Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2025) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

XIII Latin American Algorithms, Graphs, and Optimization Symposium (LAGOS 2025)

How to go from partial to full retroactivity in detail

Cristina G. Fernandes[a], Felipe C. Noronha[a]

[a]*Department of Computer Science, University of São Paulo, Brazil*

Abstract

The concept of retroactivity in data structures was introduced by Demaine, Iacono, and Langerman. In their original paper on retroactivity, they described a way to transform a partially retroactive data structure into a fully retroactive one. Their description is focused on space savings, which require the use of a persistent version of the data structure in question. We focus on the case in which one does not have or does not want to use a persistent data structure. We describe and analyze in detail how to implement their transformation in this case. As a secondary contribution, using our strategy, we implemented a (halfway) retroactive data structure for the incremental minimum spanning forest (MSF) problem, that we make available.

© 2025 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Program Committee of LAGOS 2025.

Keywords: Retroactive data structures; incremental minimum spanning forest; link-cut trees

1. Introduction

Problems in dynamic graphs have many applications, as they can be used to model a variety of real situations where the graph models a network of sorts that is changing over time. A subclass of these problems that are already interesting and challenging are the so-called incremental problems, in which the considered graph is growing with time, through the addition of edges.

The *Minimum Spanning Tree* problem consists of, given a connected graph G with costs on its edges, finding a spanning tree of G with minimum cost. To describe the incremental version of this well-known problem, we consider a generalization on graphs that are not necessarily connected. The *Minimum Spanning Forest (MSF)* problem consists of, given a graph G with costs on its edges, finding a maximal spanning forest of G with minimum cost.

The *incremental MSF* is the problem of keeping track of an MSF in a graph on n vertices that is changing through the addition of new edges with specified costs. We may assume the initial graph is empty. Frederickson [7] described how to solve this problem efficiently using link-cut trees, and addressed the more general dynamic MSF, that also allows deletion of edges. The cost per update of his method is $O(\sqrt{m})$, where m is the number of edges in the graph at the moment of the update.

E-mail addresses: cris@ime.usp.br (Cristina G. Fernandes)., felipe.castro.noronha@hotmail.com (Felipe C. Noronha).

1877-0509 © 2025 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Program Committee of LAGOS 2025.

The concept of retroactivity in data structures was introduced by Demaine, Iacono, and Langerman [3]. Its applications include practical situations where the respective data might be manipulated in imperfect ways, and once in a while there is a need to correct some erroneous operation done, or to perform some operation that was forgotten.

A data structure usually gives support to updates and queries. Generally, the order in which the updates are performed interferes with the state of the data structure. Consider a data structure that starts empty, and suffers a sequence of updates, each with a time stamp that registers the time it occurred. The goal of retroactivity is to allow one to efficiently manipulate this update sequence, and to answer queries not only on the current state of the data structure, but also on the state of the data structure at any time t, that is, the state in which the data structure would be if we applied only the updates in the sequence with time stamp at most t. Specifically, in the context of retroactivity, one wants to be able to insert into the sequence an update with a time stamp t, possibly indicating a time in the past, and to remove some update from the sequence, given its time stamp. We assume the time stamps are all distinct. Moreover, given a time t, one would like to answer queries on the state of the data structure at time t.

If one can efficiently answer only queries on the current state of the data structure, but not on its state at an arbitrary time t, the data structure is said to be *partially retroactive*. In the literature, retroactivity is sometimes used to refer to all variants of retroactivity, and the expression *fully retroactive* is then used to refer to an implementation that provides the complete set of retroactive operations: insertion and removal of updates at any time, as well as answering queries at any time.

Demaine, Iacono, and Langerman [3] described fully retroactive versions of queues, doubly ended queues, priority queue, union-find, and also a more efficient partially retroactive priority queue. They also described how to transform a partially retroactive data structure into a fully retroactive one with an $O(\sqrt{m})$ slowdown per update operation, where m is the length of the update sequence. In general, assuming that certain known conjectures in complexity theory hold, this slowdown is essentially tight [2]. The transformation is done, at first, by keeping $O(\sqrt{m})$ versions of the partially retroactive data structure. Under a condition that assures that there is an efficient persistent version of the data structure in question, they show how to achieve a more effective space consumption without losing in the time consumption. (A persistent data structure is a data structure that always preserves the previous version of itself when it is modified [6].) Years later, Demaine et al. [4] provided a transformation from *time-fusible* partially retroactive data structures into fully retroactive ones, with a logarithmic time slowdown per operation and applied this transformation to obtain a more efficient fully retroactive priority queue.

It is known that a data structure used to solve a dynamic problem, such as the dynamic MSF problem, can be used as a partially retroactive solution for the problem. For instance, an efficient data structure for the dynamic MSF problem works as an efficient partially retroactive MSF solution: the insertions and removals of edges of the graph are the updates, and the query is the cost of an MSF in the current graph. For partial retroactivity, addition or removal of edges at any time t can be made in the present version, and as addition and removal are the inverse of each other, one achieves partial retroactivity. There are efficient implementations for dynamic MSF [10, 11], that assure $O(\lg^4 n)$ time amortized per operation, for simple graphs on n vertices. So the same bound per operation holds for the partially retroactive MSF problem.

Recently, Henzinger and Wu [9] presented lower bounds for the time per operation of a fully retroactive data structure for the MSF problem and for connectivity, under the OMv conjecture [8]. The lower bounds are in terms of the number n of vertices of the graph: for any $\epsilon > 0$, there is no fully retroactive solution that takes $O(n^{1-\epsilon})$ time per operation for these problems. The authors also presented a fully retroactive data structure for connectivity, maximum degree, and MSF in $\tilde{O}(n)$ per operation.

The study of de Andrade Junior and Seabra [1] about retroactivity addresses the incremental MSF problem. In the incremental MSF problem, the only update supported is the addition of edges. So the update sequence, in this case, consists of a series of edge additions. To support full retroactivity, one would have to give support to the insertion of new edge additions at any time, and also to the removal of an edge addition that occurred at some given time t. Their implementation gives support to edge addition at any time t and answers queries at any time t. It does not allow for the removal of an edge addition from the update sequence, so we refer to this as a semi-retroactive incremental MSF solution. (Roditty and Zwick [12], studying strong connectivity, considered yet another version of retroactivity that was called *incremental*, where one is allowed to add an edge only at the present time, not in the past, but can remove from the update sequence any edge addition, given its time stamp.)

The implementation of de Andrade Junior and Seabra is inspired on the aforementioned technique of Demaine, Iacono, and Langerman [3, Theorem 5] for transforming partially retroactive data structures into fully retroactive ones. This technique uses the idea of square-root decomposition, that breaks a time line of length m into $\sqrt{m}$ checkpoints, keeping the state of the data structure at these $\sqrt{m}$ checkpoints, as well as the whole sequence of updates. To answer queries at an arbitrary time t, it computes what is the checkpoint previous to t, as close as possible to t, and then it temporarily applies, to the data structure of this checkpoint, the updates between the checkpoint and t, to be able to answer the query properly. After answering the query, it rolls back these updates to recover the checkpoint state of the data structure. For the purpose of their experimental study, de Andrade Junior and Seabra assumed the length m of the time line was known from the start, and that the updates had time stamps from 1 to m, so they do not ever rebuild the data structure. Also, as Frederickson [7], they used link-cut trees as the data structure for each checkpoint. This leads to an amortized query and edge addition time of $O(\sqrt{m}\lg n)$ in a graph with n vertices. The space used by their implementation is $\Theta(m\sqrt{m})$ because they used a collection of $\Theta(\sqrt{m})$ independent link-cut trees. Our initial goal in this work was to improve de Andrade Junior and Seabra's implementation, allowing the length of the update sequence for the incremental MSF to grow arbitrarily, without increasing its running time or space consumption.

Theorem 5 in [3] has two parts. In the first part, the authors are concerned with the running time, and describe the above mentioned technique. At this point, they do not give details on how to keep the $\sqrt{m}$ checkpoints and the data structures so that, after many insertions and removals of operations, at most $(3/2)\sqrt{m}$ operations occur between consecutive checkpoints, with m changing and all. The second part of the proof is concerned with saving space, and describes how to rebuild the data structure from time to time to keep this invariant. In order to save space, under the same time consumption, their technique uses a persistent version of the data structure in this rebuilding process. There are sophisticated functional implementations of link-cut trees described in the literature [5], based on the use of the so called *fingers*. But we were not focused on the space savings, so we concentrated on looking for a simple way to keep the $\sqrt{m}$ checkpoints and the link-cut trees so that, after many insertions and removals of operations, at most $(3/2)\sqrt{m}$ operations occur between consecutive checkpoints, again with a variable m. We did not find an efficient way along the lines of rebalancing of B-tree nodes, so we considered the rebuilding process. But rebuilding from scratch a whole new collection of link-cut trees from time to time would extrapolate the amortized time by operation. So, we came up with a simple way to use the previous version of the collection of independent link-cut trees to build its new version.

Our main contribution is that our strategy can be applied in general, to transform any partially retroactive data structure into a fully retroactive one, when one is not willing to use or does not have a persistent version of the data structure in question to apply the space saving strategy from Theorem 5 in [3]. It achieves the same slowdown in updates and queries that their technique. We applied our strategy to obtain the improvement we wanted on de Andrade Junior and Seabra's implementation. So, a secondary contribution is an implementation, whose code we made available, of a semi-retroactive version for the incremental MSF, that supports addition of edges and queries at any time, in amortized time $O(\sqrt{m}\lg n)$ per edge insertion and MSF cost query, and uses space $\Theta(m\sqrt{m})$.

The remainder of the paper is organized as follows. In Section 2, we review the strategy of Demaine, Iacono, and Langerman [3] to transform a partially retroactive data structure into a fully retroactive one. Section 3 contains the description of the new proposed rebuilding step, its correctness proof, and its time complexity analysis. In Section 4, we formalize the semi-retroactive incremental MSF and, for completeness, revise how it is implemented using the proposed rebuilding approach. Final remarks are presented in Section 5.

2. From partial to full retroactivity: a brief review

Demaine, Iacono, and Langerman [3] described a way to transform a partially retroactive data structure into a fully retroactive one. Their result considers that the data structures use the RAM model of computation, and work in the pointer-machine model of Tarjan [14]. They also use, in their result, the so called *rollback method*, in which auxiliary information is stored when certain updates are performed on the data structures, so that one can reverse these updates if needed.

For the sake of completeness, we restate their result and describe their method. Then, in the next section, we describe our simplified version of their result.

Theorem 2.1 (Theorem 5 in [3])**.** *Let m denote the number of updates in the current update sequence. Any partially retroactive data structure in the pointer-machine model with constant indegree, supporting $T(m)$-time retroac-*

tive operations and $Q(m)$-time queries about the present, can be transformed into a fully retroactive data structure with amortized $O(\sqrt{m}\ T(m))$-time retroactive operations and $O(\sqrt{m}\ T(m) + Q(m))$-time fully retroactive queries using $O(m\,T(m))$ space.

They define $\sqrt{m}$ checkpoints $t_1, \ldots, t_{\sqrt{m}}$ and maintain $\sqrt{m}$ versions $D_1, \ldots, D_{\sqrt{m}}$ of the partially retroactive data structure, where the structure D_i contains all updates that occurred before time t_i. Each t_i is defined so that, when D_i was constructed, it contained the first $i\sqrt{m}$ of the m updates, for $i = 1, \ldots, \sqrt{m}$. They keep track of the entire sequence of updates.

When a retroactive operation is performed for time t, they perform the operation on all data structures D_i with $t_i > t$, which costs $O(\sqrt{m}\ T(m))$ time. When a retroactive query is made at time t, they find the largest i such that $t_i \leq t$, and perform on D_i all updates from the current update sequence that have time between t_i and t, keeping track of auxiliary information for later rollback. Then they perform the query on the resulting structure, and rollback these updates to restore the state of the structure D_i previous to the query.

They assure that, at any time, between $\sqrt{m}/2$ and $(3/2)\sqrt{m}$ updates have to be performed on D_i to answer any query. This implies that the time to answer a query is $O(\sqrt{m}\ T(m) + Q(m))$. To achieve the $O(m\,T(m))$ space consumption, the way they assure this is by rebuilding $D_1, \ldots, D_{\sqrt{m}}$ from time to time. Let m denote the number of updates in the update sequence when the last rebuilding took place. In the beginning, $m = 0$. By assumption, the partially retroactive data structure has constant indegree, so they use a persistent version of it, obtained according to Driscoll et al. [6]. After $\sqrt{m}/2$ retroactive operations, they update the value of m and rebuild the persistent data structure from scratch in time $O(m\,T(m))$. When rebuilding the persistent data structure for the current number m of updates, they perform the sequence of m updates on a fully persistent version of an initially empty partially retroactive data structure, and keep a pointer D_i to the version obtained after the first $i\sqrt{m}$ updates, for $i = 1, \ldots, \sqrt{m}$. The retroactive updates branch off a new version of the data structure for each modified D_i. The cost for the rebuilding is therefore $O(m\,T(m))$, which adds an amortized cost of $O(\sqrt{m}\ T(m))$ per retroactive operation. They also argue that the space used is $O(m\,T(m))$.

3. Rebuilding process without a persistent data structure

In this section, we concentrate on the case in which one does not want to use a persistent data structure, or such a structure is not available. We describe a rebuilding process that is as efficient, in terms of time, as the original one by Demaine et al. [3], but that does not use a persistent version of the data structure in question.

We refer to a data structure that gives support to retroactive queries, and to retroactive insertions into the update sequence, but not to removals, as a *semi-retroactive* data structure. This kind of data structure is obviously weaker than a fully retroactive one, and is not comparable with a partially retroactive one, because the later gives support to queries only at the present, and to retroactive insertions and removals on the update sequence. For semi-retroactive data structures, we refer to retroactive updates, instead of retroactive operations, as only insertion of updates are allowed.

We will describe two variants of the rebuilding process. The first one is simpler and serves to derive a semi-retroactive data structure from a partially retroactive one. The second one serves to derive a fully retroactive data structure from a partially retroactive one.

3.1. Semi-retroactivity

Our strategy follows the same idea of Demaine el at., but it does not rely on the use of a persistent version of the data structure in question. Also, for semi-retroactivity, we propose the use of slightly different checkpoints and rebuilding moments, that make it easier to implement and analyze the correctness of the strategy.

Let m denote the number of updates in the current update sequence. As we are considering semi-retroactivity, there are no removals of updates, and m is also the number of retroactive operations that happened until now, that is, the total number of retroactive updates.

We will use D_0 to refer to an empty data structure, which is the initial state of the data structure, when $m = 0$. We will rebuild the data structures $D_0, D_1, \ldots, D_{\sqrt{m}}$ every time m is a perfect square, that is, $m = k^2$ for a positive integer k. Since $(k+1)^2 - k^2 = 2k + 1$, the data structures built when $m = k^2$ will be rebuilt after exactly $2k + 1 = \Theta(\sqrt{m})$ retroactive updates.

Let S be the list of updates when $m = k^2$. Let S^+ be the list of subsequent $2k+1$ updates, that arrived after the rebuilding that resulted in $D_0, D_1, \ldots, D_k$, and let S' be the union of S and S^+. Consider these lists sorted by the time of the updates.

When $m = k^2$, a rebuilding occurred and D_i becomes the partially retroactive data structure with the first ik updates in S for $i = 0, 1, \ldots, k$. The retroactive queries and subsequent $2k+1$ retroactive updates in S^+ are treated as in Section 2. When m reaches $(k+1)^2$, it is time to rebuild the data structures. The idea is quite simple. Let D'_0 and D'_1 be two new empty data structures and let D'_{i+2} refer to the current D_i for $i = 0, 1, \ldots, k-1$. Disregard D_k. Let $t'_0 = t'_1 = 0$ and, for $i = 2, \ldots, k+1$, let t'_i be the time of the last update in D'_i. For $i = 1, \ldots, k+1$, apply to D'_i the updates in S' after t'_i so that D'_i stores exactly $i(k+1)$ updates. The resulting $D'_0, D'_1, \ldots, D'_{k+1}$ are the new versions of the data structures for S'.

The key fact that assures that this works is the following.

Lemma 3.1. *For $i = 0, 1, \ldots, k-1$, every update in D_i is within the first $(i+2)(k+1)$ updates for the sequence S' of updates.*

Proof. When $m = k^2$, the data structure D_i contained the first ik updates in S. Let t_i be the time of the last update in D_i at that moment. Since then, the $2k+1$ updates in S^+ occurred, and any of them that had time $t \le t_i$ was applied to D_i. Because $ik + (2k+1) < ik + i + 2k + 2 = (i+2)(k+1)$, even if all the $2k+1$ updates in S^+ were applied to D_i, all updates in D_i would be among the first $(i+2)(k+1)$ updates in S'. □

Note that the statement does not hold with $i+1$ in the place of $i+2$. During the rebuilding, the number of updates applied to D_i to get D'_{i+2} is at most $(i+2)(k+1) - ik = 2k + 2 + i < 3(k+1)$, for $i = 0, 1, \ldots, k-1$. The number of updates applied to D'_1 is exactly $k+1$. That is, within the rebuilding, $O(k) = O(\sqrt{m})$ updates are applied to obtain each D'_i.

3.2. Full retroactivity

To achieve full retroactivity, we must also give support to removals of updates from the update sequence. For this, we are not able to use the perfect squares as the moments of rebuilding, because the possible length of the update sequence is not anymore related to the number of retroactive operations done so far. The length of the update sequence might grow and shrink over time. So the strategy is more similar to the original one of Demaine et al. [3].

Let m be the number of updates in the update sequence S at the moment of a rebuilding that resulted in the partially retroactive data structures $D_0, D_1, \ldots, D_k$, where $k = \lceil \sqrt{m} \rceil$. Let $\underline{k} = \lfloor \sqrt{m} \rfloor$. Then D_i contains the first $i\underline{k}$ updates in S, for $i = 1, \ldots, k-1$, and D_k contains all updates in S. We refer to the updates in S as *old*.

Let $\ell = 1$ if $m = 0$ and $\ell = 2\underline{k} - 1$ if $m \ge 1$. After ℓ retroactive operations, that now might be insertions or removals of updates, we will rebuild the data structures. Let m' be the number of updates in the current sequence S' after these ℓ operations are performed. Let $k' = \lceil \sqrt{m'} \rceil$ and $\underline{k}' = \lfloor \sqrt{m'} \rfloor$.

Claim 3.2. $|\underline{k}' - \underline{k}| \le 1$.

Proof. If $m = 0$, then $m' = m + 1 = 1$, and $\underline{k}' = 1 = \underline{k} + 1$. So suppose that $m \ge 1$, and note that $m - \ell \le m' \le m + \ell$. Then $\sqrt{m-\ell} \le \sqrt{m'}$. But $m - \ell = m - (2\underline{k} - 1) \ge m - 2\sqrt{m} + 1 = (\sqrt{m} - 1)^2$, because $m \ge 1$. Hence $\sqrt{m-\ell} \ge \sqrt{m} - 1$, which implies that $\underline{k}' \ge \lfloor \sqrt{m-\ell} \rfloor \ge \lfloor \sqrt{m} - 1 \rfloor = \underline{k} - 1$. Similarly, $\sqrt{m'} \le \sqrt{m+\ell}$, and $m + \ell = m + 2\underline{k} - 1 < m + 2\sqrt{m} + 1 = (\sqrt{m}+1)^2$. Thus $\sqrt{m+\ell} < \sqrt{m} + 1$, which implies that $\underline{k}' \le \lfloor \sqrt{m+\ell} \rfloor \le \lfloor \sqrt{m} + 1 \rfloor = \underline{k} + 1$. □

If $\underline{k}' \ge \underline{k}$, then $i\underline{k} + 2\underline{k} - 1 \le i\underline{k}' + 2\underline{k}' - 1 < (i+2)\underline{k}'$. Hence all the $i\underline{k}$ old updates that were not removed are within the $(i+2)\underline{k}'$ first updates in S', even if all the at most $2\underline{k} - 1$ new updates inserted are before t_i.

If $\underline{k}' = \underline{k} - 1$, then $m' < m$, which means that at most $\underline{k} - 1$ of the $2\underline{k} - 1$ operations that occurred since the last rebuilding are insertions. Also, as $k' \le \underline{k}' + 1$, we only need to use D_i to obtain D'_{i+2} for $i + 2 \le k'$, which means that $i \le k' - 2 < \underline{k}'$. So, $i\underline{k} + \underline{k} - 1 = i(\underline{k}' + 1) + \underline{k}' = (i+1)\underline{k}' + i < (i+2)\underline{k}'$.

Hence, we can proceed essentially as in the previous subsection. Let D'_0 and D'_1 be two new empty data structures and let D'_{i+2} refer to the current D_i for $i = 0, 1, \ldots, k' - 1$. Disregard D_k. Let $t'_0 = t'_1 = 0$ and, for $i = 2, \ldots, k'$, let t'_i be the time of the last update in D'_i. For $i = 1, \ldots, k' - 1$, apply to D'_i the updates in S' after t'_i so that D'_i stores exactly $i\underline{k}'$

updates, and apply to $D'_{k'}$ all the updates in S' after $t'_{k'}$. The resulting $D'_0, D'_1, \ldots, D'_{k'}$ are the new versions of the data structures for S'.

Note that, within the rebuilding, the number of updates we perform on D'_1 is $\underline{k}'$, the number of updates we perform on $D'_{k'}$ is $m' - m \leq 2\underline{k}' - 1$, and, for $2 \leq i \leq k' - 1$, we perform at most $i\underline{k}' - (i-2)\underline{k} + 2\underline{k} - 1 = i(\underline{k}' - \underline{k}) + 4\underline{k} - 1 \leq i + 4\underline{k} - 1 \leq k' + 4\underline{k} - 2 \leq 5k' + 2$ updates. For every i, this number is $O(k') = O(\sqrt{m'})$.

The time to execute a retroactive operation remains the same, despite the change in the number of operations between rebuildings. Let m and $\bar{m}$ be the number of updates in the sequence at the last rebuilding and when an operation is done, respectively. For a retroactive insertion or removal of an update, the amortized time is $O(\sqrt{m}\ T(\bar{m})) = O(\sqrt{\bar{m}}\ T(\bar{m}))$. Let $\underline{k} = \lfloor\sqrt{m}\rfloor$ and $\ell = 2\underline{k} - 1$. Because $m - \ell < \bar{m} < m + \ell$, we have that $\underline{k} = O(\sqrt{\bar{m}})$. For a retroactive query, the number of updates applied to the appropriate D_i and rolled back is $O(\underline{k}) = O(\sqrt{\bar{m}})$, so the time is $O(\sqrt{m}\ T(m) + Q(m))$.

As for the space used by our strategy, assuming that the space used by the partially retroactive data structure is linear, each D_i uses space $\Theta(ik)$, and thus the total space used by $D_0, D_1, \ldots, D_k$ is $\Theta(m\sqrt{m})$. The space used by Demaine et al. [3] strategy, that relies on a persistent data structure, is $O(m\,T(m))$, where $T(m)$ is the time for a retroactive update in the partially retroactive data structure.

4. Semi-retroactive incremental MSF

The approach of de Andrade Junior and Seabra [1] for the semi-retroactive incremental MSF solution offers support to the following interface:

- `add_edge`(u, v, w, t): add to the graph G, at time t, an edge of cost w and endpoints u and v;
- `get_msf`(t): return a list with the edges of an MSF of G at time t.

To implement this, one needs to keep an incremental MSF, which, in this case, is a partial retroactive data structure. Its interface is pretty similar to the semi-retroactive version, and the only difference is that we drop the argument for time t in both the edge addition and query operations. This can be implemented using link-cut trees [13] as an underlying structure for maintaining the current MSF. Specifically, every time a new edge uv is added, we can check in the link-cut trees if this new edge would form a cycle with the stored forest. If not, then we proceed to add it to the forest. Otherwise, we find an edge e with maximum cost on the path between u and v in the stored forest, and if the cost of the new edge uv is smaller than the cost of e, we remove e and add uv to the forest. Also, when a query for MSF is performed, we simply return all the edges currently stored in the link-cut trees.

Following on, to implement the semi-retroactive version of the incremental MSF, as described in Section 2, the idea of square-root decomposition is used to divide the time line of length m in blocks of size $\sqrt{m}$. Because of the restrictions imposed by de Andrade Junior and Seabra — that m is known beforehand and that each operation time is an integer in the interval $[1, m]$ — it is possible to avoid rebuilding, and to build these $\sqrt{m}$ blocks right up front, as the first step in the structure initialization. Each of these blocks is defined by a checkpoint t_i such that $t_i = i\sqrt{m}$, with $i \in [1, \sqrt{m}]$. Then, each checkpoint t_i is followed by a respective incremental MSF D_i, where D_i has all the edge insertions that took place before the moment t_i. An empty incremental MSF D_0 is also used.

From that, the implementation of de Andrade Junior and Seabra follows as expected. The operation `add_edge`(u, v, w, t) is performed by adding the respective edge to each D_i such that $t < t_i$, for $i \in [1, \sqrt{m}]$. The `get_msf`(t) consists of finding the largest i such that $t_i < t$, and then performing all the insertions that take place between t_i and t on D_i. After that, it is possible to return the current MSF stored in D_i and then roll back these last performed insertions. The empty incremental MSF D_0 is used when t is smaller than t_1.

Now, let us take a look at the time consumption of this approach. Recall that n denotes the number of vertices of the graph, and therefore in the link-cut trees. First of all, the query for the edges in the link-cut trees costs $O(n)$, and all the other routines used from the link-cut trees have an amortized cost of $O(\log n)$ per operation. For the `add_edge` routine, in the worst case, we have to add one new edge to each D_i, hence its amortized time consumption is $O(\sqrt{m}\ \log n)$. Finally, the time consumption of the `get_msf` is $O(n + \sqrt{m}\ \log n)$, because of the updates that need to be applied and rolled back, and the query for the edges in a versions of the link-cut trees.

The development of the idea presented in this paper was driven by the desire to overcome the limitations in de Andrade Junior and Seabra's solution for the semi-retroactive MSF problem. The main difference is that we implement the rebuilding steps, and hence we do not restrict the amount of operations or their time range. The rebuilding steps are implemented following the approach presented in Section 3.1. Edge insertions and queries are treated similarly to their implementation, but now the checkpoints change during the process, as the rebuildings happen.

To emphasize the simplicity of the rebuilding step, we present below the idea in pseudocode, using the notation from Section 3.1. The procedure receives an integer k, a sequence D with the link-cut trees $D_0, \ldots, D_k$, the sequence t where t_i is the last time stamp of an edge in D_i for $i = 0, \ldots, k$, and the current sequence S with $(k+1)^2$ edge addition pairs (e, s), stored for instance in a balanced binary search tree with the time stamp s as key. It returns the new block size $k+1$, the sequence D' with the link-cut trees $D'_0, \ldots, D'_{k+1}$ and the sequence t' where t'_i is the last time stamp of an edge in D'_i for $i = 0, \ldots, k+1$. In this pseudocode, for a pair $p = (e, s)$ in S, we use p.time to refer to s. The procedure newIncrementalMSF returns a new data structure representing a spanning forest with no edges. It takes O(1) time in our implementation. The procedure kth(S, i) returns the element in S with the ith smallest key, in time O(log k), because S has O(k^2) elements. The procedure addEdges(S, t_s, t_f, F) updates the MSF stored in F considering the addition of all edges in S with time stamp more than t_s and at most t_f. It takes time O(log $k + \ell \log n$), where ℓ is the number of edges added.

Algorithm 1 Rebuilding procedure

```
1:  function rebuild(k, D, t, S)
2:      D'_0 ← newIncrementalMSF()
3:      D'_1 ← newIncrementalMSF()
4:      for i ← 2 to k + 1 do
5:          D'_i ← D_{i−2}
6:      end for
7:      t_{−1} ← 0                                          ▷ sentinel
8:      t'_0 ← 0
9:      for i ← 1 to k + 1 do
10:         p ← kth(S, i(k + 1))                            ▷ i(k + 1)th edge in S
11:         t'_i ← p.time                                   ▷ time stamp of the i(k + 1)th edge in S
12:         addEdges(S, t_{i−2}, t'_i, D'_i)
13:     end for
14:     return k + 1, D', t'
15: end function
```

The running time is dominated by the insertion operations on the incremental MSFs. As argued in Section 3.1, this process will execute O(m) such operations and, because each of these operations has an amortized cost of O(log n), the total amortized cost of rebuild is O($m \log n$). This cost, distributed over the $\Theta(\sqrt{m})$ operations that take place between two rebuildings, adds an O($\sqrt{m}\ \log n$) amortized consumption time per operation.

5. Final remarks

During our study of the work of de Andrade Junior and Seabra, we noticed that they did not really implement full retroactivity, because their implementation does not allow for removals from the update sequence. Even though the problem considered is incremental, a fully retroactive version of the problem requires the ability to remove a previous addition from any point in the update sequence, a capability their implementation lacks.

Note that a version allowing for the removal of edge additions should not be confused with an implementation of a retroactive dynamic MSF. A dynamic structure for an MSF not only allows for the maintenance of edge additions but also edge removals in the update sequence.

Our current implementation also does not give support to removals of edge additions. The removal of an edge insertion with time stamp t from the update sequence requires the removal of this edge from each data structure D_i such that $t < t_i$. However, since the data structure D_i is for incremental MSF, we do not have support to edge removals.

The approach we presented to avoid the need for a fixed m, as seen in de Andrade Junior and Seabra's implementation, demonstrates an interesting method for adapting a square-root decomposition technique to our advantage. This technique could be further explored in other applications as well.

Finally, the algorithm of Holm, de Lichtenberg, and Thorup [10] maintains dynamic graphs efficiently. Their algorithm is also based on link-cut trees. It would be interesting to use their ideas to achieve an implementation of an efficient fully retroactive incremental MSF.

References

[1] de Andrade Júnior, J.W., Seabra, R.D., 2022. Fully retroactive minimum spanning tree problem. The Computer Journal 65, 973–982.
[2] Chen, L., Demaine, E.D., Gu, Y., Williams, V.V., Xu, Y., Yu, Y., 2018. Nearly optimal separation between partially and fully retroactive data structures, in: Proc. of the 16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), pp. 33:1–33:12.
[3] Demaine, E.D., Iacono, J., Langerman, S., 2007. Retroactive data structures. ACM Transactions on Algorithms 3, 13.
[4] Demaine, E.D., Kaler, T., Liu, Q.C., Sidford, A., Yedidia, A., 2015. Polylogarithmic fully retroactive priority queues via hierarchical checkpointing, in: Proc. of the Workshop on Algorithms and Data Structures (WADS), pp. 263–275.
[5] Demaine, E.D., Langerman, S., Price, E., 2008. Confluently persistent tries for efficient version control, in: Proc. of the Scandinavian Workshop on Algorithm Theory (SWAT), pp. 160–172.
[6] Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E., 1989. Making data structures persistent. Journal of Computer and System Sciences 38, 86–124.
[7] Frederickson, G.N., 1983. Data structures for on-line updating of minimum spanning trees, in: Proc. of the 15th Annual ACM Symposium on Theory of Computing (STOC), pp. 252–257.
[8] Henzinger, M., Krinninger, S., Nanongkai, D., Saranurak, T., 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture, in: Proc. of the 47th Annual ACM Symposium on Theory of Computing (STOC), pp. 21–30.
[9] Henzinger, M., Wu, X., 2021. Upper and lower bounds for fully retroactive graph problems, in: Proc. of the 17th International Symposium Algorithms and Data Structures (WADS), pp. 471–484.
[10] Holm, J., de Lichtenberg, K., Thorup, M., 2001. Poly-logarithmic deterministic fullydynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Journal of the ACM 48, 723–760.
[11] Holm, J., Rotenberg, E., Wulff-Nilsen, C., 2015. Faster fully-dynamic minimum spanning forest, in: Proc. of the European Symposium on Algorithms (ESA), pp. 742–753.
[12] Roditty, L., Zwick, U., 2016. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. SIAM Journal on Compututing 45, 712–733.
[13] Sleator, D.D., Tarjan, R.E., 1981. A data structure for dynamic trees, in: Proc. of the 13th Annual ACM Symposium on Theory of Computing (STOC), pp. 114–122.
[14] Tarjan, R.E., 1979. A class of algorithms which require nonlinear time to maintain disjoint sets. The Journal of Computer and System Sciences 18, 110–127.