(m/n)))$ work.

For a thorough and well written survey on the techniques used in many of the above algorithms, see Das and Ferragina [15].

Sliding window dynamic graphs. Dynamic graphs in the sliding window model were studied by Crouch et. al [13]. In the sliding window model, there is an infinite stream of edges $\langle e_1, e_2, \ldots\rangle$, 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.

2 PRELIMINARIES

2.1 Model of Computation

Parallel Random Access Machine (PRAM**).** The parallel random access machine (PRAM) is a parallel machine model with p processors that work in lockstep, connected to a shared memory [34]. In this paper we work with the Concurrent-Read Concurrent-Write model (CRCW PRAM), where memory locations are allowed to be concurrently read and concurrently written to. If multiple writers write to the same location concurrently, we assume that an arbitrary writer wins. We analyze algorithms in terms of their *work* and *span*, where work is the total number of instructions performed by the algorithm and span (also called depth) is the length of the longest chain of sequentially dependent instructions [7].

2.2 Tree Contraction and Rake-compress trees

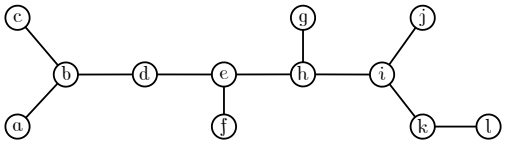
Tree contraction produces, from a given input tree, a set of smaller (contracted) trees, each with a subset of the vertices from the previous one, until the final layer which has a single remaining vertex.

Miller and Reif [41] give an algorithm for tree contraction that produces a set of $O(\lg(n))$ trees w.h.p, with a geometrically decreasing number of vertices in each one. Specifically, the technique of Miller and Reif involves sequential rounds of applying two operations in parallel across every vertex of the tree, rake and compress. The rake operation removes a leaf from the tree and merges it with its neighbor. The compress operation takes a vertex of degree two whose neighbors are not leaves and removes it, merging the two adjacent edges. The algorithm operates on bounded-degree trees, but arbitrary degree trees can easily be handled by converting them into equivalent bounded degree trees, as described in [2].

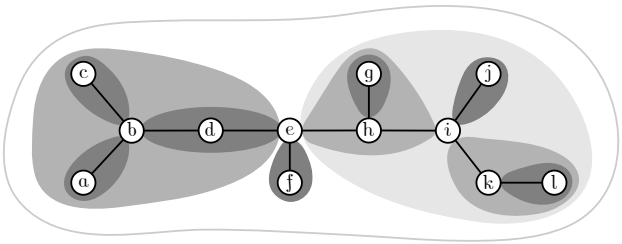
A powerful application of tree contraction is that it can be used to produce a recursive clustering of the given tree with attractive properties. Using Miller and Reif's tree contraction, a recursive clustering can be produced that consists of $O(n)$ clusters, with recursive height $O(\lg(n))$ w.h.p. Such a clustering can be represented as a so-called *rake-compress tree* (RC tree) [3]. Recent work has shown how to maintain a tree contraction dynamically subject to batch-dynamic updates, work efficiently, and with low span [2]. These results also facilitate maintaining RC trees subject to batch-dynamic updates work-efficiently and in low span. Specifically, an RC tree can be built in $O(n)$ work in expectation and $O(\lg^2(n))$ span w.h.p., and subsequently updated in $O(\ell \lg(1 + n/\ell))$ work in expectation and $O(\lg^2(n))$ span w.h.p. for batches of ℓ edges.

Rake-compress trees. RC trees encode a recursive clustering of a tree. A cluster is defined to be a connected subset of vertices and edges of the original tree. Importantly, it is possible for a cluster to contain an edge without containing its endpoints. The *boundary vertices* of a cluster C are the vertices $v \notin C$ such that an edge $e \in C$ has v as one of its endpoints. All of the clusters in an RC tree have at most two boundary vertices. A cluster with no boundary vertices is called a *nullary cluster*, a cluster with one boundary is a *unary cluster* (corresponding to a rake) and a cluster with two boundaries is *binary cluster* (corresponding to a compress). The root cluster is always a nullary cluster. Nodes in an RC tree correspond to clusters, such that a node is always the disjoint union of its children. The leaf clusters of the RC tree are the vertices and edges of the original tree. Note that all non-leaf clusters have exactly one vertex (leaf) cluster as a child. This vertex is that cluster's *representative* vertex. Clusters have the useful property that the constituent clusters of a parent cluster C share a single boundary vertex in common—the representative of C, and their remaining boundary vertices become the boundary vertices of C. See Figure 2 for an example of a tree, a recursive clustering, and its corresponding RC tree. Note that for a disconnected forest, the RC tree algorithm will simply produce a separate root cluster for each component.

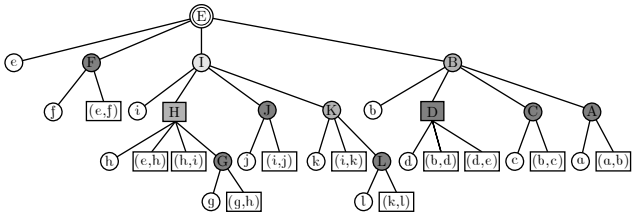
RC trees support a multitude of different kinds of queries [3], all in $O(\lg(n))$ time. In this paper, we will make use of path queries: given a pair of vertices u and v, find the heaviest edge on the path from u to v. We refer the reader to [2] and [3] for a more in-depth explanation of RC trees and their properties.



(a) A tree



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



(c) The corresponding RC tree. (Non-base) unary clusters are shown as circles, binary clusters as rectangles, and the finalize (nullary) cluster at the root with two concentric circles. The base clusters (the leaves) are labeled in lowercase, and the composite clusters are labeled with the uppercase of their representative.

Figure 2: A tree, a recursive clustering of the tree, and the corresponding RC tree [2].

3 THE COMPRESSED PATH TREE

Given an RC tree and a set of ℓ marked vertices, our algorithm produces the compressed path tree in $O(\ell \lg(1 + n/\ell))$ work in expectation and $O(\lg(n))$ span w.h.p.

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 of that cluster immediately. The algorithm uses the following key recursive subroutine.

— ExpandCluster(C : ***Cluster***) : ***Graph***
Return the compressed path tree of the subgraph corresponding to the graph $C \cup$ Boundary(C), assuming that the boundary vertices of C are marked.

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 3.1 illuminates the reason for the precise behavior of Prune. We give pseudocode for ExpandCluster in Algorithm 1. 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.

Algorithm 1 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 ← Boundary(C)
5:      if C is a BinaryCluster then
6:        local e ← (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 ← ⋃_{c∈Children(C)} ExpandCluster(c)
14:     return Prune(G, Representative(C))
```

3.1 Analysis

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

Lemma 3.1. *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.

(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 cluster corresponding to a single vertex, 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, we can conclude that the algorithm returns the compressed path tree of the graph $C \cup$ Boundary(C), assuming that the boundary vertices of C are marked. □

Theorem 3.2. *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 3.1 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 3.3. *A compressed path tree for ℓ marked vertices has at most $O(\ell)$ vertices.*

Proof. Since a compressed path tree has no non-marked leaves, it has at most ℓ leaves. Similarly, by definition, the compressed path tree has at most ℓ internal nodes of degree at most two. The result then follows from the fact that a tree with ℓ leaves and no internal nodes of degree less than two has $O(\ell)$ vertices. □

Lemma 3.4 ([2]). *Given the RC tree of an n-vertex tree, ℓ root-to-leaf paths in the RC tree touch $O(\ell \lg(1 + n/\ell))$ nodes in expectation.*

Theorem 3.5. *Given the RC tree of a tree on n vertices, the compressed path tree for ℓ marked vertices can be produced in $O(\ell \lg(1 + n/\ell))$ expected work and $O(\lg(n))$ span w.h.p. on the CRCW* PRAM.

Proof. The algorithm for producing the compressed path tree consists of two bottom-up traversals of the RC tree from the ℓ 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), Lemma 3.4 implies the work bound of $O(\ell \lg(1 + n/\ell))$ in expectation.

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 $O(\lg(n))$ w.h.p. 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. □

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. 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 $O(\lg(n))$ span w.h.p., this leaves the work of the algorithm unaffected, but increases the span to $O(\lg^2(n))$ w.h.p.

4 PARALLEL BATCH-INCREMENTAL MSF

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 [48] 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 2. 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 1. 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 2 Batch-incremental MSF

```
1: procedure BatchInsert(T : RCTree, E+ : edge list)
2:     local K ← ⋃_{({u,v},w)∈E+} {u, v}
3:     local C ← CompressedPathTrees(RC, K)
4:     local M ← MSF(C ∪ E+)
5:     T.BatchDelete(E(C) \ E(M))
6:     T.BatchInsert(E(M) ∩ E+)
```

4.1 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 4.1 (Red rule [51]). *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 4.2. *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*

$$M' = MST(G) \cup (E(M) \cap E^+) \setminus (E(C) \setminus E(M)),$$

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 $\text{MST}(G) \cup E^+$, 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. □

Corollary 4.3. *Algorithm 2 correctly updates the MSF.*

Proof. Theorem 4.2 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. □

Efficiency. We now show that the batch-incremental MSF algorithm achieves low work and span.

Theorem 4.4. *Batch insertion of ℓ edges using Algorithm 2 takes $O(\ell \lg(1 + n/\ell))$ expected work and $O(\lg^2(n))$ span w.h.p.*

Proof. Collecting the endpoints of the edges (Line 2) takes $O(\ell)$ work in expectation and $O(\lg(\ell))$ span w.h.p. using a semisort [31]. By Theorem 3.5, Line 3 takes $O(\ell \lg(1 + n/\ell))$ work in expectation and $O(\lg(n))$ span w.h.p. By Lemma 3.3, the graph $C \cup E^+$ is of size $O(\ell)$, and hence by using the MSF algorithm of Cole et. al. [12], which runs in linear work in expectation and logarithmic span w.h.p, Line 4 takes $O(\ell)$ work in expectation and $O(\lg(\ell))$ span w.h.p. Then, since $C \cup E^+$ is of size $O(\ell)$, the batch updates to the RC tree (Lines 5 and 6) take $O(\ell \lg(1+n/\ell))$ work in expectation and $O(\lg^2(n))$ span w.h.p. Lastly, since $O(\lg(\ell)) = O(\lg(n))$, summing these up, we can conclude that Algorithm 2 takes $O(\ell \lg(1 + n/\ell))$ work in expectation and $O(\lg^2(n))$ span w.h.p. □

5 APPLICATIONS TO SLIDING WINDOW

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(Δ: ***int***) Delete the oldest Δ 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 isConnected query operation.

This formulation is a natural extension of the sequential sliding-window model. Traditionally, the sliding-window model [17] entails maintaining the most recent W items, where W is a fixed, prespecified size. Hence, an explicit expiration operation is not necessary. More recently, there has been interest in maintaining variable-sized sliding windows (e.g., events in the past 11 minutes). 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 [50] 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 $O(\lg n)$ 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.

5.1 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 5.1 (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 $\ell = |B|$ new edges in $O(1 + \ell \lg(n/\ell))$ expected work and $O(\lg^2 \ell)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(1)$ worst-case work and span.*
- isConnected(u, v) *returns whether u and v are connected in $O(\lg n)$ work and span w.h.p.*

Following Crouch et al. [13], 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^{th} edge has index i.) Then, implicit in their paper is the following lemma:

Lemma 5.2 (Recent Edge [13]). *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 5.1. We maintain (i) an incremental MSF data structure from Theorem 1.1 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 $\ell = |B|$ edges, where an edge $e \in B$ is assigned a weight of $-\tau(e)$. The operation BatchExpire(Δ) is handled by advancing T_W by Δ. The cost of these operations is clearly as claimed.

The query isConnected(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 5.2). The claimed cost bound follows because the MSF is maintained as an RC tree, which supports path queries in $O(\lg n)$ [2]. □

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 $\mathcal{D}$, 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 2, Lines 5-6). We can then adjust $\mathcal{D}$ using cost at most $O(n \lg(n/t))$ work and $O(\lg^2 n)$ span (e.g., [8, 9]). The BatchExpire operation applies Split to find expired edges (costing $O(\lg n)$ work and span) and explicitly deletes these edges from the MSF (costing expected $O(\Delta \lg(n/\Delta))$ work and $O(\lg^2 n)$ span w.h.p.). With these changes, numComponents() is answered by returning $n - |\mathcal{D}|$ and SW-Conn-Eager has the following cost bounds:

Theorem 5.3 (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 $\ell = |B|$ new edges in $O(1 + \ell \lg(n/\ell))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(\Delta \lg(1+n/\Delta)+\lg n)$ expected work and $O(\lg^2 n)$ span w.h.p.*
- isConnected(u, v) *returns whether u and v are connected in $O(\lg n)$ work and span w.h.p.*
- numComponents() *returns the number of connected components in $O(1)$ worst-case work and span.*

5.2 Bipartiteness

To monitor bipartiteness, we apply a known reduction [4, 13]—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 5.4 (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 $\ell = |B|$ new edges in $O(\ell \lg(1 + n/\ell))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(\Delta \lg(1+n/\Delta)+\lg n)$ expected work and $O(\lg^2 n)$ span w.h.p.*
- isBipartite() *returns a Boolean indicating whether the graph is bipartite in $O(1)$ worst-case work and span.*

5.3 Approximate MSF Weight

For this problem, assume that the edge weights are between 1 and $n^{O(1)}$. Using known reductions [4, 11, 13], 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

$$(n - \mathrm{cc}(G_0)) + \sum_{i \geq 1} (\mathrm{cc}(G_{i-1}) - \mathrm{cc}(G_i))(1 + \varepsilon)^i, \tag{1}$$

where $\mathrm{cc}(G)$ is the number of connected components in graph G.

Let $R = O(\varepsilon^{-1} \lg n)$. 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 ℓ 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 (1), which involves R terms and calls to numComponents(). This recomputation step requires $O(R)$ work and $O(\lg R) = O(\lg^2 n)$ span. Overall, we have the following:

Theorem 5.5 (Approximate MSF). *Fix $\varepsilon > 0$. For an n-vertex graph, there is a data structure for approximate MSF weight that requires $O(\varepsilon^{-1} n \lg n)$ space and supports the following:*

- BatchInsert(B) *handles $\ell = |B|$ new edges in $O(\varepsilon^{-1} \ell \lg n \lg(1 + n/\ell))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(\varepsilon^{-1} \Delta \lg n \lg(1+n/\Delta))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- weight() *returns an $(1 + \varepsilon)$-approximation to the weight of the MSF in $O(1)$ worst-case work and span.*

5.4 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. [13] show how to maintain such decomposition on a sliding window. When extended to the batch setting, the steps are as follows: Let O_0 be the new batch of edges B. Then:

> For $i = 1, \ldots, k$, 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^+, and set $O_i = F_i^- \cup (O_{i-1} \setminus F_i^+)$.

Via known reductions [4, 13], 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. We have the following:

Theorem 5.6 (k-Certificate). *For an n-vertex graph, there is a data structure for k-certificate that requires $O(kn)$ space and supports the following:*

- BatchInsert(B) *handles $\ell = |B|$ new edges in $O(k\ell \lg(1 + n/\ell))$ expected work and $O(k \lg^2 n)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(k\Delta \lg(1 + n/\Delta))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- makeCert() *returns a k-certificate involving at most $k(n-1)$ edges in $O(kn)$ work and $O(\lg n)$ span.*

Proof. We maintain each F_i using a batch incremental MSF data structure from Theorem 1.1. To allow eager eviction of expired edges, we additionally keep for each F_i a parallel ordered-set data structure (e.g., [8, 9]) $\mathcal{D}_i$, 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 $\mathcal{D}_i$ can be updated accordingly. Note that the size of the changes to $\mathcal{D}_i$ never exceeds $O(\ell)$. The operation BatchExpire involves expiring edges in all $\mathcal{D}_i$'s. Finally, the operation makeCert is supported by copying and returning $\cup_{i=1}^{k} \mathcal{D}_i$. 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 $O(n \lg n)$ time per update is known [19]. By contrast, for the incremental setting, there is a recent algorithm with $\widetilde{O}(1)$ time per update [30]. In the sliding window model, as a corollary of Theorem 5.6, the k-certificate can be used to test k-connectivity via a parallel global min-cut algorithm (e.g., [27, 28]). Because there are $O(kn)$ edges, this takes $O(kn \lg n + n \lg^4 n)$ work and $O(\lg^3 n)$ span [28].

5.5 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 5.6 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 5.7 (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) *handles $\ell = |B|$ new edges in $O(\ell \lg(1 + n/\ell))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(\Delta \lg(1 + n/\Delta))$ expected work and $O(\lg^2 n)$ span w.h.p.*
- hasCycle() *returns true or false indicating whether the graph has a cycle in $O(1)$ work and span.*

5.6 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 ε-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 5.8 (Fung et al. [25]). *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 $p_e \geq \min\left(1, \frac{253}{c_e \varepsilon^2} \lg^2 n\right)$ and assigned a weight of $1/p_e$, then with high probability, the resulting graph is an ε-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 paper enable maintaining an ε-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 [29] and sampling sufficiently many edges at geometric probability scales (e.g., [4, 13]).

The key result for connectivity estimation is as follows: For $i = 1, 2, \ldots, L = O(\lg n)$ and $j = 1, 2, \ldots, K = O(\lg n)$, 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 $0 \leq j \leq K$, gives an estimate of uv connectivity:

Lemma 5.9 ([29]). *With high probability, for every edge e of G, $\Theta(s_e / \lg n) \leq 2^{L(e)} \leq 2c_e$, where s_e denotes strong connectivity and c_e denotes edge connectivity.*

The same argument also gives $c_e \leq \Theta(2^{L(e)} \lg n)$ 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 5.1), requiring a total of $O(K \cdot L \cdot n) = O(n \lg^2 n)$ 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. [4]: 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 Q_i, where Q_i is a k-SW-Certificate data structure (Theorem 5.6) with $k = O(\frac{1}{\varepsilon^2}\lg^3 n)$ is sufficient[2]. Maintaining these requires a total of $O(knL) = O(\varepsilon^{-2} n \lg^4 n)$ space.

Ultimately, our algorithm simulates sampling an edge e with probability $2^{-\lfloor \lg_2 \widetilde{p}_e \rfloor}$, where

$$\widetilde{p}_e = \min\left(1, O(2^{-L(e)}\varepsilon^{-2}\lg^2 n)\right),$$

which uses an estimate of $2^{L(e)}$ in place of c_e. It answers a SPARSIFY query as follows:

> For $e \in \bigcup_{i=1}^{L} Q_i$, output e in the sparsifier with weight $1/\widetilde{p}_e$ if e appears in $Q_{\beta(e)}$, where $\beta(e) = \lfloor \lg_2 \widetilde{p}_e \rfloor$.

We now show that the Q_i's retain sufficient edges.

Lemma 5.10. *With high probability, an edge e that is sampled into $H_{\beta(e)}$ is retained in $Q_{\beta(e)}$.*

Proof. Consider an edge $e = \{u, v\}$. There are c_e disjoint paths between u and v. With high probability, because $c_e \le \Theta(2^{L(e)} \lg n)$, the expected number of paths that stay connected in $H_{\beta(e)}$ is at most $2\widetilde{p}_e \cdot c_e \le O(\varepsilon^{-2}\lg^3 n)$. By Chernoff bounds, it follows that w.h.p., e has edge connectivity in $H_{\beta(e)}$ at most $k = O(\varepsilon^{-2}\lg^3 n)$ for sufficiently large constant. Hence, e is retained in $Q_{\beta(e)}$ w.h.p. □

This means that at query time, with high probability, every edge e is sampled into the sparsifier with probability $2^{-\lfloor \lg_2 \widetilde{p}_e \rfloor} \ge p_e$, so the resulting graph is an ε-sparsifier w.h.p. (Theorem 5.8). Moreover, the number of edges in the sparsifier is, in expectation, at most

$$\sum_{e \in E(G)} 2\widetilde{p}_e = O(\varepsilon^{-2}\lg^3 n) \sum_{e \in E(G)} \frac{1}{s_e} = O(\varepsilon^{-2} n \lg^3 n),$$

where we used Lemma 5.9 and the fact that $\sum_e 1/s_e \le n - 1$ [6, 25].

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 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 Q_i's, each of which takes $O(k\ell \lg(1 + n/\ell))$ expected work and $O(k\lg^2 n)$ 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 Q_i's. Finally, the query operation SPARSIFY involves considering the edges of $\bigcup_{i=1}^{L} Q_i$ in parallel, each requiring a call to $L(e)$, which can be answered in $O(LK \lg n) = O(\lg^3 n)$ work and span. In total, this costs $O(nkL\lg^3 n) = O(n\,\mathrm{polylog}(n))$ work and $O(\mathrm{polylog}(n))$ span. The following theorem summarizes our result for graph sparsification:

[2]We remark that the Q_i instances themselves contain enough information to estimate c_e for all edges, but we do not know how to do so efficiently.

Theorem 5.11 (Graph Sparsification). *For an n-vertex graph, there is a data structure for graph (cut) sparsification that requires $O(\varepsilon^{-2} n \lg^4 n)$ space and supports the following:*

- BatchInsert(B) *handles $\ell = |B|$ new edges in $O(\frac{1}{\varepsilon^2}\ell \lg(1+\frac{n}{\ell})\lg^4 n)$ expected work and $O(\varepsilon^{-2}\lg^5 n)$ span w.h.p.*
- BatchExpire(Δ) *expires Δ oldest edges in $O(\frac{1}{\varepsilon^2}\Delta \lg(1+\frac{n}{\Delta})\lg^4 n)$ expected work and $O(\lg^2 n)$ span w.h.p.*
- SPARSIFY() *returns an ε-sparsifier with high probability. The sparsifier has $O(\varepsilon^{-2} n \lg^3 n)$ edges and is produced in $O(n\,\mathrm{polylog}(n))$ work and $O(\mathrm{polylog}(n))$ span w.h.p.*

5.7 Connection to Batch Incremental

All applications studied here were built on top of the connectivity data structures (Theorem 5.1). In the related batch incremental setting, an analog of Theorem 5.1 was given by Simsiri et al. [47], where BatchInsert takes $O(\ell\alpha(n))$ expected work and $O(\mathrm{polylog}(n))$ span, and isConnected takes $O(\alpha(n))$ work and span.

With this result, we can derive an analog of Theorem 5.3 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 [26]. 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 5.3, where BatchInsert still takes $O(\ell\alpha(n))$ expected work and $O(\mathrm{polylog}(n))$ span, isConnected takes $O(\alpha(n))$ work and span, and numComponents takes $O(1)$ work and span. Ultimately, this means that replacing Theorems 5.1 and 5.3 with their analogs in each application effectively replaces the $\lg(1 + n/\ell)$ factor in the work term with an $\alpha(n)$ term, leading to the cost bounds presented in Table 1.

6 CONCLUSION

This paper presented the first work-efficient parallel algorithm for batch-incremental MSFs. 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.

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.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223. The authors would like to thank Ticha Sethapakdi for helping with the figures.

REFERENCES

[1] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. 2019. Parallel Batch-Dynamic Graph Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[2] Umut A Acar, Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-dynamic Trees via Change Propagation. *arXiv preprint* arXiv:2002.05129 [cs.DS] (2020).

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

[4] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Analyzing graph structure via linear measurements. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[5] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2005. Maintaining Information in Fully Dynamic Trees with Top Trees. *ACM Trans. Algorithms* 1, 2 (2005), 243–264.

[6] András A. Benczúr and David R. Karger. 1996. Approximating *s-t* Minimum Cuts in $\tilde{O}(n^2)$ Time. In *ACM Symposium on Theory of Computing (STOC)*.

[7] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[8] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[9] Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[10] Otakar Boruvka. 1926. O jistém problému minimálním. *Práce Mor. Prırodved. Spol. v Brne (Acta Societ. Scienc. Natur. Moravicae)* 3, 3 (1926), 37–58.

[11] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. 2005. Approximating the Minimum Spanning Tree Weight in Sublinear Time. *SIAM J. Comput.* 34, 6 (2005), 1370–1379.

[12] Richard Cole, Philip N. Klein, and Robert E. Tarjan. 1996. Finding Minimum Spanning Forests in Logarithmic Time and Linear Work Using Random Sampling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[13] Michael S Crouch, Andrew McGregor, and Daniel Stubbs. 2013. Dynamic graphs in the sliding-window model. In *European Symposium on Algorithms (ESA)*.

[14] Sajal K Das and Paolo Ferragina. 1994. An o(n) work EREW parallel algorithm for updating MST. In *European Symposium on Algorithms (ESA)*.

[15] Sajal K Das and Paolo Ferragina. 1995. *Parallel Dynamic Algorithms for Minimum Spanning Trees.* Citeseer. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.136&rep=rep1&type=pdf.

[16] Sajal K Das and Paolo Ferragina. 1999. An EREW PRAM algorithm for updating minimum spanning trees. *Parallel Process. Lett.* 9, 01 (1999), 111–122.

[17] Mayur. Datar, Aristides. Gionis, Piotr. Indyk, and Rajeev. Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *SIAM J. on Computing* 31, 6 (2002), 1794–1813.

[18] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[19] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. 1997. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM* 44, 5 (1997), 669–696.

[20] Paolo Ferragina. 1995. An EREW PRAM fully-dynamic algorithm for MST. In *International Parallel Processing Symposium (IPPS)*.

[21] Paolo Ferragina. 1995. A technique to speed up parallel fully dynamic algorithms for MST. *J. Parallel Distrib. Comput.* 31, 2 (1995), 181–189.

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

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

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

[25] Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. 2019. A General Framework for Graph Sparsification. *SIAM J. Comput.* 48, 4 (2019), 1196–1223.

[26] Hillel Gazit. 1991. An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. *SIAM J. Comput.* 20, 6 (1991), 1046–1067.

[27] Barbara Geissmann and Lukas Gianinazzi. 2018. Parallel Minimum Cuts in Near-linear Work and Low Depth. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[28] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. 2020. Faster Algorithms for Edge Connectivity via Random 2-Out Contractions. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[29] Ashish Goel, Michael Kapralov, and Ian Post. 2012. Single pass sparsification in the streaming model with edge deletions. *arXiv preprint* arXiv:1203.4900 [cs.DS] (2012).

[30] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. 2018. Incremental Exact Min-Cut in Polylogarithmic Amortized Update Time. *ACM Trans. Algorithms* 14, 2 (2018), 17:1–17:21.

[31] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[32] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *J. ACM* 48, 4 (2001), 723–760.

[33] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. 2015. Faster Fully-Dynamic Minimum Spanning Forest. In *European Symposium on Algorithms (ESA)*.

[34] Joseph JáJá. 1992. *An introduction to parallel algorithms*. Vol. 17. Addison-Wesley Reading.

[35] Vojtěch Jarník. 1930. O jistém problému minimálním. *Práca Moravské Prírodovedecké Spolecnosti* 6 (1930), 57–63.

[36] Donald B Johnson and Panagiotis Metaxas. 1992. Optimal algorithms for the vertex updating problem of a minimum spanning tree. In *International Parallel Processing Symposium (IPPS)*.

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

[38] David R. Karger, Philip N. Klein, and Robert E. Tarjan. 1995. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *J. ACM* 42, 2 (1995), 321–328.

[39] Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. 2018. Improved Worst-Case Deterministic Parallel Dynamic Minimum Spanning Forest. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[40] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[41] Gary L. Miller and John H. Reif. 1989. Parallel Tree Contraction Part 1: Fundamentals. In *Randomness and Computation*, Vol. 5. 47–72.

[42] Shaunak Pawagi. 1989. A parallel algorithm for multiple updates of minimum spanning trees. In *International Conference on Parallel Processing (ICPP)*.

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

[44] Shaunak Pawagi and IV Ramakrishnan. 1986. An O($\lg n$) algorithm for parallel update of minimum spanning trees. *Inform. Process. Lett.* 22, 5 (1986), 223–229.

[45] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401.

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

[47] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2016. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing (Euro-Par)*.

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

[49] Philip M. Spira and A Pan. 1975. On finding and updating spanning trees and shortest paths. *SIAM J. on Computing* 4, 3 (1975), 375–380.

[50] Xiaoming Sun and David P Woodruff. 2015. Tight bounds for graph problems in insertion streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*.

[51] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. SIAM.

[52] Yung Hyang Tsin. 1988. On handling vertex deletion in updating spanning trees. *Inform. Process. Lett.* 27, 4 (1988), 167–168.

[53] Peter Varman and Kshitij Doshi. 1986. A parallel vertex insertion algorithm for minimum spanning trees. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*.

[54] Peter Varman and Kshitij Doshi. 1988. An efficient parallel algorithm for updating minimum spanning trees. *Theoretical Computer Science (TCS)* 58, 1-3 (1988), 379–397.