universität wien

DISSERTATION | DOCTORAL THESIS

Titel | Title

Engineering Efficient Algorithms for the Analysis of Dynamic Data

verfasst von | submitted by

Lara Chiara Ost BSc ETH MSc ETH MSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Doktorin der technischen Wissenschaften (Dr.techn.)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt | Field of study as it appears on the student record sheet:	Informatik
Betreut von | Supervisor:	Univ.-Prof. Dr. Monika Henzinger Ass.-Prof. Dr. Kathrin Hanauer B.Sc. M.SC.

ABSTRACT

The analysis of large data sets is central to numerous fields, each posing different requirements on the kinds of analyses that are performed. One aspect many areas share is the dynamic nature of data: data sets change as new data arrives and outdated data is removed. Consider, for example, graphs representing social networks, where edges appear and disappear as people's interactions evolve, or communication graphs, where weighted edges represent changing communication demand. Any kind of continuous data stream can also be considered dynamic data. Examples are financial data, physiological data such as in electrocardiography, and seismic data. We study algorithms for the analysis of dynamic data in different application areas, namely in topological data analysis, private data analysis and reconfigurable networks.

Persistent homology is a key concept in topological data analysis, describing topological features across multiple scales. We introduce a data structure for maintaining the persistent homology of time series data, based on a novel tree-like representation called the banana tree. It supports local operations such as changing the value of an item or inserting a new item, and topological operations, i.e., cutting and concatenating, which each can be performed in time linear in the number of changes to the output. We assess its performance and properties in an experimental study on a variety of synthetic and real-world time series. The results indicate that banana trees outperform the state-of-the-art static algorithms on unbiased random and quasi-periodic data. They also reveal connections between the performance and structure of the banana trees. Our results suggest practical utility of banana trees on real-world data sets, as they share structural properties with random walks.

Private data analysis aims to publish analyses of a data set while protecting the sensitive data of users who contribute their information. Differential privacy is a framework that offers such protections. We study differential privacy for graph problems under continual observation, i.e., for dynamic graphs. We present event-level ε-differentially private algorithms for subgraph counting problems, minimum spanning tree, minimum cut, maximum matching and densest subgraph on partially dynamic inputs. These are the first differentially private algorithms for such problems with additive error poly-logarithmic in the input length T. Finally, we show that on user-level with fully dynamic inputs the additive error must be linear in the value of the output's solution range for a wide variety of graph problems.

Reconfigurable networks in datacenters provide direct connections between racks, adapted to the communication demand, which form a set of k edge-disjoint matchings. We propose dynamic algorithms to efficiently maintain these disjoint matchings while maximizing the demand served. We present seven (batch-)dynamic algorithms and compare them to static ones in extensive experiments on 176 synthetic and 39 real-world network traces. Our dynamic algorithms significantly improve the running time and reduce the number of changes to the network's configuration, while retaining solution quality.

ZUSAMMENFASSUNG

Die Analyse von großen Datensätzen ist auf vielen Gebieten von zentraler Bedeutung, wobei sich die Anforderungen an die Art der Analyse stark unterscheiden können. Ein gemeinsamer Aspekt über Anwendungen hinweg ist die Dynamik der Datensätze: neue Daten kommen hinzu und obsolete Daten werden entfernt. Beispiele finden sich in Graphen, die soziale Netzwerke repräsentieren, wo Änderungen in den Interaktionen der Teilnehmer Kanten erscheinen und verschwinden lassen, oder auch in Kommunikationsnetzwerken, deren gewichtete Kanten den fluktuierenden Kommunikationsbedarf modellieren. Allgemein kann jeder Datenstrom als dynamisch angesehen werden, zum Beispiel finanzielle Daten, physiologische Daten und seismische Daten. Diese Dissertation behandelt Algorithmen zur Analyse von dynamischen Daten in unterschiedlichen Anwendungsgebieten: in topologischer Datenanalyse, privater Datenanalyse und rekonfigurierbaren Netzwerken.

Persistente Homologie ist ein Schlüsselkonzept in topologischer Datenanalyse, das topologische Eigenschaften auf vielen Skalen beschreibt. Wir führen eine Datenstruktur zur Berechnung der persistenten Homologie von Zeitreihen ein, die auf einer neuartigen Baum-ähnlichen Struktur, dem Bananenbaum, basiert. Diese Datenstruktur unterstützt lokale Operationen, wie Änderung eines Wertes und Einfügen eines Punktes, und topologische Operationen, die Zeitreihen trennen oder zusammenfügen. Die Laufzeit für diese Operationen skaliert linear mit den Änderungen in der Ausgabe. Eine experimentelle Studie auf synthetischen und realen Zeitreihen zeigt, dass Bananenbäume auf symmetrischen Irrfahrten und quasi-periodischen Reihen Änderungen an der Eingabe schneller verarbeiten als der bisher schnellste Algorithmus. Die Ergebnisse zeigen auch Zusammenhänge zwischen der Performanz und der Struktur der Bananenbäume auf. Bananenbäume auf realen Zeitreihen ähneln denen auf symmetrischen Irrfahrten in ihren strukturellen Eigenschaften, was eine gute Anwendbarkeit nahelegt.

Private Datenanalyse zielt darauf ab, Analysen von Datensätzen zu erzeugen und gleichzeitig sensible Daten zu schützen. Differential Privacy ist ein Ansatz, der solche Analysen ermöglicht. Wir betrachten Differential Privacy für Graphprobleme im Kontext der Continual Observation; das heißt, für dynamische Graphen. Wir beschreiben ε-differentiell private Algorithmen auf Event-Level für die Probleme Zählen von Subgraphen, minimaler Spannbaum, kleinster Schnitt, maximales Matching und dichtester Subgraph auf partiell dynamischen Eingaben. Dies sind die ersten differentiell privaten Algorithmen für solche Probleme mit additivem Fehler der poly-logarithmisch mit T, der Länge der Eingabe, skaliert. Schließlich zeigen wir, dass der additive Fehler von Algorithmen auf User-Level für viele Graphprobleme linear im maximalen Funktionswert ist.

Rekonfigurierbare Netzwerke in Datencentern ermöglichen direkte Verbindungen zwischen Racks, angepasst an den Kommunikationsbedarf. Diese Verbindungen formen eine Menge an k disjunkten Matchings. Wir präsentieren dynamische Algorithmen, die diese disjunkten Matchings berechnen und dabei die kommunizierte Datenmenge maximieren. Wir vergleichen sieben

(batch-)dynamische Algorithmen mit statischen in ausführlichen Experimenten auf 176 synthetischen und 39 realen Netzwerk-Traces. Unsere Algorithmen verbessern die Laufzeit deutlich und reduzieren die nötigen Änderungen an der Netzwerkkonfiguration, ohne dabei an Lösungsqualität zu verlieren.

ACKNOWLEDGMENTS

First and foremost I am deeply grateful to my supervisors Monika Henzinger and Kathrin Hanauer for their continued support and patience throughout the last few years. Thank you, Monika, for believing that a chemist could do computer science research and for getting me in contact with such a variety of fascinating problems. Thank you, Kathrin, for your advice and guidance towards the end of my PhD.

A thesis needs to be reviewed, and I thank Gramoz Goranci and Henning Meyerhenke for agreeing to take on this task.

This thesis is the result of many fantastic collaborations: the banana trees would not have grown without the contributions of Herbert Edelsbrunner and Sebastiano Cultrera di Montesano; Hendrik Fichtenberger's guidance was integral to my first steps in differential privacy; connecting the disjoint matchings to their application would not have been possible without Stefan Schmid. There were other projects that did not make it into this thesis; still, they taught me much and I am grateful to my collaborators Christian Schulz, Darren Strash, Krishnendu Chatterjee and Wolfgang Dvořák for the many lessons.

In 2023 I spent some months in Copenhagen working with Eva Rotenberg and Ivor van der Hoog at DTU. Thank you for being amazing hosts and great collaborators, and for your encouragement in stressful times. Thank you also to Teresa Steiner for establishing this connection and endlessly simplifying my search for housing.

Throughout my PhD I had the pleasure of working alongside some exciting people: Sricharan AR, Sophia Heck, Ali Momeni, Martin Schirneck, Éva Szilágyi, Max Vötsch and many more. They made my time in Vienna fun in the good times and bearable in the hard times. I am particularly grateful to Alexander Noe, Marcelo Fonseca Faraj, Stefan Neumann and Antoine El-Hayek for being the most pleasant office-mates. Studying at the University of Vienna comes with a lot of bureaucracy and I am thankful to Ulrike Frolik-Steffan's and Bettina Hiebl's support in managing this. The experiments presented in this thesis would not have been possible without Rudolf Hürner's technical assistance and my fellow machine-users' patience.

While I was busy doing research my life decided to take some detours and I am lucky to have had a network of great people who made navigating them so much easier. I cannot mention everyone here, but my biggest thanks go to my friends Konrad von Kirchbach, Liyin Cai, and especially Nadja Singer with Amy, for their invaluable emotional support and our general fun times.

BIBLIOGRAPHIC NOTE

Some of the results in this thesis were previously published in conference proceedings. The chapters are based on the following publications and manuscripts:

- **Chapters 3 and 4:** S. Cultrera di Montesano, H. Edelsbrunner, M. Henzinger, L. Ost,
  Dynamically maintaining the persistent homology of time series,
  in *Proceedings of the Symposium on Discrete Algorithms (SODA)* (2024).
- **Chapter 5:** L. Ost, S. Cultrera di Montesano, H. Edelsbrunner,
  Banana trees for the persistence in time series experimentally,
  Manuscript, available on arXiv:2405.17920 [cs.DS] (2024).
- **Chapter 6:** H. Fichtenberger, M. Henzinger, L. Ost,
  Differentially private algorithms for graphs under continual observation,
  in *Proceedings of the European Symposium on Algorithms (ESA)* (2021).
- **Chapter 7:** K. Hanauer, M. Henzinger, L. Ost, S. Schmid,
  Dynamic demand-aware link scheduling for reconfigurable datacenters,
  in *Proceedings of the International Conference on Computer Communications (INFOCOM)* (2023).

FUNDING ACKNOWLEDGMENT. This research has been funded by the Vienna Graduate School on Computational Optimization (VGSCO), FWF-Project No. W1260-N35. The author has further received funding by the UniVie Doctoral School Computer Science (DoCS).

CONTENTS

1 INTRODUCTION

Data plays an essential role in our society. Massive amounts are produced at any moment, be it through the use of smartphones and cars, or in healthcare, scientific research and financial markets. This data has tremendous value: in a 2017 article the Economist called data "the world's most valuable resource" [1]. But more importantly, the availability of data impacts society beyond its economic worth: for example, in healthcare the analysis of medical data can lead to improvements in diagnosis, prevention, treatment and health monitoring [2–4]. In any case, data in its raw form is usually meaningless and its analysis is key to extract value and insight. Given the immense volume of data, efficient algorithms are necessary to provide analyses in a timely manner and within reasonable bounds on computational resources.

Data differs vastly across domains, but a common aspect is its dynamic nature: new data arrives constantly and other data becomes outdated. Graph structured data offers many examples: in infection networks edges represent infection events and they appear as the disease spreads through a community; in social networks edges appear and disappear as participants follow and unfollow each other; the communication demand in a datacenter fluctuates and can be modeled by a graph with changing edge weights. Other examples can be found in healthcare with the monitoring of heart rate and other physiological data, or in financial markets, where the values of assets fluctuate as trades are made.

Taking this "dynamicness" into account when designing data analysis algorithms has significant benefits. It allows us to avoid recomputation by instead updating an existing solution, which saves on computational resources and cuts down on processing time. The former leads to more efficient use of hardware, and the latter may be crucial in monitoring applications, where a timely response to an event represented in the data is desired. Additionally, some problems are inherently dynamic: for instance, when performing a privacy-preserving analysis of an evolving data set, treating it as a static data set with each update can leak private information. It appears that there is a clear need for algorithms that can work with changing input data.

We call such an algorithm a *dynamic algorithm*, schematically illustrated in Figure 1.0.1. It maintains (a representation of) a solution to a problem $\mathcal{P}$ while the input data changes. After preprocessing an initial input to construct its internal state, the algorithm receives updates that modify the input data. Each update is processed to alter the internal state such that it reflects the modified input. At any point the algorithm may be queried for the solution to $\mathcal{P}$ on the current input. Some algorithms process updates in groups; these are the *batch-dynamic algorithms*.

The requirements on dynamic algorithms differ depending on the problem and the use case. Fundamentally, there can be differences in the type of input data: we study problems on real-valued time series and on graphs. Even when considering one type of input data there are still differences in what updates are allowed. On graphs, for example, we may have edge insertions

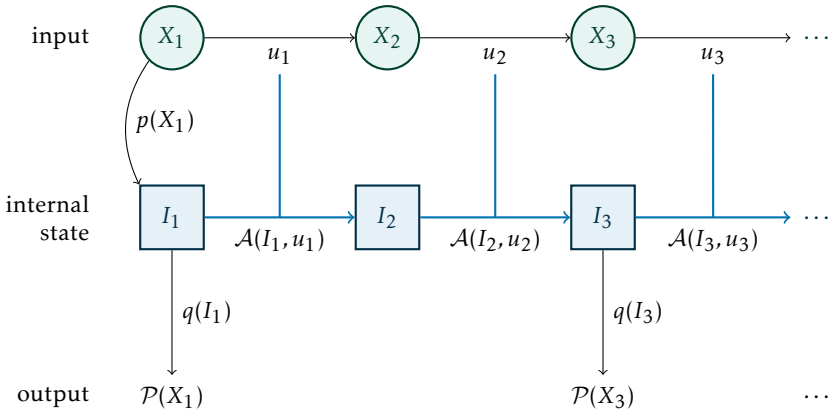

Figure 1.0.1: A dynamic algorithm to compute $\mathcal{P}$ on inputs $X_1, X_2, \ldots$. The initial input X_1 is preprocessed to obtain the initial internal state $I_1 = p(X_1)$. The input X_1 is changed via a sequence of updates $u_1, u_2, \ldots$, yielding $X_2, X_3, \ldots$. Based on its internal state I_j and the update u_j, the dynamic algorithm computes the internal state $I_{j+1} = \mathcal{A}(I_j, u_j)$, which represents input X_j. At any point the algorithm may be queried to output the solution $q(I_i) = \mathcal{P}(X_i)$. Not every update needs to be followed by a query.

and deletions, or only allow changes to edge weights[a]. In the same vein, we usually distinguish between *partially dynamic* and *fully dynamic* settings. In the former, data is either only added or only deleted, while in the latter data may be both added and deleted. Finally, what makes a "good" algorithm also differs depending on the application. Common quality criteria are the time to process an update or query the solution, the solution quality, and the *recourse*, i.e., the amount of changes to the solution due to an update. Different applications may prioritize different quality criteria, e.g., an application may require an exact solution and take increased update time into account, while others may sacrifice solution quality for faster update times.

This thesis studies dynamic data analysis problems from three application areas, namely topological data analysis of time series, private analysis of dynamic graphs and scheduling reconfigurable datacenters. We focus on the development of algorithms with good practical performance. Our contributions are as follows:

1. Topological data analysis of time series (Chapters 3 to 5): a core concept in topological data analysis is persistent homology and its representation in the persistence diagram. We introduce the first data structure to maintain the persistence diagram of a time series under a large set of updates. It allows changes to the value of a time step, as well as cutting and concatenating time series. The updates are processed in time $O(\log n + k)$, where k is the number of changes to the persistence diagram and n is bounded by the length of the time series. This data structure is based on banana trees, which are a novel representation of path-decomposed binary trees and explicitly represent the persistence diagram. We demonstrate its practical performance in experiments on synthetic and real-world data: on random walks and quasi-periodic series our data structure outperforms the state of the art in terms of

a While the latter can be emulated by the former—delete an edge, then insert it with the new weight—handling a weight change directly allows for increased efficiency.

running time by several orders of magnitude. Finally, we analyze how the performance is connected to structural properties of the banana trees, and show that banana trees of real-world time series resemble those on random walks and quasi-periodic series. This indicates strong performance on real-world data sets and practical utility.

2. Private analysis of dynamic graphs (Chapter 6): we study differential privacy on graphs under continual observation, that is, on dynamic graphs. This framework allows to release the results of an analysis without leaking sensitive information contained within the graph. In the event-level setting we present partially dynamic differentially private algorithms for several graph problems, including subgraph counting problems, the degree histogram, minimum spanning tree, minimum cut and densest subgraph. These are the first such algorithms with additive error poly-logarithmic in the length of the input sequence, i.e., the number of queries; the algorithms for minimum spanning tree, minimum cut and densest subgraph are overall the first dynamic differentially private algorithms for these problems. We complement these algorithms with lower bounds, which show that the error has to be at least logarithmic in the length of the input sequence. For the stronger user-level setting we show that the additive error has to be linear in the maximum function value for a wide range of problems, which precludes differentially private algorithms on sufficiently long inputs.

3. Dynamically scheduling reconfigurable datacenters (Chapter 7): modern technologies allow networks in datacenters to be adapted to the communication demand. Determining a configuration of such a network that maximizes the transmitted data requires efficient algorithms. We give dynamic algorithms for the disjoint matchings problem motivated by this application and are the first to study it in the dynamic setting. In experiments on real-world instances they outperform the best known static algorithms, even though the batches update almost all of the graph, and they yield high quality solutions with fast running times and small recourse.

In the following chapter we give a brief introduction to each of the three application areas and summarize the problem statements and our results.

2 OVERVIEW AND SUMMARY

This chapter provides an overview of the problems studied in this thesis and of our results. We begin with topological data analysis of time series in Section 2.1, continue with private analysis of dynamic graphs in Section 2.2, and finish with reconfigurable networks in Section 2.3. Section 2.4 places our work in context of later results and briefly discusses some open problems.

2.1 Topological Data Analysis of Time Series

Motivation

Topological data analysis aims to determine the shape of a data set and study its topological properties. Consider, for example, the set of points in Figure 2.1.1a, which takes the shape of an annulus. We would like a more reliable approach than visual inspection to find this shape. Similarly, in the noisy time series in Figure 2.1.1b we may want to recover the underlying signal. We could smooth the series by removing small oscillations, but then how do we decide what is "small"? Topological data analysis offers methods to solve these tasks. A central tool is *persistent homology* [5], which considers topological features at all scales and identifies those that persist over a large range of scales. It is a versatile methodology that can be applied to many kinds of data, such as in medicine [6–10], biology [11–13], finance [14], computer vision [15] and chemistry [16–19], to name a few.

Informally, *homology* classifies a shape by its number of connected components, cycles, holes and higher-dimensional analogs. *Persistent homology* considers a representation of a data set parameterized by some scale parameter α. For the point set in Figure 2.1.1a this may be a set of balls of radius α centered on the points, and for the time series in Figure 2.1.1b the set of times t for which $f(t) < \alpha$. Persistent homology then tracks how the homology changes as the parameter α is swept from its smallest to its greatest value. A

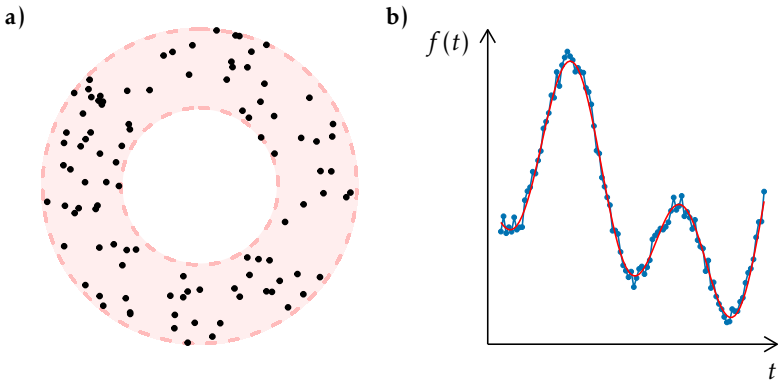

Figure 2.1.1: **a)** A set of points sampled from the annulus shaded in red. **b)** A noisy time series in blue, with the underlying noiseless signal in red.

feature, e.g., a cycle, that exists over a large range of α can be considered a key feature of the data set's shape.

We are interested in persistent homology for time series. In this context persistent homology is often applied to an embedding of the time series into higher dimensions. Examples include the work by Perea et al. [12, 20], who use persistent homology of a time-delay embedding to detect periodicity in time series, and that by Chung et al. [8], who use persistent homology of an embedding to classify sleep stages from heart rate data. However, it may also be applied directly to the time series itself. This has been used to detect patterns in gene expression data [11], analyze heart rate data to classify sleep stages [8] or detect stroke episodes [9], predict weight loss success from accelerometer data [10], and to engineer topological features for time series forecasting [21]. The two approaches can, of course, be combined: for example, Chung et al. [8] use results from both approaches to train a classifier on heart rate data.

We consider the second approach, i.e., computing the persistent homology of the time series itself. More precisely, we consider the *extended* persistent homology [22] of time series data. We introduce a new data structure to maintain a representation of extended persistent homology called the *augmented persistence diagram* as the input time series undergoes changes.

Defining Extended Persistent Homology of Time Series

By time series we mean a list $c_1, c_2, \ldots, c_m$ of real numbers. To define the augmented persistence diagram for this list we view it as a continuous map on a closed interval. We write $f(i) = c_i$ for the value of *item* i. By linearly interpolating between consecutive values c_i we obtain a piecewise-linear map $f : [1, m] \to \mathbb{R}$. We assume all c_i to be distinct, which makes f *generic*. Appropriate tie-breaking rules or small perturbations ensure that this assumption holds in practice[a]. Extended persistent homology of time series depends solely on the *critical items* of the series. These are the *minima*, whose adjacent items have greater value, and the *maxima*, whose adjacent items have smaller value. The remaining items are called *non-critical*. Endpoints require special treatment, but for the purposes of this section we think of them as minima or maxima.

The extended persistent homology of functions on a closed interval is a special case of persistent homology and is entirely described by the connected components of *sub- and superlevel sets*. In this introduction we provide a simplified explanation that is sufficient to summarize our contribution. For details see Section 3.2.1 and we refer to the book by Edelsbrunner and Harer [24] for a general introduction to persistent homology.

Extended persistent homology is encoded in the *extended persistence diagram*, which we can obtain in a two-phase process. This process tracks the evolution of connected components of the sublevel sets $f_t = f^{-1}(-\infty, t]$, the set of $x \in [1, m]$ with $f(x) \leq t$, and superlevel sets $f^t = f^{-1}[t, \infty)$, the set of $x \in [1, m]$ with $f(x) \geq t$, as t increases and decreases, respectively. In the first phase we sweep t from $-\infty$ to ∞ and observe how connected components of f_t appear and disappear. When t passes the value $f(a)$ of a minimum a a new component

a Stability of persistence diagrams with respect to small perturbations, as proved by Cohen-Steiner et al. [23], guarantees that this has negligible impact on the output.

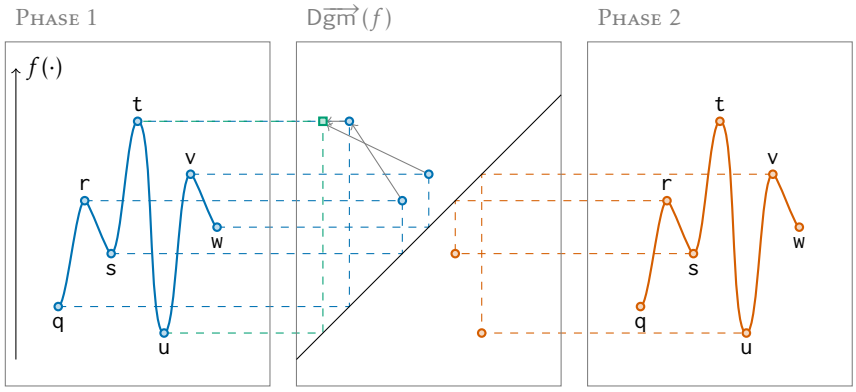

Figure 2.1.2: A map f along with its (augmented) persistence diagram $\overrightarrow{\mathrm{Dgm}}(f)$. On the diagonal line birth is equal to death. Tracing the dashed lines shows how components relate to points. Blue circles (above the diagonal) are points in the ordinary subdiagram $\mathrm{Ord}(f)$, the green square is the sole point in the essential subdiagram $\mathrm{Ess}(f)$, and red circles (below the diagonal) are points in the relative subdiagram $\mathrm{Rel}(f)$. Arrows indicate that components corresponding to points merged.

is *born*; when t passes the value $f(b)$ of a non-endpoint maximum b two components merge into one. We say that the component born at a greater value of t *dies at* b. Consider two components, one born at a and one born at p with $f(p) < f(a)$ that merge at a maximum b: the component born at a dies at b and the merged component is said to have been born at p. The component born at the global minimum does not die in phase one. In the second phase we repeat the process for the connected components of the superlevel sets f^t as we sweep t from ∞ to $-\infty$. Components are now born at maxima and die at minima that are not endpoints. Alternatively, we can repeat the first phase for $-f$ and obtain the same information.

From the births and deaths of connected components we obtain the extended persistence diagram $\mathrm{Dgm}(f)$. A component that was born at a and died at b is represented by a point $(f(a), f(b))$ in $\mathrm{Dgm}(f)$. See Figure 2.1.2 for an example. This diagram is split into subdiagrams: the *ordinary subdiagram* $\mathrm{Ord}(f)$ containing points representing components with birth and death in the first phase, the *relative subdiagram* $\mathrm{Rel}(f)$ containing points representing components with birth and death in the second phase, and the *essential subdiagram* $\mathrm{Ess}(f)$. For topological reasons the connected component born at the global minimum in the first phase is considered to die at the global maximum of f in the second phase. It is represented by the sole point $(\min_x f(x), \max_x f(x))$ in $\mathrm{Ess}(f)$. The component born at the global maximum in the second phase is not represented in any diagram[b]. The three subdiagrams are disjoint and

$$\mathrm{Dgm}(f) = \mathrm{Ord}(f) \sqcup \mathrm{Rel}(f) \sqcup \mathrm{Ess}(f). \tag{1}$$

To give a brief example, Figure 2.1.3 shows the ordinary and essential subdiagram for the series in Figure 2.1.1b. The series has three obvious "valleys", which show up as three points away from the diagonal in the persistence diagram. Points closer to the diagonal correspond to small "wiggles", which

b This is due to the use of relative homology for phase two in the original definition of extended persistent homology, which is beyond the scope of this introduction. Briefly and informally, points in the relative subdiagram record the birth and death of relative cycles, and no such cycle is born when t passes the global maximum in phase two.

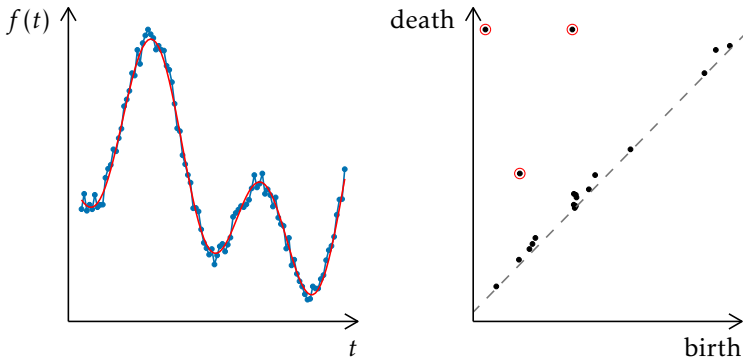

Figure 2.1.3: The time series from Figure 2.1.1b on the left and its ordinary and essential subdiagrams on the right. The dashed line indicates where birth equals death. Points close to the diagonal are considered noise; the three "valleys" of the series correspond to the circled points away from the diagonal.

we can consider noise. We quantify the importance of these features by their *persistence*, which is the difference between the death and birth of a component: a component that is born at a and dies at b has persistence $f(b) - f(a)$.

We obtain the *augmented persistence diagram* $\mathsf{D}\overrightarrow{\mathsf{gm}}(f)$ from the extended persistence diagram by adding arrows that encode the merging of components. We add an arrow from $(f(a), f(b))$ to $(f(p), f(q))$ if the component born at a dies at b to the component born at p with death at q. In Figure 2.1.2 there are arrows between points in $\mathsf{Ord}(f)$ and $\mathsf{Ess}(f)$; there are no arrows in the relative subdiagram, as they would point to the component born at t in the second phase, which is not represented in the diagram.

Biswas et al. [25] characterize persistent homology of maps on closed intervals in terms of windows[c]. They relate points in the persistence diagram to *windows* in f, which can be visualized as rectangles in $[1, m] \times \mathbb{R}$ framing the component corresponding to the pair of critical items. Windows can be nested and the nesting relation—more precisely, its transitive reduction—is represented in the arrows of the augmented persistence diagram. This characterization is fundamental to our data structure, as it reveals some useful symmetries that our update algorithms exploit.

Related Work

Before we describe our data structure we briefly discuss existing algorithms that compute the persistence diagram of maps on a closed interval. There is an $O(m)$ time algorithm by Glisse [26], which computes the persistence diagram by reading the list from left to right and only storing an item if it cannot be paired immediately; the resulting list follows a specific pattern which allows to pair the remaining items in a second pass over the list. However, this algorithm does not compute the nesting relation of windows, i.e., the arrows in the augmented persistence diagram. An algorithm that computes the full augmented persistence diagram, including the nesting relation, is known [27]: After sorting the critical items by order of increasing function value the augmented persistence diagram can be constructed using a disjoint-set data

c They study persistent homology of maps on closed intervals as a special case of maps on geometric trees and geometric networks.

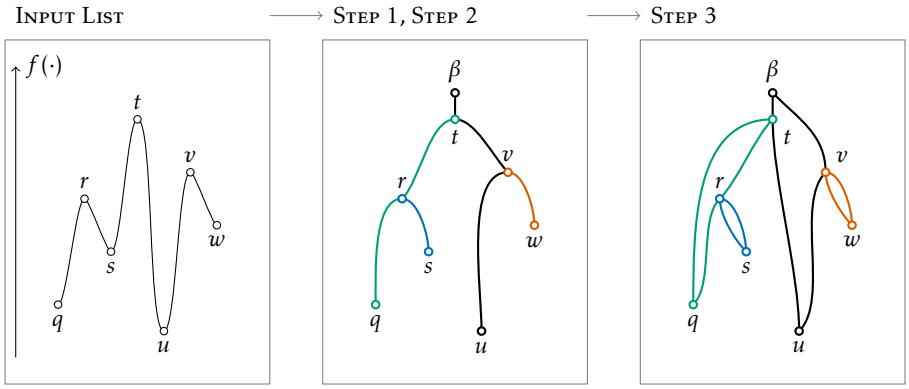

Figure 2.1.4: The process to obtain a banana tree (right panel) from a list (left panel). The critical items q to w are placed in a Cartesian tree (Step 1) with an additional *special root*, labeled β. This tree is decomposed into edge-disjoint paths (Step 2); see the colors in the middle panel. There are paths $P(q,t)$ (green), $P(s,r)$ (blue), $P(w,v)$ (red) and $P(u,\beta)$ (black) corresponding to points $(f(q),f(t))$, $(f(s),f(r))$, $(f(w),f(v))$ and $(f(u),f(t))$ in the persistence diagram, respectively. Paths are then split into trails (Step 3), resulting in the final banana tree, with bananas $B(q,t)$, $B(s,r)$, $B(w,v)$ and $B(u,\beta)$. The bananas $B(s,r)$ and $B(w,v)$ have two empty trails; node r is internal to the path $P(q,t)$ and hence part of a trail of $B(q,t)$.

structure. In total, this takes time $O(m + n\log n)$, where n is the number of critical items.

When the input is represented as a map on a simplicial complex, we can apply general algorithms for persistent homology. The classical algorithm computes a reduction of a matrix representation of the complex, which takes time $O(n^3)$ for a simplicial complex with n simplices[d] [5]. Existing algorithms to update persistent homology when the input changes are based on this approach. The first such algorithm is the one by Cohen-Steiner et al. [28], which handles updates where two simplices that are adjacent in the ordering by value switch position. Updating the matrix reduction takes time linear in the number of simplices. Dey et al. [29] show that on graphs this update time can be improved to $O(\log n)$.

2.1.1 Contribution

Given a list $c_1,\ldots,c_m$ we aim to maintain the augmented persistence diagram of the associated piecewise-linear function f as the list changes. We allow *local updates*, that is, changes to the value of a single item, and *topological updates*, which concatenate two lists or cut a list in two. We denote by n the number of critical items in the list. In Chapters 3 and 4 we introduce a data structure for this problem.

Central to our data structure are the *banana trees*. These are novel tree-like structures that explicitly represent the augmented persistence diagram. They are best described by a three-step process, illustrated in Figure 2.1.4:

Step 1: We begin with the unique Cartesian tree [30] constructed on the critical items of the list. This is a binary tree with items stored in order

d In a representation of a time series as a simplicial complex each simplex corresponds to a critical item, and n is equal to the number of critical items.

and with the item value increasing on any path from a leaf to the root. The leaves of the tree are the minima and the internal nodes are the maxima.

STEP 2: We add a parent to the root, which we call the *special root*, such that the tree has as many internal nodes as it has leaves. We decompose the tree into edge-disjoint paths between the leafs and the internal nodes. Each path $P(a,b)$ is chosen such that, when following it from the leaf a towards the special root, b is the first internal node with a leaf in its subtree that has smaller value than a; if no such internal node exists then b is the special root. In this decomposition the path $P(a,b)$ between leaf a and internal node b corresponds to a point $(f(a), f(b))$ in the persistence diagram, and for every maximum q on the path from a to b there is an arrow from the point $(f(\cdot), f(q))$ to the point $(f(a), f(b))$.

STEP 3: Finally, we split each path $P(a,b)$ into two trails: One trail contains nodes internal to $P(a,b)$ at which a path hangs off to the left; the other contains nodes internal to $P(a,b)$ at which a path hangs off to the right. Both trails share the minimum a and the maximum b. The trails are independently ordered by value, increasing from a to b. We call a pair of trails a *banana* spanned by a and b, which we write as $B(a,b)$.

The trails of a banana $B(a,b)$ are stored as two doubly-linked lists, one for each trail. There are further pointers linking the ends of bananas and representing the nesting of bananas. These pointers allow us to extract the augmented persistence diagram in time $O(n)$ by iterating over nodes.

We maintain two banana trees: the up-tree $\mathrm{Up}(f)$ and the down-tree $\mathrm{Dn}(f)$. Applying the process above to f yields the up-tree; applying it to $-f$ yields the down-tree. The up-tree represents the ordinary and essential subdiagram; the down-tree represents the relative subdiagram. There is an important symmetry between the up-tree and the down-tree: minima in the up-tree are maxima in the down-tree and vice versa. Our update algorithms exploit this property and thus we always maintain both banana trees, even if we are only interested in one of the subdiagrams. Additionally, we maintain two binary search trees: one for the minima and one for the maxima. In total, we require $O(m)$ space: $O(m)$ to represent the input and $O(n)$ for the banana trees and binary search trees.

We give a linear time algorithm to construct a banana tree from a list of items. It iterates over the list from left to right and processes each item twice: once to insert it into its banana and another time to correctly assign all pointers. This algorithm is the first to compute the augmented persistence diagram in linear time, i.e., the extended persistence diagram along with the nesting relation of windows. The algorithm by Glisse [26] also runs in linear time, but does not output the nesting relation.

We allow updates that change the value of an item—the local updates—and those that *cut* a list in two or *concatenate* two lists into one—the topological updates; see Figure 2.1.5. The former lead to structural changes within the banana trees; to process the latter we *split* and *glue* the banana trees.

We leave the description of the algorithms to Chapter 3 and only discuss the motivation for splitting the paths and maintaining both banana trees. For

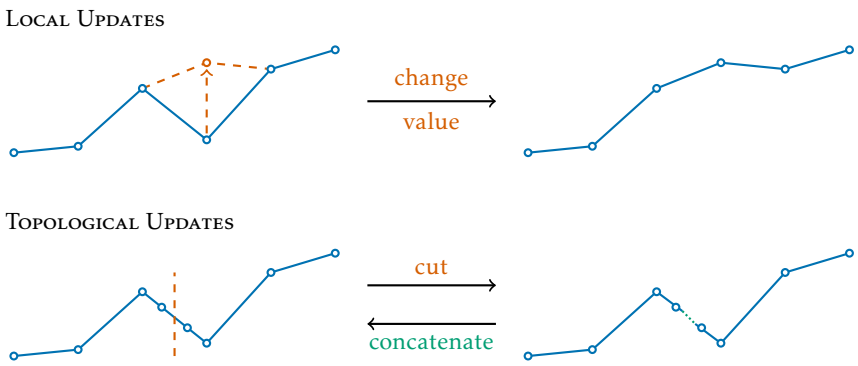

Figure 2.1.5: *Local updates* (top) change the value of a single item. *Topological updates* (bottom) cut the list in two or concatenate two lists into one.

a comprehensive overview of the algorithms see the technical overview in Section 3.3.

When changing the value of an item, i.e., when performing a local update, the position of this item in the order of items by function value changes. Whenever two consecutive items swap their position in this order, there may be a change in the augmented persistence diagram. However, most of these swaps do not lead to any change, and it is imperative that we only process those that do, in order to achieve a fast update time.

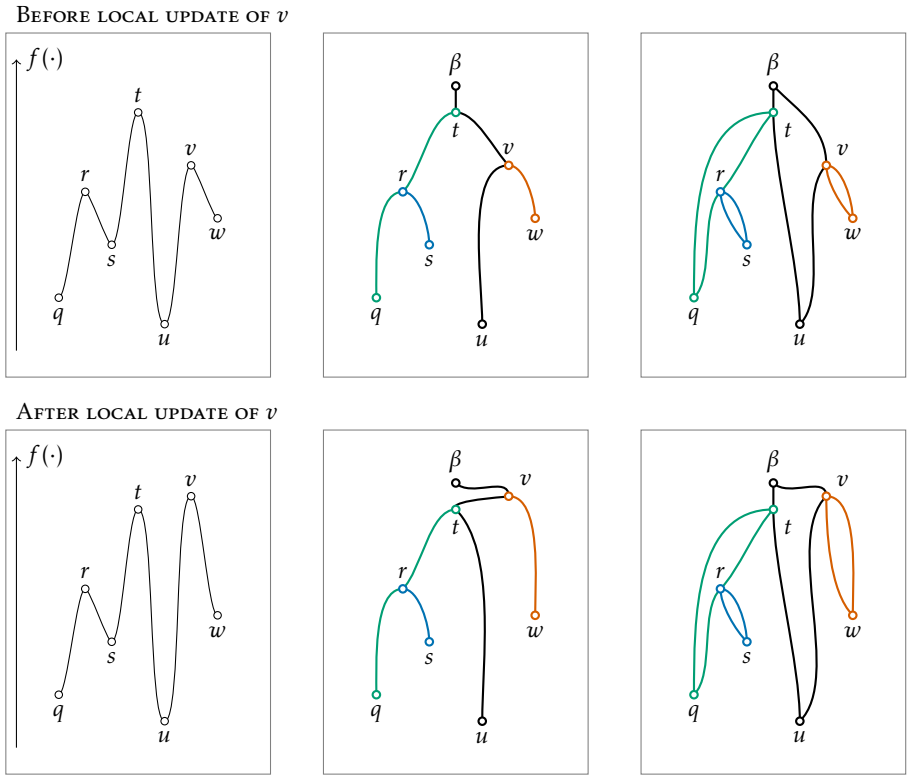

Figure 2.1.6: A local update increasing the value of item v, which leads to an interchange of maxima with t. The state before the update is shown on top; that after the update is shown below. Maintaining a correct path-decomposed binary tree requires a rotation; the structure of the banana tree is unchanged.

Swaps between two maxima or two minima are particularly interesting. These are the *interchanges of maxima* and *interchanges of minima* and the reason for splitting paths into trails. An interchange of maxima only has an effect on the augmented persistence diagram if the two maxima lie on the same path of the path-decomposed binary tree. However, this condition is not sufficient: consider the example in Figure 2.1.6 where maxima v and t interchange; even though v and t lie on the same path, the augmented persistence diagram is unchanged, i.e., the pairing of items and the arrows remain identical after the update. Maintaining the path-decomposed binary tree requires a rotation, as otherwise the value of nodes along the path $P(u, \beta)$ is no longer strictly increasing towards the root. Splitting the path into two trails avoids this issue: the banana tree—right panel of Figure 2.1.6—is unaffected by the interchange. In fact, an interchange of maxima only results in a change in the augmented persistence diagram if the two maxima lie on the same trail. As a consequence, the splitting of paths into trails makes banana trees oblivious to any interchanges that do not cause a change in the augmented persistence diagram, and we do not perform any work for those.

Interchanges of minima are directly connected to interchanges of maxima. When increasing the value of a minimum a, detecting the minimum p with which it interchanges is non-trivial. If a together with a maximum b spans the banana $B(a, b)$, then p must be the lower end of a banana on a trail of $B(a, b)$. There are many such minima and searching for the correct one is too expensive. Instead, we use the fact that whenever two minima interchange in f, two maxima interchange in $-f$. We show that an interchange of minima a and p only leads to changes in the augmented persistence diagram if the interchange of maxima a and p in $-f$ does. Hence, when updating a minimum a we perform the corresponding interchanges of maxima in the down-tree and use those to identify the minimum with which to interchange a. This allows us to again only pay for interchanges of minima that actually lead to changes in the augmented persistence diagram, and ignore those that do not.

A local update is decomposed into a sequence of local operations, which are the cancellations, anti-cancellations and slides, in addition to the interchanges. Processing an interchange takes time $O(k)$, where k is the number of changes in the augmented persistence diagram. The remaining operations appear only a constant number of times in the sequence of operations and take time $O(\log n + k)$, with the exception of anti-cancellations. These insert a new banana into the banana tree, which requires a search that takes time $O(k')$, where k' is the number of changes in the transitive closure of the augmented persistence diagram. This is bounded by the height of the path-decomposed binary tree, which is $O(n)$ in the worst-case. The topological updates, cutting and concatenating, involve *splitting* and *gluing* banana trees, which can also be done in time $O(\log n + k)$.

Our main goal is to obtain a data structure that is performant in practice. In Chapter 5 we provide an experimental evaluation of banana trees. We study how the running time is linked to their structural properties, in order to understand on what kind of data the banana trees perform well. We focus on two structural properties:

1. The nesting depth, measuring how deep the nesting hierarchy of bananas is. For a banana $B(a, b)$ it is defined as the smallest number of bananas

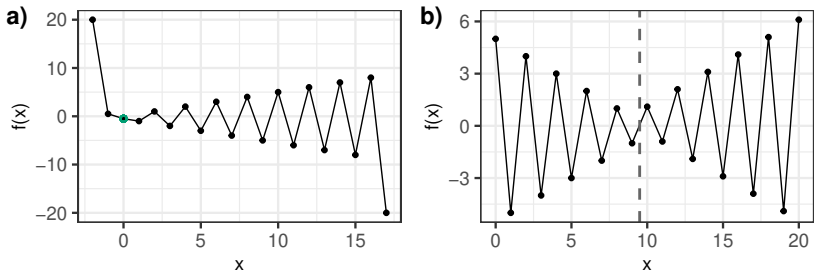

Figure 2.1.7: **a)** Worst-case instance for local updates with $n = 20$ items. Increasing the value of the item circled in blue results in an anti-cancellation that takes linear time, followed by up to $n/2$ interchanges. **b)** Worst-case instance for topological operations with $n = 21$ items. Cutting at the dashed line affects $\Theta(n)$ bananas. Concatenating the resulting lists again affects $\Theta(n)$ bananas.

that need to be traversed from b in order to reach the special root. The nesting depth bounds the time for the search in anti-cancellations and the number of processed bananas in the topological operations.

2. The trail length, i.e., the number of nodes internal to trails. This bounds the time for processing an interchange of minima and operations that appear in topological updates.

To our knowledge there is no other implementation of a dedicated dynamic algorithm for the persistent homology of time series. Hence, we compare to the static linear-time algorithm by Glisse [26] as implemented in `Gudhi` [31].

We perform experiments on synthetic time series: random walks, quasi-periodic series and worst-case instances. We define a random walk of length n with parameters μ and σ as

$$r(0) = 0; \quad r(t) = r(t-1) + \gamma_t \text{ for } 1 \leq t \leq n, \tag{2}$$

where $\gamma_t \sim \mathcal{N}(\mu, \sigma)$ are independent and normally distributed with mean μ and standard deviation σ. The parameter μ controls the *bias*: a random walk with $\mu = 0$ is *unbiased*, a random walk with $\mu \neq 0$ is *biased*. For quasi-periodic time series we set $\gamma_t \sim \mathcal{N}(\mu_t, \sigma)$ with $\mu_t = \sin(2\pi\omega t)$. By adjusting the standard deviation σ the series becomes more or less periodic: as σ approaches 0 the noise becomes negligible and the signal resembles a sine wave; for $\sigma \gg 1$ the series resembles an unbiased random walk. Finally, the worst-case instances are tailored to require update times linear in the input size. We construct such instances for local updates and for topological updates, as illustrated in Figure 2.1.7.

Our experiments demonstrate that banana trees can efficiently update the persistence diagram when performing local or topological changes. On random walks and quasi-periodic inputs they are faster than the static reference algorithm by several orders of magnitude for inputs of length greater than 1000. We connect these results on performance to the structural properties of banana trees. On unbiased random walks we observe short trails—around 0.5 on average—and small nesting depth logarithmic in the input length; biased random walks yield banana trees with smaller nesting depth, but the longest trail contains around 95% of the internal nodes for $|\mu| = 1$. Our algorithms

perform best on unbiased random walks and the performance degrades as the bias increases. We observe a similar trend on quasi-periodic time series. As we decrease the parameter σ the nesting depth decreases and the trail length increases. Local and topological maintenance also become slower with decreasing σ. With small σ the quasi-periodic series are locally similar to biased random walks; with large σ they resemble unbiased random walks.

On the worst-case instances maintaining banana trees is slower than the static algorithm, with slowdowns between 20 and 50. This is expected, since both algorithms take linear time, but banana trees perform more bookkeeping. However, mixing the worst-case instances with random walks shows that these scenarios are unstable: a small perturbation leads to drastic improvements in the running time of the banana tree maintenance algorithms.

Finally, we analyze structural properties of banana trees on real-world time series obtained from electrocardiography, audio recordings and human physical activity recordings. We observe that these banana trees exhibit structural properties that are similar to those on random walks and quasi-periodic series. We thus expect banana trees to perform well in practical use cases on real-world data.

2.2 Private Analysis of Dynamic Graphs

Motivation

Many data sets, such as medical records or frequently visited locations, contain sensitive information. Revealing this information puts the individuals contributing their data at risk. Thus, when publishing any information about the data set or parts of the data set, we have to take care to not leak this sensitive data.

The traditional approach to anonymize the data is by removing any "personally identifying information", such as names, addresses, ZIP-codes. However, this does not stop an adversary from re-identifying entries in the database, especially if auxiliary data is available. There are many examples where anonymization fails: Sweeney [32] describes how anonymized medical records can be re-identified using voter registration data; in a blog post Tockar [33] demonstrates how an anonymized taxi data set released by the New York City Taxi and Limousine Commission can be used to identify taxi trips of celebrities or frequent customers of a nightclub; Narayanan and Shmatikov [34] identify Netflix users in a data set of anonymized movie ratings by cross-referencing it with the Internet Movie Database (IMDb); Calandrino et al. [35] present algorithms to infer a consumer's behavior from the output of recommender systems, exploiting their dynamic nature. For further examples see the survey by Narayanan et al. [36].

While there are many attempts to mitigate such attacks, for example k-anonymity [37], density-based perturbation [38, 39], l-diversity [40] and t-closeness [41], all of these are prone to privacy leaks, especially in the presence of auxiliary information. Since it is impossible to control what auxiliary information is available to an adversary, an approach to privacy that is independent of the adversary's knowledge and computational power is needed [36]. *Differential privacy* [42, 43] was developed to satisfy this requirement and offers protection against the attacks described above: Tockar [33] explains how to

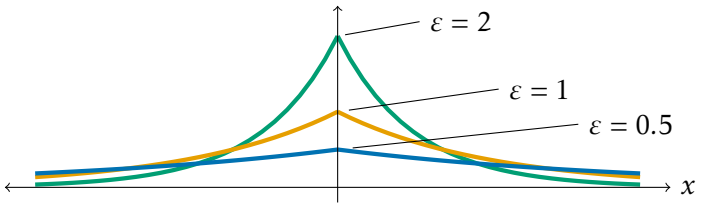

Figure 2.2.1: The probability density functions for Lap(1/ε) with ε = 2 (green), ε = 1 (yellow) and ε = 0.5 (blue). Decreasing the privacy loss ε in the Laplace mechanism increases the variance of the Laplace noise, resulting in higher error.

use differential privacy to protect the taxi customers' privacy, and Calandrino et al. [35] list differential privacy among the possible mitigation strategies. Differential privacy has been applied in a variety of areas, such as commuting data from the US Census [44], end-user statistics in the Chrome web browser [45], ad analytics on LinkedIn [46] and general SQL queries [47].

Basic Concepts of Differential Privacy

We now give an overview of differential privacy, focusing on the concepts relevant to our results. The possible applications of differential privacy are diverse and thus the space of models and methods is quite large. For broader overviews we refer to the book by Dwork and Roth [48] and the tutorial by Vadhan [49].

We are in the setting of *centralized* differential privacy and take on the role of a trusted curator, who has sole access to the database and aims to publish some analysis without revealing sensitive information. Assume that we want to privately compute a function $f : \mathcal{X} \to \mathcal{R}$, where $\mathcal{X}$ is the set of databases and $\mathcal{R}$ is the space of possible outputs, e.g., the set of real numbers. Differential privacy requires the output of an algorithm to be similarly distributed when supplied with *adjacent* inputs from $\mathcal{X}$. Usually, when speaking of databases, inputs are considered adjacent when they differ in the presence of a single entry. The exact definition of adjacency depends on the type of input and on the attributes to be protected. The similarity of output distributions is formalized in the following definition, where Range($\mathcal{A}$) denotes the set of all possible outputs of $\mathcal{A}$.

Definition 2.2.1 (Differential Privacy [42])**.** A randomized algorithm $\mathcal{A}$ taking as input elements of $\mathcal{X}$ is ε-differentially private if for any $R \subseteq \text{Range}(\mathcal{A})$ and any adjacent $x, x' \in \mathcal{X}$

$$\Pr[\mathcal{A}(x) \in R] \leq e^{\varepsilon} \Pr[\mathcal{A}(x') \in R], \tag{3}$$

where $\varepsilon > 0$ is the *privacy loss*.

With lower privacy loss the distributions become more similar, leading to increased privacy protection. Vice versa, higher privacy loss implies decreased privacy protection. Consider the case $\varepsilon = 0$, where the distributions are identical and no information can be gained about the database, and the case $\varepsilon = \infty$, where a non-private exact algorithm satisfies (3).

A basic algorithm to achieve differential privacy is the *Laplace mechanism* [42]. Assume that we want to release the value of a function $f : \mathcal{X} \to \mathbb{R}$ where $|f(x) - f(x')| \leq S$ for any adjacent $x, x' \in \mathcal{X}$. The Laplace mechanism outputs

$$\mathcal{M}(x) := f(x) + \gamma, \quad \gamma \sim \mathrm{Lap}(S/\varepsilon), \tag{4}$$

where $\mathrm{Lap}(s)$ is the Laplace distribution with mean 0 and scale s, whose density at r is $\frac{1}{2s}e^{-|r|/s}$. The constant S is a bound on the *global sensitivity* of the function f, which is defined as $\max_{\text{adj. } x,x' \in \mathcal{X}} |f(x) - f(x')|$. Accuracy of the output is quantified with the *additive error* $|\mathcal{M}(x) - f(x)|$, which for the Laplace mechanism satisfies

$$\Pr[|\mathcal{M}(x) - f(x)| \leq (S/\varepsilon) \cdot \ln(1/\delta)] \geq 1 - \delta. \tag{5}$$

It quantifies the *utility* of the algorithm: the lower the additive error, the more accurate and thus useful the output. Equations (4) and (5) illustrate how ε controls the trade-off between utility and privacy (see Figure 2.2.1): decreasing ε improves privacy protection, as discussed above, but increases the variance of the Laplace distribution in (4), which increases the error and lowers the utility. The parameter δ in (5) is called the *failure probability*.

For functions with high global sensitivity the Laplace mechanism adds a lot of noise, which may render the algorithm useless, in particular if the error is of similar magnitude as the true function value. Adding noise that is scaled to the global sensitivity is not always necessary: approaches such as smooth sensitivity [50] and the propose-test-release framework [51] reduce the scale of the noise dependent on the specific input and its neighborhood. These are not directly relevant to our work, but we mention them for completeness.

We now turn to differential privacy for functions on graphs. A *graph* $G = (V, E)$ consists of a set of *nodes* or *vertices* V along with a set of undirected *edges* $E \subseteq V \times V$. We also consider *edge-weighted graphs*, in which case $G = (V, E, w)$ with $w : E \to \mathbb{N}$ mapping each edge to its weight. An edge $e = \{u, v\}$ is said to be *incident* to u and v. Two nodes u and v are *adjacent* if $\{u, v\} \in E$.

There are two common definitions of adjacency on graphs: two graphs are *edge-adjacent* if they have the same set of vertices but differ in exactly one edge; similarly, two graphs are *node-adjacent* if they differ in exactly one vertex and its incident edges. These notions of adjacency give rise to *edge-differential privacy* and *node-differential privacy*. In edge-differential privacy the set of vertices can be considered public knowledge[e] and the edges are the sensitive information. In node-differential privacy both the nodes and the edges are sensitive information. Which one is preferred in a specific setting depends on the data that is represented by the graph. If the data associated with the nodes is public knowledge, such as the users in a social network, then edge-differential privacy is sufficient. Sometimes however, revealing whether a user is part of the graph reveals sensitive information, e.g., in infection networks. Such cases call for node-differential privacy.

Differential Privacy in the Dynamic Setting

We consider a setting where the input changes dynamically over time and the output is updated regularly. This is the *continual observation model* introduced

e Because edge-adjacent graphs share the same vertex set, an algorithm that outputs the set of vertices is 0-edge-differentially private.

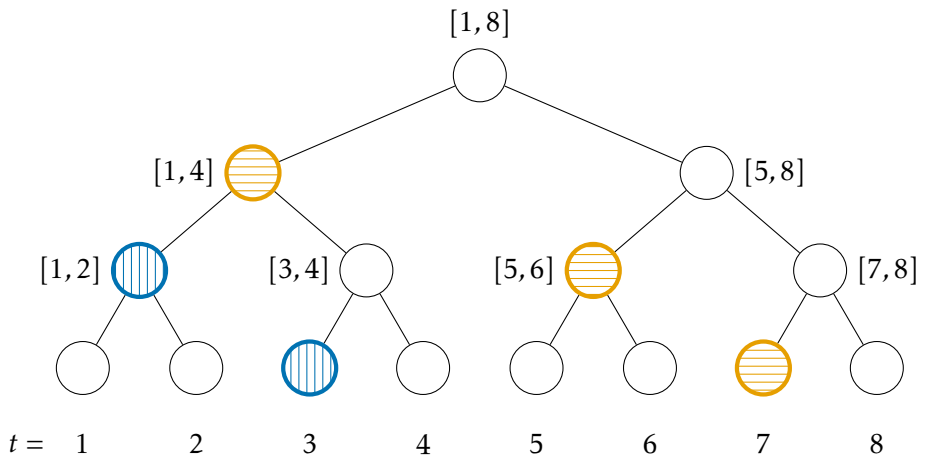

Figure 2.2.2: The binary tree used to compute the sums in the binary mechanism. Each node corresponds to a noisy sum over an interval of the stream, as annotated. The differentially private sum of the stream up to time step 3 is computed from the sums at the blue vertically striped nodes; the sum up to time step 7 is computed from the sums at the yellow horizontally striped nodes.

by Dwork et al. [52]. It aims to protect privacy in cases where statistics are released regularly. This model is, for example, applicable in protecting recommender systems against the attacks described by Calandrino et al. [35]. Song el al. [53] motivate their work on computing graph statistics under continual observation with the example of HIV transmission networks collected over several years.

The simplest problem in this model is that of maintaining a binary counter. The input is a stream of T ones and zeroes, and the objective is to output the number of ones after each time step. Two streams are considered *event-level adjacent* if they differ in a single bit[f]. A differentially private counter can be maintained by constructing a binary tree, where leaves represent the items of the stream and internal nodes represent the intervals spanned by their children. Each node stores a sum of the corresponding interval with added Laplace noise. The output can then be computed by summing up to $\log t$ of these noisy sums, where t is the current length of the stream, as illustrated in Figure 2.2.2. This algorithm is known as the binary mechanism[54, 55]. It provides event-level ε-differential privacy and at each time step has error $O(\varepsilon^{-1} \cdot \log^{3/2} T \cdot \log \delta^{-1})$ with probability at least $1-\delta$. Our results in Section 6.3 build on a more general version of this algorithm, where items in the stream are bounded integers and streams may differ in multiple time steps.

Dwork et al. [54] show that this error is tight up to a factor $\sqrt{\log T}$. They present a lower bound on the additive error of differentially private algorithms for counting under continual observation, which states that any such algorithm for streams of length T must have error $\Omega(\log T)$, even if $\varepsilon = 1$.

Our work also builds on the sparse vector technique (SVT) [56, 57]. SVT answers threshold queries on a sequence of function values $f_1, \ldots, f_T$, i.e., whether f_i exceeds some threshold t_i, where only $c \ll T$ queries are expected to exceed the threshold. An ε-differentially private algorithm for SVT is

f Dwork et al. [54] define continual observation with the stronger notion of *user-level privacy* in mind. For simplicity we focus on event-level privacy here, and leave discussion of user-level privacy to Section 2.2.1.

described in [48] (Algorithm 2 in Section 3.6). It answers at most c queries positively, such that with probability at least $1-\beta$ for all positively answered queries $f_i \geq t_i - \alpha$ and for all negatively answered queries before the c-th positive answer $f_i \leq t_i + \alpha$, where $\alpha = O(\varepsilon^{-1} c(\log T + \log(c/\beta)))$. Observe that the negatively answered queries contribute only a logarithmic factor to the error.

2.2.1 Contribution

In Chapter 6 we present differentially private algorithms for graphs under continual observation, i.e., for dynamics graphs, along with lower bounds on the error. Prior to our work only Song et al. [53] studied differential privacy in this setting. They gave algorithms for problems in partially dynamic graphs with maximum bounded degree. Our contribution is three-fold:

1. We generalize the model introduced by Song et al. [53] and give a formal definition of adjacency;

2. We give two differentially private algorithms for graphs under continual observation;

3. We give lower bounds for the additive error of differentially private algorithms on event-level and user-level.

We consider undirected graphs $G = (V, E)$, as defined above, which may be edge-weighted. In that case, $G = (V, E, w)$, where $w : E \to \mathbb{N}$ are the edge weights. The input to our algorithms are *graph sequences* $\mathcal{G} = (G_1, G_2, \ldots, G_T)$, where each graph $G_t = (V_t, E_t)$ is derived from $G_{t-1} = (V_{t-1}, E_{t-1})$ by edge and vertex updates. These updates are given as sets of deletions and insertions, which are subtracted and added, respectively, to V_{t-1} and E_{t-1}. Whenever a node is deleted its incident edges are also deleted. The initial graph $G_0 = (V_0, E_0)$ is given and may or may not be empty. The number of graphs in the graph sequence is its length. Here, $\mathcal{G}$ has length T. A graph sequence is said to be *fully dynamic* if there are both insertions and deletions; otherwise, it is *partially dynamic*. An *incremental* graph sequence has no deletions, and a *decremental* one has no insertions.

We define adjacency between two graph sequences by the differences in their sets of updates. Informally, two graph sequences are *edge-adjacent* if they differ in exactly one update to exactly one edge e^*. Similarly, two graph sequences are *node-adjacent* if they differ in exactly one update to exactly one node v^* and its incident edges. Figure 2.2.3 illustrates two edge-adjacent graph sequences. Taking $\mathcal{X}$ in Definition 2.2.1 to be the set of all graph sequences and considering these definitions of adjacency yields the definition for *edge-* respectively *node-differential privacy* on *event-level*.

Definition 2.2.2 (Differential privacy on event-level)**.** An algorithm $\mathcal{A}$ taking as input a graph sequence is ε-edge-differentially private (ε-node-differentially private) if for any two edge-adjacent (node-adjacent) graph sequences $\mathcal{G}, \mathcal{G}'$ and any $R \subseteq \text{Range}(\mathcal{A})$ we have

$$\Pr[\mathcal{A}(\mathcal{G}) \in R] \leq e^{\varepsilon} \cdot \Pr[\mathcal{A}(\mathcal{G}') \in R].$$

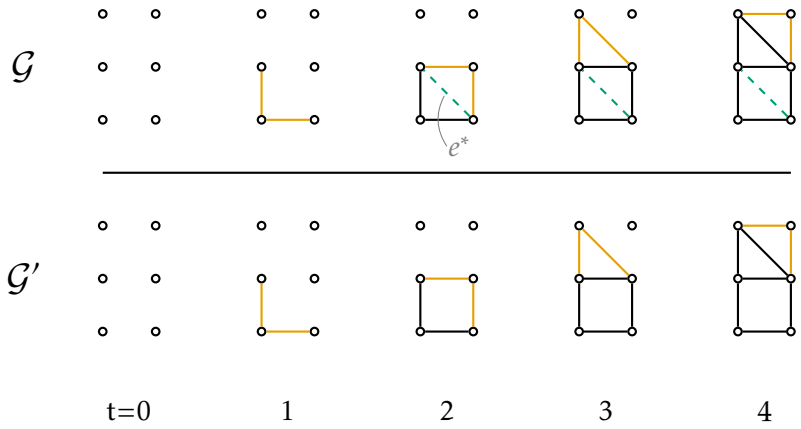

Figure 2.2.3: Two incremental edge-adjacent graph sequences $\mathcal{G}$ and $\mathcal{G}'$. Apart from the insertion of the dashed green edge e^* from time $t = 1$ to $t = 2$ into $\mathcal{G}$ the updates to both sequences are the same. Newly inserted edges are highlighted in orange.

We also consider a stronger version of differential privacy where graph sequences are allowed to differ in multiple updates to the same edge e^*. This gives rise to ε-edge-differential privacy on *user-level*. The definition for user-level node-differential privacy is analogous.

We present two event-level differentially private algorithms for several functions on partially dynamic graph sequences. These algorithms take as input a graph sequence $\mathcal{G} = (G_1, G_2, \ldots, G_T)$ and compute the sequence $f(\mathcal{G}) = (f(G_1), f(G_2), \ldots, f(G_T))$. We also give lower bounds on the additive error of edge-differentially private algorithms on event-level in the partially dynamic setting, and on the additive error of edge-differentially private algorithms on user-level in the fully dynamic setting.

Our first algorithm is based on the *difference sequence* approach introduced by Song et al. [53]. Let $\mathcal{G}$ and $\mathcal{G}'$ be two adjacent graph sequences of length T and write f for the function that we compute privately. The sequences $f(\mathcal{G})$ and $f(\mathcal{G}')$ may differ in arbitrarily many time steps, which implies that the sequence f has *global sensitivity* linear in the length T, where global sensitivity of f is defined as the maximum value of $\|f(\mathcal{G}) - f(\mathcal{G}')\|_1$ over all adjacent graph sequences $\mathcal{G}$,$\mathcal{G}'$. If we use the Laplace mechanism naïvely, the output then has error linear in T. Song et al. [53] observed that the *difference sequence of* f, i.e., the sequence $\Delta f(\mathcal{G}) = (f(G_1) - f(G_0), f(G_2) - f(G_1), \ldots)$, has global sensitivity independent of T for a wide range of functions. Adding Laplace noise to each element in the difference sequence and computing the sum up to time step t yields a differentially private approximation to the function value for time step t. The error of this approach scales with the square root of the length of the sequence. We compute the differentially private sum of the difference sequence via the counting algorithms of Chan et al. [55], which reduces the error by a factor $\log^{3/2} T/\sqrt{T}$. In order to be able to use the counting algorithm for binary streams we show (1) that it can be generalized to streams of bounded integers and (2) that it can be applied to difference sequences.

Theorem 2.2.3. *Let $\varepsilon, \delta > 0$. Let f be a graph function whose difference sequence has global sensitivity S. There exists an event-level ε-differentially private algo-*

rithm to compute f that at each time step, with probability at least $1-\delta$, gives an answer with additive error $O(S \cdot \varepsilon^{-1} \cdot \log^{3/2} T \log(1/\delta))$.

This algorithm is applicable whenever the difference sequence has global sensitivity independent of the length of the input. This is the case in several problems in the partially dynamic setting, such as subgraph counts, the degree histogram, the number of high-degree nodes in graphs with bounded maximum degree or the value of a minimum spanning tree in graphs with bounded edge weight. Other problems, such as the value of a minimum cut or maximum matching, require a different approach, as in their case the difference sequence has global sensitivity linear in the length T of the input sequence. These are examples of *non-local* problems, whose value cannot be derived from the frequency histogram of constant-size subgraphs. For these problems, we introduce a second algorithm, which can be applied to any function on partially dynamic graphs, independent of the difference sequence, as long as its values are monotone with respect to updates to the graph.

Consider a function f with range $[1, r]$ that is monotonically increasing on incremental graph sequences[g]. The algorithm employs the sparse vector technique to privately detect when the function value increases by a factor $(1+\beta)$, where $\beta > 0$ is a parameter. It then outputs the last multiple of $(1+\beta)$ that the function value crossed. That is, if the sparse vector technique answers k queries positively, the algorithm outputs $(1+\beta)^k$. As there are at most $\log_{1+\beta} r$ such increases, the sparse vector technique answers at most $O(\log r)$ queries positively. For the formal statement of this result we need to distinguish between the global sensitivity of the sequence of function values as used above and the *static global sensitivity* $\mathrm{GS}_{\text{static}}$ with respect to graphs, i.e., the maximum of $|f(G) - f(G')|$ over any two adjacent graphs G, G'.

Theorem 2.2.4. *Let f be a function that is monotone on any input with range $[1, r]$ and static global sensitivity $\rho := \mathrm{GS}_{\text{static}}(f)$. Let $\varepsilon > 0$, $\delta, \beta \in (0, 1)$. There exists an ε-differentially private algorithm for computing f with multiplicative error $(1+\beta)$, additive error $O(\varepsilon^{-1} \log_{1+\beta}(r) \rho \ln(T/\delta))$, and failure probability δ.*

While the first algorithm could be applied to any kind of graph sequence, it is limited to partially dynamic sequences in practice, since for most functions the global sensitivity of the difference sequence depends on the input length T in the fully dynamic setting. The second algorithm is also restricted to partially dynamic inputs in practice, as graph functions are usually not monotone on fully dynamic sequences and the number of value increases cannot be bounded in that case. The fully dynamic setting is left as an open problem.

We now turn to lower bounds. Dwork et al. [54] gave a logarithmic lower bound on the additive error of algorithms for binary counting under continual observation. By reducing graph functions on incremental sequences to binary counting we show that differentially private algorithms for these functions require additive error logarithmic in the length T of the input. For minimum spanning tree, minimum cut and maximum matching this results in lower bounds of $\Omega(W \log T)$, where W is the maximum edge weight; for subgraph counting problems, the number of high-degree nodes and the degree histogram we obtain lower bounds of $\Omega(\log T)$.

g The algorithm can also be applied to monotonically decreasing functions or decremental graph sequences.

These lower bounds hold on event-level and both for edge-adjacency and node-adjacency. We also provide lower bounds for edge-differentially private algorithms on user-level in the fully dynamic setting. Informally, the additive error has to be greater than the maximum value of the function to guarantee differential privacy, which makes differential privacy on user-level in the fully dynamic setting impossible to achieve in practice with acceptable utility. For minimum spanning tree, minimum cut and maximum matching the resulting lower bound is $\Omega(Wn)$, where W is again the maximum edge weight.

2.3 Dynamically Scheduling Reconfigurable Networks

Motivation

Data intensive applications, such as machine learning tasks on large data sets, place high communication demands on networks in datacenters [58, 59]. There are usually few large communication flows which account for most of the transmitted data [59, 60]. Serving these flows via a static network, e.g., an electrical packet-switched network, is problematic: either the network is tailored to the worst-case, in which case bandwidth goes unused, or the network is tailored to the average case, in which case there will be congestion and thus decreased application performance. Demand-aware reconfigurable networks allow to avoid both extremes, by routing the large flows via a network that provides direct connections between the communicating endpoints.

Many reconfigurable network architectures are based on *optical circuit switches* [61–69], which route optical signals without requiring any intermediate conversion to an electrical signal. These switches establish direct one-to-one connections between communication endpoints, e.g., the racks in a datacenter, and allow for reconfiguration within microseconds [58]. The optical network is usually combined with a static electrical packet-switched network, over which the remaining demand can be transmitted [61, 64–69]. Besides optical circuit switches reconfigurable networks may also be built on top of free-space optics [70, 71] and wireless [72, 73].

Determining an optimal network configuration based on the current demand is a bottleneck. A key observation is that each optical circuit switch augments the network with a matching. This motivated Hanauer et al. [74] to study the problem of computing k heavy disjoint matchings in a graph representing the network, where each of the matchings then corresponds to one of k optical circuit switches. They describe algorithms for this problem in a static setting, which require to recompute the configuration from scratch with every change in the communication demand. Since datacenter traffic often exhibits temporal locality and only a small number of changes to the configuration may be required, recomputing from scratch may be inefficient. Hence, we study the problem in a dynamic setting, where existing solutions are updated rather than recomputed.

Reconfigurable Networks via Disjoint Matchings

We assume an architecture as illustrated in Figure 2.3.1: optical circuit switches form connections between racks, each of which houses multiple servers; racks are also connected via a static network, which routes any de-

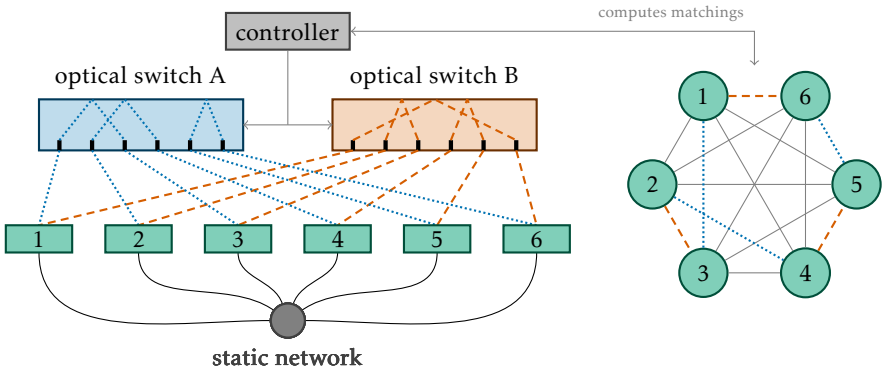

Figure 2.3.1: Left: a datacenter consisting of six racks, labeled 1–6, connected by a static network and a reconfigurable optical network. A centralized controller configures the optical switches A and B, which establish direct connections between pairs of racks. Right: the network is modeled as a graph, where nodes correspond to racks. The connections established by each switch correspond to a matching in this graph, shown as dotted blue lines for switch A and dashed red lines for switch B. Demand along gray edges is not served by the optical network.

mand that is not served by the optical network; a centralized controller observes the communication demand and configures the optical switches accordingly.

The network is modeled as an undirected weighted graph $G = (V, E, w)$ with nodes V, edges $E \subseteq V \times V$ and non-negative edge weights $w : E \to \mathbb{N}_0$. The nodes correspond to racks in the datacenter and edges describe their communication demand: an edge $\{a, b\}$ implies that the communication demand between the racks associated with a and b is $w(\{a, b\})$. If $w(\{a, b\}) = 0$ the racks do not communicate and we consider the edge to be absent. We call an edge e *incident* to a node u if $u \in e$. Two edges e and e' are *adjacent* if they are incident to the same node, or equivalently, satisfy $e \cap e' \neq \emptyset$. The subgraph *induced* by a set of edges S is the graph $(\bigcup_{e \in S} e, S, w)$.

An optical switch can connect arbitrary pairs of racks, limited to one connection per rack. That is, the connections established by a switch form a *matching* $M \subseteq E$, in which no two edges share an endpoint, i.e., for all $u, v \in M$ we have $u \cap v = \emptyset$. There are k optical switches and for each we want to find a matching in G to determine which racks they should connect. This leads us to compute *k (edge-)disjoint matchings* $\mathcal{M} = \{M_1, \ldots, M_k\}$, a set of k matchings $M_1, \ldots, M_k$, such that $M_i \cap M_j = \emptyset$ for all $i, j \in [k]$, $i \neq j$. To maximize the demand served by the optical switches we aim to maximize the weight of these matchings, which is defined as

$$w(\mathcal{M}) = \sum_{i=1}^{k} w(M_i) = \sum_{i=1}^{k} \sum_{e \in M_i} w(e).$$

We are only concerned with maximizing the demand routed over the optical network and do not consider the remaining demand further. We assume that it can be routed via the static network.

This problem is equivalent to computing a *partial (edge) k-coloring* $\mathcal{C} : E \to [k] \cup \{\perp\}$, which either assigns a color or the symbol $\perp$ to each edge. If $\mathcal{C}(e) = \perp$ the edge e is *uncolored*, otherwise, if $\mathcal{C}(e) \in [k]$, the edge is *colored*. Two adjacent edges do not share the same color, but they may both be uncolored: that is,

for two adjacent edges e, e' either $\mathcal{C}(e) \neq \mathcal{C}(e')$ or $\mathcal{C}(e) = \mathcal{C}(e') = \perp$. We say that a color c is *free* at an edge $e = \{u, v\}$ if there is no edge e' incident to u or v such that $\mathcal{C}(e') = c$. Whenever a color c is free at e, setting $\mathcal{C}(e) = c$ yields a valid coloring.

Since two adjacent edges do not have the same color, the set $\{e \in E \mid \mathcal{C}(e) = c\}$ is a matching for any $c \in [k]$. Similarly, given k disjoint matchings $\mathcal{M} = \{M_1, \ldots, M_k\}$ we can obtain a partial k-coloring $\mathcal{C}_\mathcal{M}$ where $\mathcal{C}_\mathcal{M}(e) = i$ if $e \in M_i$ and $\mathcal{C}_\mathcal{M}(e) = \perp$ if e is not in any matching. Thus, a partial k-coloring is equivalent to k disjoint matchings. In analogy to the weight of disjoint matchings we define the weight of a coloring to be the sum of the weights of the colored edges:

$$w(\mathcal{C}) = \sum_{e \in E \mid \mathcal{C}(e) \neq \perp} w(e).$$

Related Work

The k disjoint matchings problem is equivalent to a weighted version of the *maximum k edge-colorable subgraph problem*, which has previously been studied in the static setting. Feige et al. [75] showed that for every $k \geq 2$ there exists an $\varepsilon > 0$ such that the problem is NP-hard[h] to approximate within a factor $1 - \varepsilon$. They give approximation algorithms for general k and for $k = 2$. Among these is a greedy algorithm, that in each of k iterations computes a maximum matching, assigns a color to the matched edges and removes them from the graph. This algorithm achieves an approximation ratio of $1 - (1 - \frac{1}{k})^k$. Using greedy matchings instead of maximum matchings yields an approximation ratio of $\frac{1}{2}$, which is tight [74]. The current best approximation algorithms for small k are due to Chen et al. [76] ($k = 2$), and Kamiński and Kowalik [77] ($3 \leq k \leq 7$). For general $k > 7$ the best known approximation ratio is $\frac{k}{k+1}$ due to Feige et al. [75]. The problem has also been studied in the online setting [78] and in the context of parameterized complexity [79, 80].

2.3.1 Contribution

We study algorithms that maintain a solution to the k disjoint matchings problem under updates to the edge weights. An update $\mathcal{U}(e, \delta)$ changes the weight of the edge e by an amount $\delta \neq 0$, i.e., it assigns $w(e) \leftarrow w(e) + \delta$. Edges may also be deleted or inserted, which we treat as changes of the weight to and from zero, respectively. The updates arrive in a sequence that is partitioned into batches, where every edge is updated at most once in a batch. Whenever a batch ends the algorithms are expected to output the current solution weight and the change in the solution since the last batch. That is, with solutions $\mathcal{C}'$ and $\mathcal{C}$ before and after the batch, respectively, the algorithm outputs $w(\mathcal{C})$ and $\{e \in E \mid \mathcal{C}(e) \neq \mathcal{C}'(e)\}$. The output of the change in the coloring is required to run in time linear in its cardinality, which is called the *recourse*.

In Chapter 7 we present seven heuristic algorithms for the problem, with the focus on practical performance. In our model batches arrive as a sequence

h For $k \geq 3$ this follows from the NP-hardness of deciding whether a graph is k edge-colorable; for $k = 2$ they prove hardness via a reduction from MAX 3 SAT-3.

of single edge updates and we distinguish the algorithms based on how they handle this sequence:

1. *Batch-dynamic* algorithms first collect the edge updates and then process them all at once when the batch is complete.

2. *Fully dynamic* algorithms process each update individually as soon as it arrives.

3. *Hybrid* algorithms combine fully dynamic algorithms with static algorithms, where the latter are oblivious to the updates and only consider the graph with the updates already applied.

Our two batch-dynamic algorithms `batch-greedy` and `batch-NC` update the coloring after a batch by first uncoloring any updated edge along with its adjacent edges. They then apply the static algorithms `GreedyIt` and `NodeCentered` by Hanauer et al. [74], respectively, to the subgraph induced by these edges, while respecting the colors of the remaining edges. The `batch-greedy` algorithm assigns colors to edges by iterating over colors; in each iteration it computes a greedy matching for the given color on the edges that have not been processed in a previous iteration. The `batch-NC` algorithm proceeds node by node in order of the total weight of the k heaviest incident edges; at each node up to k incident edges are colored with any currently available color.

We give two *fully dynamic algorithms*—`dyn-greedy` and `dyn-kEC`. Both only handle weight increases of uncolored edges and weight decreases of colored edges. Weight increases of colored edges and weight decreases of uncolored edges are ignored, since a good solution before such an update is also a good solution after the update. The algorithm `dyn-greedy` greedily colors candidate edges by assigning a free color or, if no color is free, after uncoloring adjacent edges as long as it leads to an increase in weight. Optionally, the algorithm applies the same process recursively to color any edges that have become uncolored. Candidates are uncolored edges whose weight increases and uncolored edges adjacent to a colored edge whose weight decreases. The second fully dynamic algorithm, `dyn-kEC`, colors candidate edges using a subroutine from the static algorithm `kEC` described by Hanauer et al. [74], which is an adaptation of the edge-coloring algorithm by Misra and Gries [81] to the disjoint matchings problem.

Our third set of algorithms are the *hybrid algorithms*. If the batches update a large part of the graph, then a static algorithm may outperform dynamic ones; similarly, on small batches dynamic algorithms tend to be faster. Our hybrid algorithms recompute from scratch via a static algorithm if the number of updates is sufficiently large, and otherwise use a dynamic algorithm to update the existing coloring. The algorithms decide whether a batch is large by a simple heuristic: the static algorithm is used when more than n edges are updated, i.e., one edge per node on average. We hybridize the algorithm `kEC`, which our experiments confirm to be the fastest static algorithm, with the fully dynamic algorithms `dyn-greedy` and `dyn-kEC`. This yields the algorithms `hybrid-greedy` and `hybrid-kEC`, respectively.

In our instances we are confronted with very large batches, which is a challenging scenario for dynamic algorithms. However, many updates change edge weights only by a small factor. We introduce an update filtering strategy

that removes these updates to reduce the batch size. Such updates are expected to have little effect on the solution weight. Ignoring them should therefore not impact the solution too much, while simultaneously reducing the time to process the batch.

Finally, we give a post-processing algorithm, which improves the weight of any given solution and guarantees a $\frac{1}{2}$-approximation factor by maintaining the following invariant:

Invariant 2.3.1. *For any uncolored edge e and any color c, the edges adjacent to e colored with c are heavier in sum than e.*

Uncolored edges are processed in order of decreasing weight and the invariant is restored for each by swapping colors with adjacent colored edges that violate the invariant. It can also be run as a stand-alone batch-dynamic algorithm `batch-2apx`, which collects any edge for which Invariant 2.3.1 may be violated due to an update. These collected edges are then processed in the same way.

We evaluate the algorithms on 39 real-world and 176 synthetic instances and compare against the best static algorithm `kEC` [74]. The instances tend to be challenging for dynamic algorithms, as the batches are large relative to the graph. Nevertheless, our fastest algorithms are competitive with `kEC` in terms of running time and match its solutions in terms of weight. Compared to `kEC` and other static algorithms the dynamic algorithms generally have low *recourse*, i.e., the number of edges whose color is changed by the algorithm is small. In a datacenter application lower recourse implies fewer changes in the network's configuration and thus reduced overhead. The filtering and post-processing routines are crucial to the success of our dynamic algorithms: filtering leads to significant speedups and any resulting loss in solution weight is recovered by the post-processing.

For different scenarios we identify the algorithms offering the best trade-off between our three performance metrics update time, solution weight and recourse: For small batches and few matchings `batch-2apx` offers the best time-weight trade-off; for small batches and many matchings the dynamic algorithms `dyn-greedy` and `dyn-kEC` provide solutions faster without loss in weight. Finally, when batches are large the hybrid algorithms provide the best performance when considering all metrics.

2.4 Discussion and Outlook

We present new algorithms for data analysis scenarios where the input can change over time. A focus of our work is on performance in practical applications, which we demonstrate with experimental evaluations. An exception are the differentially private algorithms in Chapter 6. In the following we briefly discuss open problems and subsequent related work.

In topological data analysis we contributed the first dynamic algorithm to maintain the persistence diagram of time series. The same approach may be extended to other maps on 1-dimensional objects, such as those studied in [25], i.e., cycles, geometric trees and geometric networks. Persistent homology on such maps has a similar characterization in terms of windows as that on intervals. Adapting banana trees to these settings should be possible without excessive modification. In general, a data structure similar to ours

could be designed for any type of map whose persistent homology admits a characterization similar to the one in [25]. We designed the banana trees without a specific application in mind. In principle, any applications that computes the persistence diagram of a sliding window may benefit from banana trees, e.g., the topological feature engineering approach described by Zeng et al. [21]. Furthermore, banana trees can update the persistence diagram as the time series grows, which should allow applications such as those mentioned in Section 2.1 to be adapted to dynamic settings.

In private analysis of graphs we presented differentially private partially dynamic algorithms for graphs under continual observation, that improve on the state of the art for several subgraph statistics. We also gave partially dynamic algorithms for the value of a minimum spanning tree, a minimum cut, a maximum matching and the densest subgraph. These are complemented by lower bounds that are tight up to logarithmic factors in the length of the graph sequence. Furthermore, we showed that differentially private algorithms on user-level need to have additive error on the order of the maximum function value for a large class of problems. We did not perform an experimental analysis to measure performance of these algorithms. However, they only maintain a small set of counts or thresholds, and their overhead over equivalent non-private algorithms is expected to be negligible. There have since been more results on differentially private algorithms for partially dynamic graphs. Fichtenberger et al. [82] showed how to compute a synthetic graph sequence that approximates all cuts in a graph. Jain et al. [83] presented a method for projecting general graph sequences onto graph sequences with bounded maximum degree, which allows algorithms requiring bounded degree to be applied to general graphs. By combining their projection technique with our algorithms they obtain node-differentially private algorithms under continual observation, which do not require a bound on the maximum degree. While our algorithm based on the difference sequence can in principle be applied to the fully dynamic settings, it turns out that this leads to large errors on most problems, when following our analysis. Recently, Raskhodnikova and Steiner [84] described differentially private algorithms for fully dynamic graphs, along with stronger lower bounds for the event-level and user-level settings. Their algorithm for triangle counting in graph sequences with bounded maximum degree is similar to ours: it employs the difference sequence approach in combination with the binary mechanism and differs from ours is in its use of Gaussian noise over Laplace noise. An improved analysis allows the application to fully dynamic sequences.

In reconfigurable network scheduling we give algorithms to assign pairs of racks to switches by solving the k disjoint matchings problem. Our algorithms can be applied to existing datacenter architectures. However, our approach does not take into account restrictions such as bandwidth of the physical connections. Both the practical evaluation and improved modeling are left as open work. While our algorithms perform well in practice—in comparison to the static baseline—they lack approximation guarantees. Following our work, El-Hayek et al. [85] gave deterministic and randomized approximation algorithms for the unweighted k disjoint matchings problem. Incorporating their ideas into our framework may be of interest. The (weighted) k disjoint matchings problem has recently also been studied in the semi-streaming setting: Ferdous et al. [86] presented semi-streaming algorithms along with

an experimental evaluation, which showed that they are faster and use less memory than the static algorithms by Hanauer et al. [74] with similar solution quality. These semi-streaming algorithms provide a new baseline for ours, and the primal-dual approach may also yield new insight into the dynamic setting.

3 DYNAMICALLY MAINTAINING THE PERSISTENT HOMOLOGY OF TIME SERIES

3.1 Introduction

Persistent homology is an algebraic method aimed at the topological analysis of data; see e.g. [24, 87]. It applies to low-dimensional geometric as well as to high-dimensional abstract data. In a nutshell, the method is the embodiment of the idea that features exist on many scale levels, and rather than preferring one scale over another, it quantifies the features in terms of the range of scales during which they appear. More precisely, the goal of persistent homology is to compute a representation of the features in a book-keeping data structure, called the *persistence diagram* [24].

Importantly, persistent homology has fast algorithms that support the application to large data sets. Specifically, for arbitrary dimensional inputs, persistence is generally computed by analyzing an underlying complex, with worst-case time cubic in the size of the complex. These algorithms are, however, observed to run much faster in applications; see e.g. [88] for a survey on popular implementations. We restrict ourselves to one-dimensional input data, i.e., a list of m points (or *items*) in an interval of $\mathbb{R}$ with each item i being assigned a *value* $f(i)$. Persistent homology has been applied to such time series data in multiple contexts, for example to heart-rate data [8, 9], gene expression data [11, 12], and financial data [14].

The persistent homology of one-dimensional data can be derived from the merge tree [89], which records more detailed information about the structure of the persistence diagram, called the *history of the connected components in the filtration of sublevel sets*. Without recovering this history, the persistence information can be computed in $O(m)$ time [26]. To the best of our knowledge, the new $O(m)$ time algorithm in this paper is the first linear-time algorithm that can also recover the history, which we store in the *augmented persistence diagram of the filtration of sublevel sets*. This diagram is the extended persistence diagram of [22] together with a relation that encodes the merge tree. It is defined formally in Section 3.2.3.

As the data may change, it is an interesting question whether persistent homology can be maintained efficiently under update operations. The historically first such algorithm [28] takes time linear in the complex size per swap in the ordering of the simplices; see also [90, 91]. For one-dimensional data this corresponds to a change of the value of an item, which reduces to a sequence of interchanges of f-values, each costing time linear in the size of the persistence diagram. More recently, [29] has shown that this can be strengthened to logarithmic time if the input complex is a graph. The current paper is the first to maintain the persistent homology of dynamically changing one-dimensional input with a tailor-made data structure under a larger suite of update operations, which includes the *insertion* of a new item, the *deletion* of an item, the *adjustment* of the value of an item, the *cutting* of a list of items into two, and the *concatenation* of two lists into one. The running time per operation is $O(\log n + k)$, in which n is the current number of critical items,

and k is the number of changes to the augmented persistence diagram caused by the operation.

Our novel dynamic data structure is based on the characterization of the items in the persistence diagram through *windows*, as recently established in [25]. The main data structure is a *binary tree* ordered by position as well as value (see [30] for the introduction of such a binary tree), a *path-decomposition* of this tree dictated by persistent homology such that each path represents a window, and a final relaxation obtained by splitting each path into a *left trail* of nodes with right children on the path and a *right trail* of nodes with left children on the path. This split of each path into two trails is crucial for our results, and without it, the update time would have a linear dependence on the depth of the tree, which might be $\Theta(n)$.

OUTLINE. Section 3.2 provides the background needed for this paper: lists and maps, persistent homology, and the hierarchy of windows that characterizes the augmented persistent diagram. In Section 3.3 we present a technical overview of our technique. Section 3.4 introduces the data structures we use to represent a linear list: a doubly-linked list, two dictionaries, and two path-decomposed ordered binary trees whose paths are stored as pairs of trails. Section 3.5 presents the algorithms for maintaining the augmented persistence diagram of a linear list. Section 3.6 concludes the paper.

3.2 Background

This section explains how a linear list can be viewed as a continuous map on a closed interval. Looking at the sub- and superlevel sets of such a map, we define its augmented persistence diagram and review the hierarchical characterization in terms of windows, as proved in [25]. A modest amount of topology suffices to describe these diagrams, and we refer to [24] for a more comprehensive treatment needed to place the results of this paper within a larger context.

3.2.1 Lists Viewed as Maps

By a *linear list* we mean a finite sequence of real numbers, $c_1, c_2, \ldots, c_m$. To view the list as a continuous map, we set $f(i) = c_i$, for $1 \leq i \leq m$, and linearly interpolate between consecutive values. The result is a piece-wise linear map on a closed interval, $f : [1, m] \to \mathbb{R}$, with *items* i and *values* c_i, for $1 \leq i \leq m$. Important about the items is their ordering and not their precise positions along the interval, so we use consecutive integers for convenience. To simplify discussions, we will often assume that the map is *generic*, by which we mean that its items have distinct values. This is no loss of generality since a small perturbation may be simulated and implemented by appropriate tie-breaking rules.

Given $t \in \mathbb{R}$, the *sublevel set* of f at t, denoted $f_t = f^{-1}(-\infty, t]$, is the set of points in domain of f such that its value $f(x)$ is not larger than any fixed point $t \in \mathbb{R}$. Since f is defined on a closed interval, the homology of f_t is fully characterized by the number of connected components. A point x in the closed interval, $[1, m]$ is a *(homological) critical point* of f if the number of

connected components of f_t changes at the moment t passes $f(x)$. Assuming genericity, a critical point is necessarily an item of f, and the only two types in the interior of $[1,m]$ are *minima* and *maxima*. When t passes the value of a minimum from below, then the number of connected components of f_t increases by 1, and if t passes the value of a maximum from below, the number of connected components decreases by 1. The endpoints of $[1,m]$ are special: the number of connected components changes at an *up-type endpoint* and it remains unchanged at a *down-type* endpoint. Symmetrically, we call $f^t = f^{-1}[t,\infty)$ the *superlevel set* of f at t. Observe that f^t is the sublevel set of $-f$ at $-t$, and that the minima and maxima of $-f$ are the maxima and minima of f. Similarly, the endpoints swap type.

3.2.2 Extended Persistent Homology

Persistent homology tracks the evolution of the connected components while the sublevel set of f grows, and formally defines when a component is born and when it dies. Complementing this with the same information for the superlevel sets of f, we get what is formally referred to as *extended persistent homology*, which we explain next (see [22] for more details).

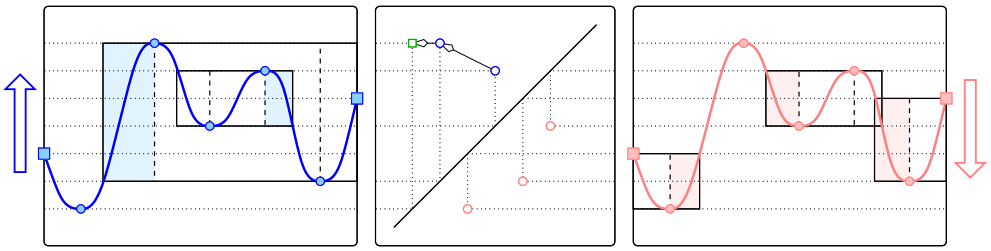

Figure 3.2.1: *Left:* a real-valued map on a closed interval, f, with three minima and two maxima. *Right:* the map $-f$ drawn upside-down. *Middle:* the augmented persistence diagram with two (*blue*) points in the ordinary subdiagram $\mathsf{Ord}(f)$ above the diagonal, three (*pink*) points in the relative subdiagram $\mathsf{Rel}(f)$ below the diagonal, and one (*green*) point in the essential subdiagram $\mathsf{Ess}(f)$.

In *Phase One*, we track the connected components of the sublevel set, f_t, as t increases from $-\infty$ to ∞. A component is *born* at the smallest value of t at which a point of the component belongs to f_t. This point is necessarily a minimum in the interior of $[1,m]$, or an up-type endpoint. The component *dies* when it merges with another component that was born earlier. The point at which the two components merge is necessarily a maximum in the interior of $[1,m]$. The *ordinary subdiagram* of f, denoted $\mathsf{Ord}(f)$, records the birth and death of every component with a point in the plane whose abscissa and ordinate are those values of t at which the component is born and dies, respectively; see Figure 3.2.1.

In *Phase Two*, we track the connected components of the superlevel set, f^t, as t decreases from ∞ to $-\infty$. Birth and death are defined accordingly, and the components are recorded in the *relative subdiagram*, denoted $\mathsf{Rel}(f)$. By construction, the points in $\mathsf{Ord}(f)$ lie above and those of $\mathsf{Rel}(f)$ lie below the diagonal; see Figure 3.2.1. The component born at the global minimum of f is special because it does not die during Phase One. Instead, it dies at the global minimum of $-f$, which is the global maximum of f. In topological

terms, this happens because the one connected component still alive at the beginning of Phase Two dies in relative homology when its first point enters the superlevel set.[a] This class is represented by the sole point in the *essential subdiagram*, denoted $\mathsf{Ess}(f)$; see again Figure 3.2.1. The *extended persistence diagram*, denoted $\mathsf{Dgm}(f)$, is the disjoint union of the three subdiagrams. Hence, $\mathsf{Dgm}(f)$ is a multi-set of points in $\mathbb{R}^2$, and so are the three subdiagrams, unless the map is generic, in which case the diagram is a set.

3.2.3 Windows and Augmented Persistence Diagram

The points of the persistence diagram can be characterized using the concept of windows recently introduced in [25]. Let a and b be two (homological) critical points of f, with values $A = f(a) < f(b) = B$. Note that $f^{-1}[A,B]$ contains all points in the interval whose values lie between A and B. It consists of one or more connected components, and it is quite possible that a and b belong to different components. However, if they belong to the same component, then we denote by $[x,y]$ the connected component of $f^{-1}[A,B]$ that contains both a and b and we call $[x,y]\times[A,B]$ the *frame* with *support* $[x,y]$ spanned by a and b. There are two orientations of the frame: from left to right if $a<b$, and from right to left if $b<a$. In the former case, we call x the *mirror* of b and $W(a,b)$ a *triple-panel window* if $f(y)=A$. In the latter case, we call y the *mirror* of b and $W(a,b)$ a *triple-panel window* if $f(x)=A$. We say a and b *span* $W(a,b)$. We distinguish a special type of window, referred to as a *global window*, whose support covers the entire interval, $[x,y]=[1,m]$, and which is exempt of the requirement at x or y.

In [25], the authors prove that there is a bijection between the windows and the points in $\mathsf{Dgm}(f)$. Furthermore, each critical point in the interior of the interval spans exactly one window in each phase—even though the pairing may be different in the two phases—while each endpoint spans a window in only one phase. In Figure 3.2.1, there are five triple-panel windows, two on the left and three on the right. There is always exactly one global window spanned by the global minimum, α, and the global maximum, β, of f. What follows is a special case of a more general characterization of persistence proved in [25].

Proposition 3.2.1 (Persistence in Terms of Windows [25])**.** *Let $f:[1,m]\to\mathbb{R}$ be a generic piecewise linear map on a closed interval, and let a,b be homological critical points of f, or of $-f$, with $f(a)=A$ and $f(b)=B$. Then*

(i) $(A,B)\in\mathsf{Ord}(f)$ *iff $W(a,b)$ is a triple-panel window of f,*

(ii) $(B,A)\in\mathsf{Rel}(f)$ *iff $W(b,a)$ is a triple-panel window of $-f$,*

(iii) $(A,B)\in\mathsf{Ess}(f)$ *iff $W(a,b)$ is the global window of f.*

a In the original formulation in [22], Phase Two in the construction of the (extended) persistence diagram tracks the 1-dimensional relative homology groups of the pairs (f_∞, f^t), which are isomorphic to 0-dimensional homology groups of the f^t. To appreciate the guiding hand of algebraic topology, we need to understand how the absolute and relative homology groups of different dimensions relate to each other. However, for the purpose of this paper, this is not necessary and we can track the connected components of the superlevel set in lieu of the relative cycles in the interval modulo the superlevel set.

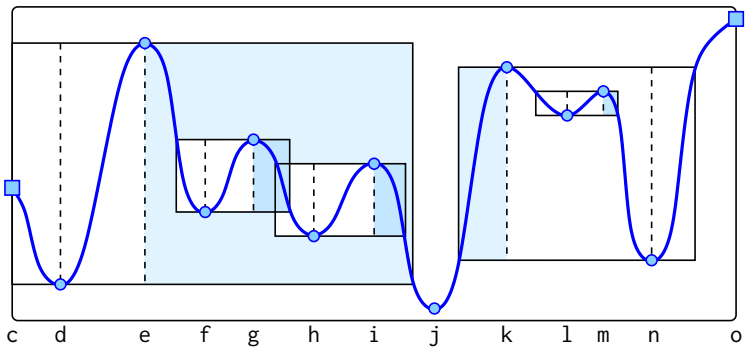

Figure 3.2.2: The graph of a generic map on a closed interval. All windows shown are with simple wave, except for the *leftmost* window, whose wave is short. The global window as well as the (tiny) windows caused by the hooks (which will be introduced in Section 3.4) are not shown. The *light-blue shaded* out-panels are part of the triple- but not of the double-panel windows.

Suppose $W(a,b)$ is a triple-panel window, with support $[x,y]$. Unless a is an endpoint of $[1,m]$, cutting $[x,y]$ at a and b produces three segments, $[x,a]$, $[a,b]$, $[b,y]$, in which we assume $a<b$. We call the corresponding products with $[A,B]$ the *in-panel*, *mid-panel*, *out-panel*, and the graph within the window a *wave*. This wave is *simple* or *short* depending on whether the function value at the mirror is equal to or smaller that at the maximum: $f(x)=B$ or $f(x)<B$. In Figure 3.2.2, we have four simple waves (defined by $\mathtt{f},\mathtt{g}$, by $\mathtt{h},\mathtt{i}$, by $\mathtt{l},\mathtt{m}$, and by $\mathtt{n},\mathtt{k}$) and one short wave (defined by $\mathtt{d},\mathtt{e}$). Note that a simple wave of f is also a simple wave of $-f$, albeit upside-down. A similar statement does not hold for short waves, which explains the violations of symmetry in the persistence diagram; see again Figure 3.2.1.

The *double-panel* version of a triple-panel window consists of the in-panel and the mid-panel but drops the out-panel. Any two double-panel windows of f are either disjoint or nested, and all these windows are nested inside the global window [25, Lemma 3.3]. We use this property to augment the persistence diagram with the merge tree information; that is: we draw an arrow between two points if the corresponding double-panel windows are nested without any other window nested between them. The result is the *augmented persistence diagram*, denoted $\mathsf{D}\overrightarrow{\mathsf{gm}}(f)$.

In this paper, we consider operations that maintain the augmented persistence diagram to reflect the change from a map, f, to any other map, g. We quantify this difference by the number of points and arrows that change or, more formally, belong to the symmetric difference between the two diagrams. What are typical differences? Every local change in the function reduces to a sequence of *interchanges* between two minima or two maxima, possibly ending in a *cancellation* or beginning with an *anti-cancellation*. Every cancellation removes a point from the diagram, and every anti-cancellation adds a point to the diagram, so the diagrams before and after differ by a constant amount. An interchange may or may not swap two critical values between points (two coordinates, which are values of two critical items) or the reversal of an arrow, and if it does not, then this should not incur any cost. On the other hand, if it does, then this can cost a constant amount of time. Our declared goal is to have operations that run in time $O(\log n+k)$, in which n is the number of criti-

cal points and k measures the difference between the augmented persistence diagrams before and after the operations.

3.3 Technical Overview

To achieve this goal, we propose a novel collection of data structures—some classic and some new—for maintaining nested windows. We show that using these data structures allows us to maintain the augmented persistence diagram to reflect the change from one map to another in the desired time bound. We maintain the following data structures:

(1) a *doubly-linked list* of all items, critical or not, ordered by their positions in the interval;

(2) two *binary search trees*, called *dictionaries*, one storing the minima and the other storing the maxima, both ordered by their positions in the interval;

(3) two *banana trees* (described next) representing the information in the augmented persistence diagram by storing the minima and maxima while reflecting their ordering by position as well as by function value.

Note that the minima and maxima are subsets of all items and are therefore represented in all of the above data structures. To reduce the special cases in the algorithms, we add two artificial items, called *hooks*, at the very beginning and the very end of the interval. They make sure that the formerly first and last items are proper minima or maxima, but we ignore them and the technicalities involved in this overview and give slightly simplified descriptions of the data structures and the algorithms.

BANANA TREES. We introduce these trees in three stages. In the *first stage*, we organize all windows in a full binary tree, whose leaves are the minima and whose internal nodes are the maxima, such that (i) the in-order traversal of the tree visits the nodes in increasing order of their positions in the interval, and (ii) the nodes along any path from a leaf to the root are ordered by increasing function value. Such a tree always exists and it is unique. In the following, we do not distinguish between a node in the tree and the critical item it represents.

In the *second stage*, we add a *special root* labeled with value larger than the global maximum whose only child is the previous root. Call the resulting tree T and note that it has an equal number of leaves and internal nodes. We then form paths, each starting at a leaf, a, and ending at an internal node, b, such that a and b span a window, $W(a,b)$. Call this path $P(a,b)$. Based on structural properties of T, this leads to a partition of T into edge-disjoint (but not vertex-disjoint) paths; see the left drawing in Figure 3.3.1. The node b ending the path that starts at a is locally determined: it is the first internal node encountered while walking up from a, such that a and the descending leaf with minimum function value lie on different sides (in different subtrees) of this internal node. Hence, every maximum on the path from a to b spans a window that is immediately nested in $W(a,b)$. It follows that every maximum, b, belongs to two paths: $P(a,b)$ and $P(p,q)$ such that $W(a,b)$ is immediately

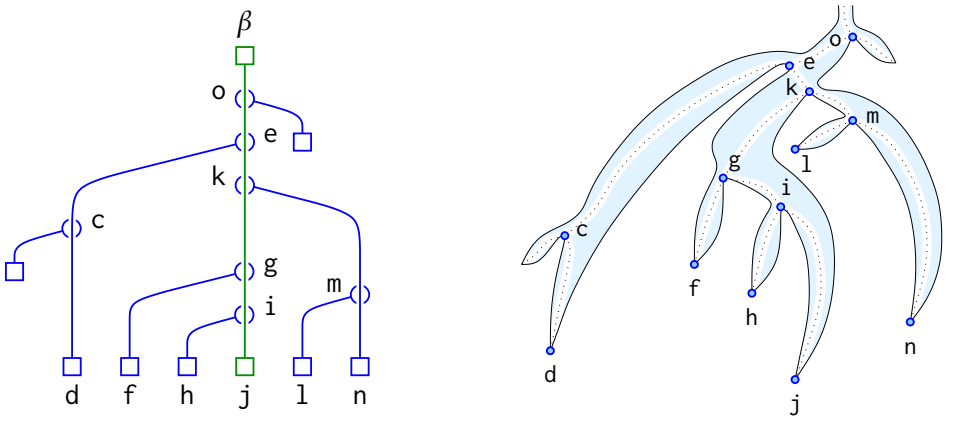

Figure 3.3.1: The path-decomposed binary tree associated to the map in Figure 3.2.2 with special root, β, on the *left*, and the corresponding banana tree on the *right*.

nested in $W(p,q)$. This even holds for the root (of the full binary tree), which spans a path and also belongs to the path that ends at the special root. Given a map, f, the tree, T, and its partition into paths are unique. The strict dependence on f may force T to be unbalanced, and indeed have linear depth, so that efficient maintenance algorithms are challenging. This is why we need another modification.

In the *third stage*, we split each path into two trails; see the right drawing in Figure 3.3.1. The *left trail* of $P(a,b)$ contains a and every maximum u on $P(a,b)$ with $u \leq a$, while the *right trail* contains a and every maximum u on $P(a,b)$ with $a \leq u$. A node v on $P(a,b)$ is the right child of its parent, u, in the binary tree iff u is on the left trail. Furthermore, v is the left child of u iff u is on the right trail. Thus, to which trail u belongs to can be decided based on local information at u. To simplify language, we also give a second name to the trails. If $a < b$, then $W(a,b)$ consists of the in-panel on the left, the mid-panel in the middle, and the out-panel on the right. In this case, u belongs to the left trail iff the window it spans is nested inside the in-panel, so we alternatively call the left trail the *in-trail*, and we alternatively call the right trail the *mid-trail*. If $b < a$, then the right trail is the in-trail and the left trail is the mid-trail. So what happened to the out-trail? It is indeed needed, but only if $W(a,b)$ is with simple wave. Such a window corresponds to $P(a,b)$ in the banana tree of f and to $P(b,a)$ in the banana tree of $-f$. The out-panel of $W(a,b)$ is the in-panel of $W(b,a)$, so we can get information about the out-panel from the in-panel of the same window in the other banana tree when we need it.

To speed up the algorithms, we maintain various pointers, such as between the occurrences of the same critical item in the banana trees, the dictionaries, and the doubly-linked list, for example. Importantly, every internal node, b, stores a pointer to the descending leaf with minimum function value, $\mathsf{low}(b)$, and every leaf, a, stores a pointer to the endpoint of its path, $\mathsf{dth}(a)$. Observe that $q = \mathsf{dth}(\mathsf{low}(b))$ is the maximum that spans the window in which $W(a,b)$ is immediately nested, so this window can be obtained in constant time.

CONSTRUCTION. While the main focus of this paper is the maintenance of the data structures through local updates, we also consider the construc-

tion from scratch. It is straightforward to derive the augmented persistence diagram during a single traversal of the banana trees in linear time, so the question we study is how fast this diagram can be constructed from a given one-dimensional input list. Assuming all non-critical items have been removed and we are given the remaining sequence of n critical items, there are standard algorithms that can be adapted to construct the banana trees in $O(n \log n)$ time. There is also an $O(n)$ time algorithm for computing the persistence diagram [26], but this algorithm does not extend to the augmented persistence diagram. To the best of our knowledge our algorithm is the first to construct the augmented persistence diagram in $O(n)$ time.

The main structure of the algorithm is a left-to-right scan of the data. We interpret the item i with value $c_i = f(i)$ as the point (i, c_i) in the plane and maintain a decreasing staircase such that all processed items are points on or below the staircase. Each step of the staircase corresponds to an unfinished banana. One of the difficulties is that before a banana is completed, we do not know whether it will be attached to a left or a right trail. We tentatively assume it will be attached to a left trail but are prepared to move the banana to the other side when this turns out to be necessary. When we process the next item, we may remove any number of steps, turning each into a finished banana, but we can add at most one new step. Since a step that is removed was added earlier, this proves that the algorithm runs in $O(1)$ amortized time per item, and therefore in $O(n)$ time altogether.

LOCAL MAINTENANCE. Given a list of m items with real function values, we consider the operations that *insert* a new item, *delete* an item, and *change the value* of an item. All three operations reduce to a sequence of *interchanges*—which can be between two maxima or between two minima—possibly preceded by an *anti-cancellation* or a *slide*, and possibly succeeded by a *cancellation* or a *slide*. In a slide, a minimum or maximum next to a non-critical item becomes non-critical, and the non-critical item becomes a minimum or maximum, respectively. Similarly in a cancellation, a minimum and a neighboring maximum simultaneously become non-critical. Here we will focus on the interchanges, because they are most common as well as most interesting, and on the anti-cancellations, because they pose an unexpected challenge.

Consider two maxima, b and q, of f, and assume $f(b) < f(q)$ before the interchange. To avoid confusing language, we write $g(b)$ and $g(q)$ for the values after the operation but assume that f and g agree on all items except for b. Furthermore, we assume that $g(b) > g(q)$ and that b and q are the only two items for which the ordering by f-value differs from the ordering by g-value. In many cases, the interchange of b and q does not affect the banana trees. Indeed, only if b is a child of q is it necessary to update the order of the two nodes. And even if b and q are consecutive maxima on a path, there is no structural change unless b and q also belong to a common trail. This is the main reason for splitting each path into two trails as explained in the third stage of the introduction of the banana trees: to avoid any cost to occur for interchanges that have no structural affect on the augmented persistence diagram. When b and q interchange while belonging to different trails, then the banana tree is oblivious to this change and requires no update. On the other hand, if b and q belong to the same trail, then they swap positions along

this trail, and there is a change of the augmented persistence diagram that pays for the time it takes to update the banana tree.

The interchange of two minima, a and p, is quite different because it does not affect the ordered binary tree at the first stage of the banana tree. However, the interchange affects the path-decomposition, so some of the bananas may have to be updated. To be specific, assume $f(a) > f(p)$ before and $g(a) < g(p)$ after the operation. As before, we also assume that f and g agree on all items except for a, and that a and p are the only two items for which the ordering by f-value differs from the ordering by g-value. Let b and q be the internal nodes so that $P(a,b)$ and $P(p,q)$ are paths in the decomposition of the banana tree of f. The interchange of a and p has no effect on the banana tree, unless b is a node on $P(p,q)$. If b lies on $P(p,q)$, then we extend $P(a,b)$ to $P(a,q)$ and we shorten $P(p,q)$ to $P(p,b)$. The nodes u on the path from b to the child of q change their pointer from $\mathsf{low}(u) = p$ to $\mathsf{low}(u) = a$. There can be arbitrarily many such nodes, but each change causes the adjustment of an arrow in the extended persistence diagram, to which it can be charged in the running time analysis.

This shows that it is possible to perform an interchange of two minima within the desired time bound, but it is not clear how to find them. Considering the scenario in which a decreases its value continuously, it may cause a sequence of interchanges with other minima, but since these minima are not sorted by function value, it is not clear how to find them, and how to ignore the ones without structural consequences. Here is where the relation between the banana trees of f and $-f$ becomes important. The minima of f are the maxima of $-f$, so the interchange of two minima in the banana tree of f corresponds to the interchange of two maxima in the banana tree of $-f$. We already know how to find the interchanges of two maxima and how to ignore the ones without structural consequences, so we use them to identify the interchanges of minima. More precisely, we prove that the interchange of the minima, a and p of f, affects the structure of the banana tree of f only if the interchange of the maxima, a and p of $-f$, affect the structure of the banana tree of $-f$. The converse does not hold, but the implication suffices since the interchange of the maxima of $-f$ leads to a change in the augmented persistence diagram which can be charged for the interchange of the minima, which costs only $O(1)$ time if there are no structural adjustments.

Next, we sketch what happens in the remaining operations. A *slide* occurs if a maximum decreases its value so that it becomes non-critical, while a neighboring non-critical item becomes a maximum. However, if this neighboring item is a maximum, then both items become non-critical at the same time, in which case the operation is called a *cancellation*. There are also the symmetric operations in which a minimum increases its value and becomes non-critical, while a neighboring item changes from non-critical to minimum (a *slide*) or from maximum to non-critical (a *cancellation*). The corresponding updates are easily performed within the required time bounds.

A more delicate operation is the *anti-cancellation*, in which two neighboring non-critical items become critical at the same moment in time. Let a and b be these two items and assume a is a minimum and b is a maximum after the operation, so $g(a) < g(b)$ and, by assumption, $f(a') > f(a) > f(b) > f(b')$, in which a', a, b, b' are four consecutive items in the list. Hence, a and b are not present in the banana tree of f, but $P(a,b)$ is a path in the path-decomposed

tree of g, so a, b form a minimal banana in the banana tree of g. All we need to do is find the correct place to attach this banana, but this turns out to be difficult. We first explain why it is difficult, then present an algorithm that works within the current data structure, and finally sketch a modification of the data structure that accelerates the anti-cancellation to $O(\log n)$ time. While this leads to a speed-up in the worst case for this type of operation, it slows down other operations by a logarithmic factor.

We use mirrors to explain in what situation an anti-cancellation is difficult. Their representation in the banana tree is indirect: the maximum is the upper end of a mid-trail, and its mirror is the upper end of the matching in-trail. In the tree, the two upper ends are the same node, but if we traverse the trails of the banana trees in sequence, we visit them at different times, and these times correspond to the positions along the interval, which are different for the maximum and its mirror. Every maximum has at most one mirror, so adding all mirrors to the list increases it to less than double the original size. Nevertheless, it is easy to construct a case in which there is a long subsequence of mirrors, and these mirrors are consecutive with the exception of a and b appearing somewhere in their midst. In this situation, finding the correct attachment of $P(a, b)$ in the banana tree of g needs the mirrors immediately to the left and right of b. The data structure as described provides no fast search mechanisms among the mirrors, so we find the correct attachment by linear search starting from the first maximum to the left of a. Suppose we pass k mirrors before we find this attachment. Then we have a nested sequence of k windows, and $W(a, b)$ is nested inside all of them. Hence, there is one new immediate nesting pair, while the transitive closure of the nesting relation gains k new pairs. The cost of the anti-cancellation can be charged to the change of this transitive closure. In many cases, the change in the transitive closure will be comparable in size to that of the transitive reduction, but it can also be significantly larger, like in the case we just described. We thus entertain alternatives.

There is a modification of the data structure that allows for fast searches among subsequences of mirrors: for any consecutive min-max pair in the list, store the mirrors between them in a binary search tree. In practice, there will be many very small such trees, but it is possible that the mirrors accumulate and produce a few large trees. With this modification, an anti-cancellation can be performed in $O(\log n)$ time. Note however, that these binary search trees have to be maintained, which adds a factor of $O(\log n)$ to the time of every operation, in particular to every interchange, of which there can be many.

TOPOLOGICAL MAINTENANCE. We call an operation *topological* if it cuts the list of points into two, or it concatenates two lists into one. More challenging topological operations, such as gluing the two ends of a list to form a cyclic list, or gluing several lists to form a geometric tree or more complicated geometric network are feasible but beyond the scope of this paper.

When we *cut* the list of data into two, we *split* the banana trees of the map and its negative into two each. We mention both banana trees since we need information from the other to be able to split one within the desired time bound. Splitting one banana tree is superficially similar to splitting a binary search tree, but more involved. Let $f\colon [1, m] \to \mathbb{R}$ be the map before the operation and $g\colon [0, \ell] \to \mathbb{R}$ and $h\colon [\ell + 1, m] \to \mathbb{R}$ the maps after the operation.

We write $z = \ell + \frac{1}{2}$ for the position at which the time series is cut. Since the banana trees are determined by the maps, we need to understand the difference between the windows of f and those of g and h. Whether or not a frame of f is also a window depends solely on the restriction of f to the support of the frame. To decide about the future of a window of f, let $[x, y]$ be its support. This window is also a window of g if $y < z$, and it is also a window of h if $z < x$. The windows that need attention are the ones with $x < z < y$. A triple-panel window consists of three panels, so we distinguish between three cases:

- $W(a, b)$ suffers an *injury* if z cuts through the in-panel. Then a and b lie on the same side of z and they still span a window, albeit with short wave.
- $W(a, b)$ suffers a *fatality* if z cuts through the mid-panel. Then a and b lie on different sides of z and need to find new partnering critical items to span new windows.
- $W(a, b)$ suffers a *scare* if z cuts through the out-panel. These windows are difficult to find in the banana tree of f, but they are easy to find in the banana tree of $-f$, in which $W(b, a)$ suffers an injury.

The splitting of the banana tree proceeds in three steps: first, find the smallest banana that suffers an injury, fatality, or scare; second, find the remaining such bananas; and third, split the banana tree of f into the banana trees of g and h. We address all three steps and highlight the most interesting feature in each.

The first step is difficult because of the lack of an appropriate search mechanism in the banana tree. To explain this, we recall that the banana tree stores the mirrors implicitly, as the upper ends of the in-trails. Like in the case of an anti-cancellation, the challenging case is when mirrors accumulate and we have to locate z in their midst. As before, we locate z by linear search, scanning the mirrors in the order of decreasing function value. In contrast to the anti-cancellation, we can now charge the cost for the search to the changes in the extended persistence diagram. Indeed, every mirror we pass belongs to a window that experiences a scare. Such a window of f is neither a window of g nor of h, so its spanning critical items will re-pair and span new windows after the operation.

The second step traverses a path upward from the smallest affected banana we just identified. In each step of the traversal, we determine the corresponding window and push it onto the stacks of windows that experience an injury, fatality, or scare. A window that experiences an injury remains a window, but now with short instead of simple wave. Whether this window with short wave belongs to the banana tree of g or that of h depends on whether the spanning minimum is to the right or the left of the spanning maximum. A window that experiences a fatality falls apart, with one of the two spanning critical items in g and the other in h. Finally, a window that experiences a scare stops to be a window, in spite of having both critical points in g or in h. As mentioned earlier, such a window is difficult to find in the banana tree of f, but it is easy to find in the banana tree of $-f$, where it experiences an injury. We thus process both banana trees simultaneously, and distributed the windows or

their corresponding bananas as needed. The upward traversal halts when we reach the *spine* of the tree, by which we mean the sequence of left children that start at the root, or the sequence of right children that start at the root. This is important because moving further until we reach the root is not necessary and can be costly because the spine can be arbitrarily long and the steps towards the root cannot be charged to changes in the extended persistence diagram.

The third step re-pairs the critical items of the windows that experience a fatality or scare. All new windows are with short wave, for else they would be windows of f before the splitting, which is a contradiction. Hence, their bananas belong to the spines of the banana trees of g and h. The bananas in the spines have the particularly simple structure that their critical items come in sequence. For the right spine of the banana tree of g, this sequence increases from right to left, for the left spine of the banana tree of h, the sequence increases from left to right, and both are consistent with the sequence in which we collect the corresponding windows in the second step. It follows that the available critical items can be paired up in sequence, which takes $O(1)$ time per item.

Corresponding to the concatenation of two lists, we pairwise *glue* the banana trees of the two maps. The operation is the inverse of splitting, so we omit further details. A final word about the cost paid by the changes in the augmented persistence diagram. How do we compare one diagram with two, which we get after splitting? To deal with this issue, we consider the maps g and h to be *one* map, namely a map on two intervals. Then we still have one augmented persistence diagram and the symmetric difference between the diagrams before and after the operation is well defined.

3.4 Data Structures

We use five inter-connected data structures to represent a linear list or, equivalently, the implied piecewise linear map: a doubly-linked list of all items, two dictionaries storing the minima and maxima for quick access, and two ordered trees representing the information in the augmented persistence diagram for quick update. The novel aspect is the implementation of the ordered trees as *banana trees*, to be described shortly. There are two such trees, storing the homological critical points of the map and its negation, which are subsets of all items. Indeed, the non-critical items are stored only in the doubly-linked list, but they enter the dictionaries and banana trees when they become critical. The difference between critical and non-critical items is defined in terms of the piecewise linear map implied by the items. Let $m \geq 2$ and write $c_1, c_2, \ldots, c_m$ for the list of values, which we assume are distinct. We construct the map, $f : [0, m+1] \to \mathbb{R}$, by setting $f(i) = c_i$, for $1 \leq i \leq m$, and adding artifical ends by setting

$$f(0) = c_1 + \varepsilon \cdot \operatorname{sign}(c_2 - c_1), \tag{6}$$

$$f(m+1) = c_m + \varepsilon \cdot \operatorname{sign}(c_{m-1} - c_m), \tag{7}$$

in which $\operatorname{sign}(c) = \pm 1$ depending on whether $c > 0$ or $c < 0$, and $\varepsilon > 0$ is arbitrarily small and in any case smaller than the absolute difference between any two of the c_i. We call these artificial ends *hooks*; they make sure that items 1 and m are proper minima or maxima.

3.4.1 Dictionaries

We maintain the items in a *doubly-linked list* ordered according to their position in the interval. For reasons that will become clear shortly, we additionally store the minima and maxima in a *dictionary* each, both ordered like the doubly-linked list. Besides searching, the dictionaries support the retrieval of the minimum or maximum immediately to the left or the right of a given $x \in \mathbb{R}$. It will be convenient to write n for the number of maxima so that $n-1$, n, or $n+1$ is the number of minima. In addition, we will pretend that the items—and sometimes just the minima and maxima—are at consecutive integer locations along the real line. Obviously, this cannot be maintained as we insert and delete items, but it makes sense locally and simplifies the notation and discussions without causing any confusion.

An opportune data structure for the dictionary is a binary search tree, which supports `access`, `insertion`, and `deletion` of an item in $O(\log n)$ time each. Similarly it supports the `cutting` of a dictionary into two, and the `concatenation` of two dictionaries into one—provided all items in one dictionary are smaller than all items in the other—again in $O(\log n)$ time. For comparison, doubly-linked lists can be `cut` and `concatenated` in constant time, provided the nodes where the cutting and concatenation is to happen are given.

3.4.2 Banana Trees

Let $f : [0, m+1] \to \mathbb{R}$ be a generic piecewise linear map constructed from a list of m distinct values. The main data structure consists of two trees, one for f and the other for $-f$, which we explain in three steps. In order to ensure that all critical items are represented in both trees, we require homological critical points of f to also be homological critical points of $-f$, and vice versa. This affects how we treat up- and down-type items: up-type items in f are already homological critical points; in $-f$, however, they are down-type items, which are not homological critical points. We use the hooks at down-type items to treat them as proper maxima, and ignore the hooks at up-type when constructing the trees. To describe the tree for f, let $a_0 < b_1 < a_1 < \ldots < b_n < a_n$ be the homological critical points of f, and note that the a_i are minima—with the possible exception of a_0 and a_n, which may be up-type endpoints that are artificially added as hooks. On the other hand, all b_i are proper maxima.

FIRST STEP. We construct a full binary tree whose nodes are the homological critical values. Among the many choices, we arrange these values such that

I.1 the in-order sequence of the nodes in the tree is the ordering of the homological critical points in the linear list, i.e. of their position in the interval;

I.2 the nodes along any path from a leaf to the root are ordered by increasing values.

It is not difficult but important to see that there is a unique full binary tree that satisfies Conditions I.1 and I.2. Indeed, the largest value is a maximum, b_j, which is necessarily the root of the tree. The left subtree is the recursively

defined tree of $a_0, b_1, \ldots, a_{j-1}$, and the right subtree is the recursively defined tree of $a_j, b_{j+1}, \ldots, a_n$. By induction, the two subtrees are unique, so the entire tree is unique.

SECOND STEP. We decompose the tree into edge-disjoint paths, each connecting a leaf to an internal node. Since there is one extra leaf, we add a *special root*, β, whose only child is the root, as an extra internal node. We define β to be greater than all items, both in terms of function value and along the interval; that is: $f(i) < f(\beta)$ and $i < \beta$ for all items i. With this small change, the paths define a bijection between the leaves and the internal nodes. Again there are many choices, and we pick the paths such that

II each path connects a leaf, a, to the nearest ancestor, b, for which a does not have the smallest value in the subtree of b, and to the special root, if no such ancestor exists.

After fixing the tree in the first step, Condition II implies a unique partition of the edges into $n+1$ paths; see Figure 3.4.1. Comparing with the windows introduced in Section 3.2, we note that each path corresponds to a double-panel window: the lowest and highest nodes are the minimum and maximum spanning the window, and all other nodes of the path are maxima of nested windows in the in-panel or the mid-panel of the double-panel window. The out-panel is indirectly represented by the requirement that its highest node is interior to another path whose lowest node has smaller value than the lowest node of the current path. If the window is with simple wave, then the triple-panel window is also represented in the down-tree, except that in- and out-panels are exchanged.

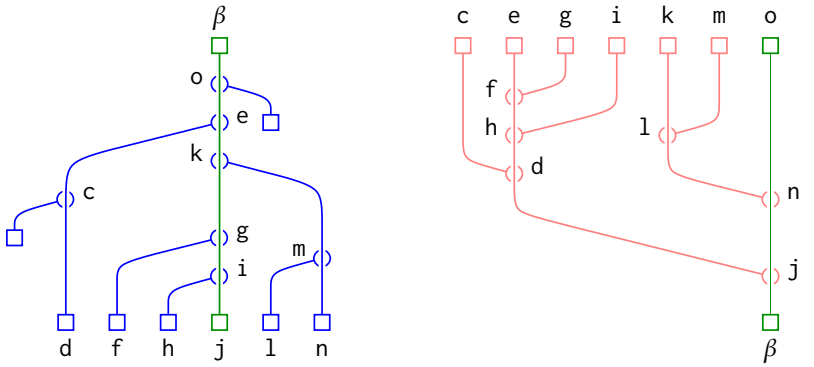

Figure 3.4.1: *Left:* the path-decomposition of the binary tree for the map, f, displayed in Figure 3.2.2. The nodes in its spine (to be defined shortly) are c, e, β, and o. *Right:* upside-down drawing of the path-decomposition of the binary tree for $-f$. The nodes in its spine are d, j, and β.

THIRD STEP. We split every path into two parallel trails. Let $a = q_0, q_1, \ldots, q_{\ell-1}, q_\ell = b$ be such a path, and recall that $f(q_i) < f(q_{i+1})$ for $0 \le i \le \ell - 1$. Assuming $a < b$, the *in-trail* consists of a, every q_i whose right child is q_{i-1}, and b, and the *mid-trail* consists of a, every q_i whose left child is q_{i-1}, and b. The items in the in-trail are smaller than those in the mid-trail, which motivates the alternative terminology of the *left trail* for the former and the *right trail* for

the latter. If $a > b$, the in-trail is the right trail and the mid-trail is the left trail, so the in-trail corresponds to the in-panel and the mid-trail to the mid-panel in either case. Both trails *start* at a and *end* at b. All other q_i are *interior* to the trails. For the special root, β, and the corresponding lower end of its path, α, the order is not defined, so we make an arbitrary choice and call the left trail the *in-trail* and the right trail the *mid-trail*. We call a trail *empty* if it contains no interior nodes. In both trails, the items as well as their values are sorted. To state this formally, assume again that $a < b$:

III.1 writing $a = u_0, u_1, \ldots, u_j = b$ for the nodes along the in-trail, we have $u_i > u_{i+1}$ for $0 \le i \le j-2$ and $f(u_i) < f(u_{i+1})$, for $0 \le i \le j-1$;

III.2 writing $a = v_0, v_1, \ldots, v_k = b$ for the nodes along the mid-trail, we have $v_i < v_{i+1}$ and $f(v_i) < f(v_{i+1})$, for $0 \le i \le k-1$.

The respective first inequalities in III.1 and III.2 imply that the nodes of the in-trail precede the nodes of the mid-trail. In contrast, there is no particular order between the values of nodes in different trails. We draw the trails roughly parallel, like the outline of a banana. Letting a, b be the lower and upper ends, we call the pair of trails the *banana* spanned by a, b. Except if b is the special root, b is also internal to another trail, so b is where the banana spanned by a, b connects to another banana; see Figure 3.4.2. We summarize the crucial properties of bananas in a lemma whose proof follows directly from the construction and is thus omitted.

Lemma 3.4.1 (Bananas and Windows)**.** *Let a, b be critical points of a generic map, f, with $f(a) < f(b)$. The two points span a banana in $Up(f)$ iff $W(a, b)$ is a window of f. Furthermore, another window, $W(p, q)$, is immediately nested in the in-panel or mid-panel of $W(a, b)$ iff its maximum, q, is an interior node of the in-trail or mid-trail of the banana spanned by a and b, respectively.*

Each node stores pointers to its neighbors along the trails: for a node p, $\mathsf{in}(p)$ and $\mathsf{mid}(p)$ point to the first node on the in- and mid-trails beginning at p, respectively; a maximum q has additional pointers $\mathsf{up}(q)$ and $\mathsf{dn}(q)$, which point to the node above and below q on the same trail. The banana tree uses

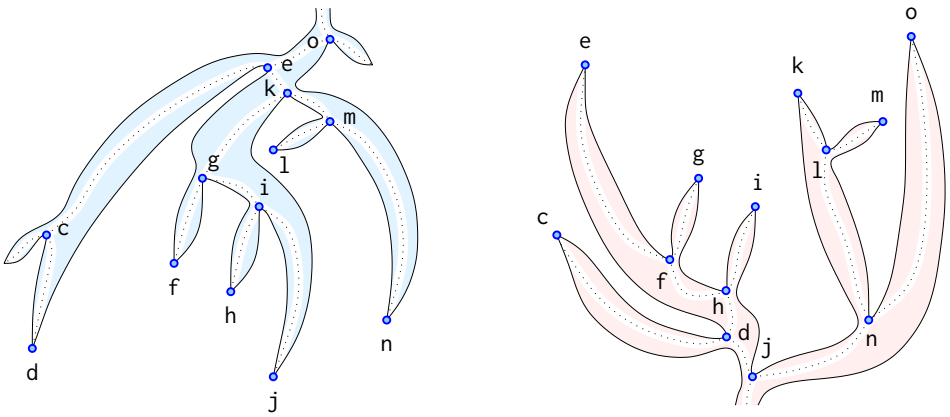

Figure 3.4.2: *Left:* the banana tree of the map in Figure 3.2.2, with dotted curves showing the tree before splitting its paths; compare with the *left* drawing in Figure 3.4.1. *Right:* upside-down drawing of the banana tree for the negated map; compare with the *right* drawing in Figure 3.4.1.

additional pointers to connect the ends of its trails: letting a and b be the lower and upper ends of a trail, we store $\mathsf{dth}(a) = b$ and $\mathsf{low}(p) = a$, in which p is any node in the banana other than b. For convenience, this includes $p = a$, which thus stores a pointer to itself. To obtain the lower end of a banana from its upper end we define $\mathrm{Bth}(b) = \mathsf{low}(\mathsf{in}(b)) = \mathsf{low}(\mathsf{mid}(b))$. Observe that each of the two banana trees stores exactly one node per critical item and thus requires $O(n)$ space. The dictionaries and the doubly-linked list require additional $O(m)$ space and hence the total space used by our data structure is linear in the total number of items.

We conclude this section with the assertion that the banana tree of a linear list is unique. This will be useful in proving some of the algorithms in the subsequent section correct.

Lemma 3.4.2 (Uniqueness of Banana Tree)**.** *Given a linear list of distinct values, there is a unique path-decomposed binary tree satisfying Conditions* I.1, I.2, *and* II, *and a resulting unique banana tree satisfying Conditions* III.1 *and* III.2.

Proof. The algorithm that constructs the banana tree of a linear list is deterministic and thus computes a unique such tree, which we denote B. It therefore suffices to prove that no other banana tree satisfies Conditions I.1, I.2. II, III.1, and III.2. We have already seen that there is a unique ordered binary tree that satisfies I.1 and I.2. Similarly, there is a unique decomposition of this tree into paths that satisfies II.

To get a contradiction, assume $B' \neq B$ is another banana tree that satisfies I.1 to III.2. Write B_2 and B'_2 for the path-decomposed ordered binary tree we get by merging the two trails of each banana in B and B', respectively. To satisfy Condition I.2, the merging must preserve the ordering by value, and by Conditions III.1 and III.2 this is indeed possible. But the step from B' to B'_2 is deterministic, and so is the step from B to B_2. We have already established that $B'_2 = B_2$, which implies $B' = B$, as claimed. □

The reverse of Lemma 3.4.2 does not hold for the trivial reason that the banana trees do not store the non-critical items. There is also the more subtle reason that different path-decomposed ordered binary trees can map to the same banana tree. Indeed, the banana trees do not specify the order of values between parallel trails, so merging them can result in different ordered paths.

3.4.3 String and Spine

It is possible to connect the trails of a banana tree into a single curve such that the homological critical points are listed from left to right according to their positions in the interval. Considering a banana tree as a graph, the minima and the special root have two neighbors, and the maxima have four neighbors each, so the maxima would appear twice, but one appearance is the mirror of the maximum, namely the upper end of the in-trail. We list the maximum when we reach the upper end of the mid-trail. Equivalently, we adopt the following rule for a banana spanned by a, b: if $b < a$, we first list b, then walk down the interior nodes of the left trail, then list a, and finally walk up the interior nodes of the right trail, and if $b > a$, we do the same except that we list b at the end rather than at the beginning. The sub-bananas are listed recursively when their upper ends are encountered.

As an example consider the banana tree sketched in Figure 3.4.2 on the left: after starting at the special root, we first encounter the left hook, then c, d, e and so on until n, o, and finally the right hook before returning to the special root. We call this the *string* of the banana tree. It is not difficult to see that this is also the in-order traversal of the tree after the first step of the banana tree construction.

We continue with the definition of an important subset of nodes in a banana tree. The *left spine* consists of the special root, β, as well as the first interior node of every left trail with upper end in the left spine. Symmetrically, the *right spine* consists of β as well as the first interior node of every right trail with upper end in the right spine. The two overlap in β, and the *spine* is the union of the left and the right spines; see Figure 3.4.1 for an example. The nodes in the spine can also be characterized in terms of the windows they span.

Lemma 3.4.3 (Spines and Windows)**.** *Let $\mathrm{Up}(f)$ be the banana tree of the map f. Then*

(i) the special root β and the global minimum span the unique global window of f;

(ii) a node $b \neq \beta$ of the left spine spans a window with wave that is short on the left;

(iii) a node $b \neq \beta$ of the right spine spans a window with wave that is short on the right;

(iv) all other internal nodes of $\mathrm{Up}(f)$ span windows with simple waves.

Proof. It is not necessary to give an argument for (i). Since (ii) and (iii) are symmetric, it suffices to prove (ii). By the first step of the construction of a banana tree in Section 3.4.2, a node b belongs to the left spine iff $f(b) > f(q)$ for all items $q < b$. It follows that the window spanned by b is not constrained on the left, so it extends to the beginning of the list. In other words, the window is with short wave, and the wave is short on the left.

The condition $f(b) > f(q)$ for all $q < b$ is also necessary to have a short wave on the left, so all such windows are spanned by nodes in the left spine. Symmetrically, all windows with short wave on the right are spanned by nodes in the right spine. Hence, all other internal nodes span windows with simple wave, which proves (iv). □

According Lemma 3.4.3, the special root spans the global window, the remaining nodes in the spine span windows with short wave, and the remaining internal nodes span windows with simple wave. This holds for $\mathrm{Up}(f)$ as well as for $\mathrm{Up}(-f) = \mathrm{Dn}(f)$. Since a window with short wave of f is not a window of $-f$, and vice versa [25, Theorem 4.2], this implies that the min-max pairs with the maximum in the spine (other than the special root) are distinct in the two trees. In contrast, the min-max pairs with the maximum not in the spine are the same in $\mathrm{Up}(f)$ and $\mathrm{Dn}(f)$.

The algorithms for splitting and gluing banana trees in Section 3.5.3 need to recognize nodes in the spine. We thus label these nodes and maintain the labeling when changes occur.

3.5 Algorithms

Call the banana trees of a map and its negative the *up-tree* and *down-tree* of f, denoted $\mathrm{Up}(f)$ and $\mathrm{Dn}(f)$. We describe the construction of both trees, the extraction of the augmented persistence diagram from these trees, and operations that maintain the trees subject to local changes of the data. We begin with the construction of the up-tree, which takes time linear in the number of items. Together with the extraction, this gives a linear-time algorithm for the augmented persistence diagram; compare with the algorithm of Glisse [26], which constructs the persistence diagram in linear time but not the augmentation. We prove the correctness of the local and topological maintenance operations in Sections 4.1 and 4.2.

3.5.1 Construction

We explain the construction of the up-tree for a generic piecewise linear map defined by a list of m items: $f(i) = c_i$ for $1 \leq i \leq m$. For convenience, we add items 0 and $m+1$ at the two ends, with $f(0) = \infty$ and $f(m+1) = \infty - 1$. The algorithm processes the map from left to right and uses a stack to store a subset of the minima and maxima so far encountered. Specifically, while processing j, the stack stores pairs $(a_0, b_0), (a_1, b_1), \ldots, (a_k, b_k)$ such that

- $0 = a_0 = b_0 < a_1 < b_1 < a_2 < \ldots < a_k < b_k < j$,
- $f(i) \leq f(b_\ell)$ for all $1 \leq \ell \leq k$ and $b_{\ell-1} < i < j$,
- $f(i) \geq f(a_\ell)$ for all $1 \leq \ell \leq k$ and $b_{\ell-1} < i \leq b_\ell$.

The first two properties imply that the points $(b_\ell, f(b_\ell))$ form a descending staircase in the plane, and all points $(i, f(i))$ with $i < j$ that are not steps of the staircase lie below the staircase. The third property says that item a_ℓ minimizes the value among all items i vertically below the step of b_ℓ.

The only relevant items are the critical points, so we eliminate non-critical items in a preliminary scan and connect the remaining items with prv- and nxt-pointers. Here we consider the artificially added items as maxima, so 0 is the first item in this list, followed by $\mathsf{nxt}(0)$, $\mathsf{nxt}(\mathsf{nxt}(0))$, etc., until we reach item $m+1$. For later use, each remaining item in the list is provided with the appropriate subset of initially `null` in-, mid-, up-, dn-, low-, and dth-pointers. During the main scan the algorithm maintains unfinished bananas, each corresponding to a pair on the stack, as well as a value A, which, after the current item has been processed, is the item with the minimum value to the right of the top-most item on the stack. Whenever we pass a minimum, j, we set $A = j$, as the immediately preceding item must have been a maximum. Whenever we pass a maximum, the stack is accessed through the standard functions Push and Pop, as well as through function Top, which returns the item b at the top of the stack, but without removing the pair (a, b) from the stack.

Suppose (a, b) is the pair at the top of the stack, and we pop it off because $f(b) < f(j)$. If $f(A) < f(a)$, we now know that the unfinished banana spanned by a, b is correct, so we finalize it in FixBanana. Otherwise, A, b span a banana, which must belong to a right trail as $b < A$. We, thus, detach b on the left,

attach it on the right, and finalize the banana in FixBanana. Then we set $A = a$ and push (A, j) on the stack after creating its temporary banana and attaching it on the left:

```
dn(0) = 1; Push(a_0 = 0, b_0 = 0); j = 0;
repeat j = nxt(j);
        if j is minimum then A = j endif;
        if j is maximum then
           while f(j) > f(Top) do (a, b) = Pop;
                 if f(A) < f(a) then FixBanana(a, b)
                    else attach b below j on the right;
                         FixBanana(A, b); A = a
                 endif
           endwhile;
           attach j below b on the left; Push(A, j);
           if j = m + 1 then FixBanana(A, j) endif
        endif
until j = m + 1.
```

To "attach j below b on the left", where j is the currently processed item and b is a past item, we create an unfinished banana with upper end j and temporary in- and mid-pointers, and set the relevant pointers of j, as illustrated in Figure 3.5.1 on the left:

$$\mathsf{up}(j) = b;\ \mathsf{in}(j) = \mathsf{dn}(b);\ \mathsf{mid}(j) = \mathsf{prv}(j);\ \mathsf{dn}(j) = \mathsf{nxt}(j);$$
$$\mathsf{dn}(b) = \mathsf{up}(\mathsf{in}(j)) = \mathsf{up}(\mathsf{mid}(j)) = j.$$

Whenever the pair containing j is later popped off the stack, and a comparison of the values shows that j belongs to a left trail, then these pointers remain unchanged in the final banana.

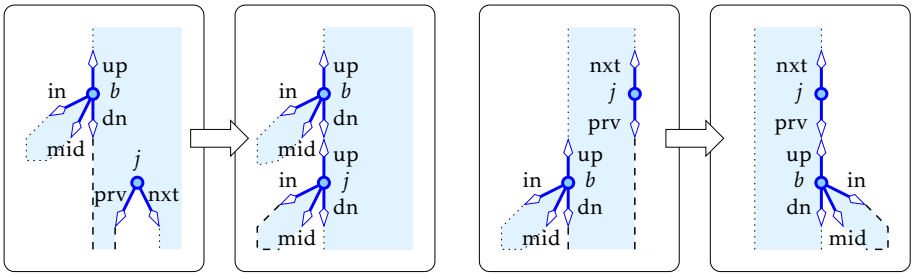

Figure 3.5.1: *Left:* item j it temporarily attached on the left, which creates a banana whose upper end, j, is interior to a left trail. *Right:* the temporary attachment of item b on the left is resolved, and b is attached on the right, which creates a banana whose upper end, b, is interior to a right trail.

If, however, this is not the case, we execute "attach b below j on the right", which suitably updates these values to reflect the fact that j belongs to a right trail. Specifically, we detach b on the left and attach it on the right, as illustrated in Figure 3.5.1 on the right:

$$\mathsf{dn}(\mathsf{up}(b)) = \mathsf{in}(b);\ \mathsf{up}(\mathsf{in}(b)) = \mathsf{up}(b);$$
$$\mathsf{up}(b) = j;\ \mathsf{in}(b) = \mathsf{prv}(j);\ \mathrm{aux} = \mathsf{dn}(b);\ \mathsf{dn}(b) = \mathsf{mid}(b);\ \mathsf{mid}(b) = \mathrm{aux};$$
$$\mathsf{prv}(j) = \mathsf{up}(\mathsf{in}(b)) = b.$$

To fix a banana, we set the low-pointers of its interior nodes and the minimum, as well as the in-, mid-, and dth-pointers of the minimum:

```
function FixBanana(a, b):
  q = b; p = in(b);
  while p ≠ a do low(p) = a; q = p; p = dn(p) endwhile;
  in(a) = q;
  q = b; p = mid(b);
  while p ≠ a do low(p) = a; q = p; p = dn(p) endwhile;
  mid(a) = q;
  low(a) = a; dth(a) = b.
```

To argue the correctness of the algorithm, we formulate three invariants maintained by the algorithm. Let j denote the current item, and distinguish between *past items*, $b < j$, and *future items*, $b > j$.

(i) The prv-pointers of the future items are unchanged, except possibly the first one, which points to j. In contrast, all nxt-pointers remain unchanged throughout.

(ii) If a past item, b, is on the stack, then the banana rooted at b is temporary and unfinished. The latter means that the in- and mid-pointers of b and the up- and dn-pointers of b and all interior nodes are in place, but not necessarily the remaining pointers that define the banana. In contrast, all its sub-bananas are complete.

(iii) Right before taking b off the stack, it satisfies (ii) except that the banana rooted at b is no longer temporary. After taking b off the stack, its banana is complete, which means that all pointers of its nodes are in place.

It is easy to see that (i) is maintained, as there is only one place where a prv-pointer is altered, and there is no place where a nxt-pointer is altered. Invariant (ii) holds because all interior nodes of the banana rooted at b have been taken off the stack in the past, and their bananas are complete by Invariant (iii). In particular, this implies that each unfinished banana spanned by a, b has a path from $\mathsf{in}(b)$ along dn-pointers to a defining its in-trail, and similarly from $\mathsf{mid}(b)$ along dn-pointers to a defining its mid-trail. Assuming (ii), function FixBanana adds the necessary pointers to complete the banana, which implies (iii).

EXTRACTION. We next discuss how to compute the augmented persistence diagram from the up- and the down-tree of f. Each banana in $\mathrm{Up}(f)$ corresponds to a point in $\mathrm{Ord}(f)$, with the exception of the banana of the special root, which corresponds to a point in $\mathrm{Ess}(f)$. We find these points and the nesting relation by recursively walking along the trails from bottom to top. We are handed the upper end of each pair of trails, so first we jump to the lower end. The recursive function that enumerates all points and arrows is called with two parameters: the upper end of two parallel trails, and the point in the persistence diagram that corresponds to the pair of trails that contain that upper end as an interior node. Initially, the upper end is the special root, and the point is empty:

function WALK(b, pnt):
  $a = \text{Bth}(b)$;
  output $\text{Point}(a,b) = (f(a), f(b))$ and $\text{Arrow}(a,b) = (\text{Point}(a,b), pnt)$;
  $x = \text{in}(a)$; while $x \neq b$ do WALK$(x, \text{Point}(a,b))$; $x = \text{up}(x)$ endwhile;
  $x = \text{mid}(a)$; while $x \neq b$ do WALK$(x, \text{Point}(a,b))$; $x = \text{up}(x)$ endwhile.

To complete the augmented persistence diagram, we also construct $\text{Dn}(f)$ and apply the recursive function to its parallel trails, with the only difference that the banana of the special root does not correspond to any point in $\text{Dgm}(f)$.

SUMMARY. After removing all non-critical items, which takes $O(m)$ time, the construction of the two banana trees as well as the extraction of the augmented persistence diagram takes only $O(n)$ time. Indeed, the main scan of the construction algorithm completes each banana only once, and altogether touches each item only a constant number of times. We summarize our findings for later reference.

Theorem 3.5.1 (Time to Construct)**.** *Let $f : [0, m+1] \to \mathbb{R}$ be a generic piecewise linear map with n maxima. After removing all non-critical items in $O(m)$ time, $Up(f)$ and $Dn(f)$ can be constructed in $O(n)$ time, and the augmented persistence diagram of f can be computed from these trees in $O(n)$ time.*

3.5.2 Local Maintenance

Recall that the banana trees store all critical items but none of the non-critical items. As a temporary exception, we *delete* items by first adjusting their values until they become non-critical. To formalize the operation that *adjusts* the value of an item, j, from c_j to d_j, we write $f, g : [0, m+1] \to \mathbb{R}$ for the maps before and after adjustment, so $g(i) = f(i)$ for $0 \leq i \neq j \leq m+1$, and $f(j) = c_j$, $g(j) = d_j$. To avoid complications near the endpoints, assume $3 \leq j \leq m-2$. We give details on how to treat endpoints in Section 4.1.6. We modify the trees while following the *straight-line homotopy* from f to g, which is the 1-parameter family of maps $h_\lambda : [0, m+1] \to \mathbb{R}$ defined by $h_\lambda(x) = (1-\lambda)f(x) + \lambda g(x)$ for $0 \leq \lambda \leq 1$. Clearly, $h_0 = f$ and $h_1 = g$.

The homotopy reduces to a sequence of *interchanges*—of two maxima or two minima—followed or preceded by a *cancellation* or an *anti-cancellation*, which reflect the disappearance or appearance of a point into or from the diagonal of the persistence diagram, or a *slide*, which occurs when a critical item becomes non-critical due to a non-critical neighbor becoming critical. We will discuss these operations in detail below. Among the adjustments, we consider two scenarios:

A: item j is non-critical in f and increases its value;

B: item j decreases its value until it becomes non-critical in g.

The scenario in which j is non-critical and decreases its value is symmetric to A, and either of the two is applied after we insert a new item. The scenario in which j increases its value until it becomes non-critical is symmetric to B, and either of the two is needed before we delete an item. Other adjustments are subsequences or compositions of Scenarios A and B or their symmetric versions.

SCENARIO A. We increase the value of a non-critical item, j, and we assume that it becomes critical, else up-tree and down-tree would not be affected. We consider the case in which $f(j-1) < f(j) < f(j+1)$ and $f(j-1) < g(j) > f(j+1)$, i.e., the item j becomes a maximum in g. If $j+1$ is non-critical in f, it becomes a minimum in g and the number of critical items increases by two. If $j+1$ is critical (a maximum), then it becomes non-critical in g and item j replaces it as maximum; the number of critical items does not change. In the former case we perform an anti-cancellation to introduce the banana spanned by $j+1$ and j; in the latter case we perform a slide. Afterwards we fix the position of j in the up-tree and down-tree by performing a sequence of interchanges:

```
if g(j) > f(j+1) then
  if f(j+1) < f(j+2) then anti-cancel j and j+1 in Up(f) and Dn(f)
                     else slide j+1 to j in Up(f) and Dn(f)
  endif;
  set q = up(j) in Up(f);
  while f(q) < g(j) do interchange j and q in Up(f) and Dn(f);
                       q = up(j) in Up(f)
  endwhile
endif.
```

Note that we iterate with $q = \mathsf{up}(j)$ and not with $q = \mathsf{up}(q)$. This is not a mistake because the interchange in $\mathrm{Up}(f)$ is of two maxima, j and q, which swaps the two nodes (see below). In contrast, the interchange in $\mathrm{Dn}(f)$ is of two minima, albeit they are the same two items, j and q. Performing these interchanges simultaneously, we save the effort of independently finding the next relevant interchange of minima, which would be costly. The correctness of this strategy is guaranteed by Lemma 3.5.2, which we will state and prove after formalizing the notion of an interchange.

SCENARIO B. The symmetric version of Scenario A—in which j decreases its value—is implemented accordingly, by switching the roles of the up-tree and the down-tree. We use the simultaneous interchange of maxima and minima also in the implementation of this inverse of Scenario A. There are again symmetric cases, and we consider the case in which $f(j-1) < f(j+1)$ and $f(j-1) < f(j) > f(j+1)$. Furthermore, we assume that item j is interior to a left trail. The operation begins with a sequence of interchanges of maxima in the up-tree and of minima in the down-tree, followed by a cancellation or by a slide:

```
loop q = arg max{f(dn(j)), f(in(j)), f(mid(j))} in Up(f);
  if f(q) > f(j+1) then interchange q and j in Up(f) and Dn(f)
                   else exit endif
forever;
if f(j+1) < f(j+2) then cancel j with j+1 in Up(f) and Dn(f)
                   else slide j to j+1 in Up(f) and Dn(f)
endif.
```

Note that q is either a maximum or its value is less than $f(j+1)$ as $f(j-1) < f(j+1)$, so we will not attempt to interchange a maximum, j, with a minimum, q, which would be impossible indeed. We continue with the details for the interchanges and (anti-)cancellations.

INTERCHANGE OF MAXIMA. An interchange between two maxima with $f(j) < f(q)$ has structural consequences only if $q = \mathsf{up}(j)$, as this is the only case in which a uniqueness condition, namely III.1 or III.2, is violated. We will see that the algorithm does not ever get into the situation of attempting any other interchange. Assume that q is on a left trail, which is the situation depicted in Figure 3.5.2. Let i and p be such that $j = \mathsf{dth}(i)$ and $q = \mathsf{dth}(p)$. If $j = \mathsf{dn}(q)$, we distinguish two cases depending on the order of $f(i)$ and $f(p)$.

CASE 1: $f(i) < f(p)$. Remove q from its trail and add it right below j in j's in-trail, as illustrated in Figure 3.5.2 in the top left. Adjust the pointers of the involved nodes accordingly.

CASE 2: $f(i) > f(p)$. Exchange j and q as upper ends of their respective bananas, remove q from its trail and add it right below j in j's mid-trail, as illustrated in Figure 3.5.2 in the top right. Adjust pointers, and in particular set $\mathsf{dth}(i) = q$ and $\mathsf{dth}(p) = j$. The in-trail of i becomes its mid-trail and vice versa.

In both cases, j joins the left spine of the up-tree iff q is a node of the left spine already before the operation. The cases where $j = \mathsf{mid}(q)$ or $j = \mathsf{in}(q)$ are similar to the reverse of the cases with $j = \mathsf{dn}(q)$, and are illustrated in the bottom row of Figure 3.5.2. There are also the inverse operations (reading Figure 3.5.2 from right to left), which apply when $j = \mathsf{up}(q)$, i.e., the case $f(j) > f(q)$, and the symmetric cases where q is on a right trail.

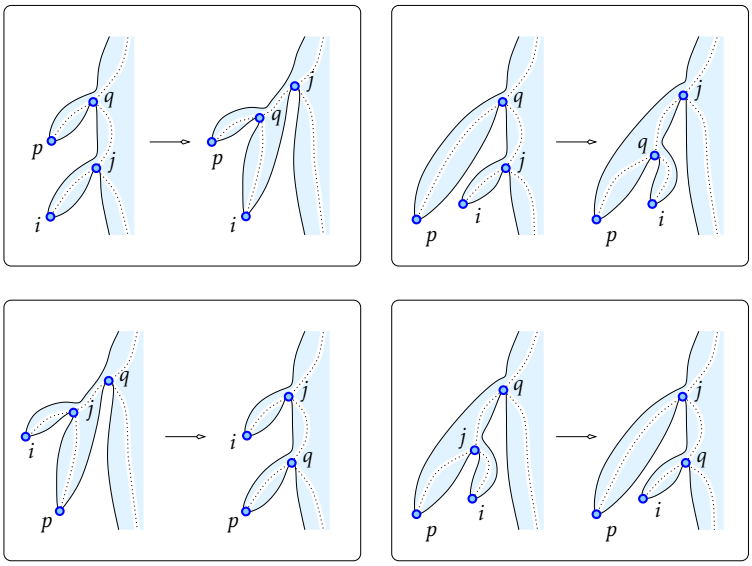

Figure 3.5.2: The interchange of two maxima, j and q. In the *dotted* underlying tree, the operation corresponds to a rotation. Before the interchange, the pairs are i, j and p, q. *Top left:* $f(i) < f(p)$ which preserves the pairs. *Top right:* $f(i) > f(p)$ and the pairs change to i, q and p, j. *Bottom left:* $j = \mathsf{in}(q)$ which preserves the pairs. *Bottom right:* $j = \mathsf{mid}(q)$ and the pairs change to i, q and p, j.

INTERCHANGE OF MINIMA. An interchange of maxima in $\mathrm{Up}(f)$ is always done in parallel with an interchange of minima in $\mathrm{Dn}(f)$. It can, however, happen that the interchange of j and q has a structural effect on $\mathrm{Up}(f)$ but not on $\mathrm{Dn}(f)$. An example is the interchange of nodes i and g in Figure 3.4.1 and

3.4.2. They are internal nodes in the up-tree, so the effect of the interchange is as depicted in Figure 3.5.2 on the left. The two nodes are leaves in the down-tree whose bananas do not meet, so the interchange of minima has no effect.

In general, the interchange of two minima with $f(j) < f(q)$ has structural consequences only if $p = \mathsf{dth}(q)$ is interior to the banana spanned by j and $i = \mathsf{dth}(j)$, as this is the only case in which $g(j) > g(q)$ leads to a violation of the uniqueness conditions, namely of II. To describe how the trails are updated, we assume that p is interior to the left trail. We split this trail into the upper part above p and the lower part below p (with neither part including p). Similarly, we split the parallel right trail connecting i to j into upper and lower, with values larger and smaller than $f(p)$, respectively. Then we concatenate the upper part of the left trail with the left trail connecting p to q (but without the upper end, which is p), and we concatenate the upper part of the right trail with the right trail connecting p to q (this time including p). Finally, we join the two lower parts to form parallel trails connecting p to j.

Observe that splitting the left trail is easy because it contains p as an interior node. We split the right trail by traversing it one node at a time from the upper end until we reach the first node with value less than $f(p)$. Note that all the travsersed node on the right trail will have a change in their low-pointer and we will use this fact when we analyze the running-time of the operation at the end of this subsection.

SIMULTANEITY OF INTERCHANGES. Next, we prove the correctness of coupling interchanges as described in Scenarios A and B. The moment two maxima of f interchange is of course also the moment at which they interchange as minima of $-f$. However, many interchanges are irrelevant, in the sense that they cause no structural changes to the banana trees. Many of these irrelevant interchanges go unnoticed by our algorithm (and fortunately so), but it is important that no relevant interchange is overlooked. We claim that every relevant interchange of minima corresponds to a relevant interchange of maxima.

To formalize this claim, let $F\colon [0, m+1] \to \mathbb{R}$ be a piecewise linear map determined by its values at the integers, and assume that these values are distinct, with the exception of $F(j) = F(q)$ at maxima $j \neq q$ of F. Let $f, g\colon [0, m+1] \to \mathbb{R}$ be piecewise linear maps defined by the same values at the integers, except that $f(j) = F(j) - \varepsilon$ and $g(j) = F(j) + \varepsilon$ for a sufficiently small $\varepsilon > 0$. The straight-line homotopy from f to g is an interchange of maxima, namely of j and q, and that from $-f$ to $-g$ is an interchange of minima, again of j and q. We call the former *relevant* if $\mathrm{Up}(f)$ and $\mathrm{Up}(g)$ differ by a rotation, namely of j and q, and we call the latter *relevant* if $\mathrm{Dn}(f)$ and $\mathrm{Dn}(g)$ differ by a change in the pairing.

Lemma 3.5.2 (Coupling of Interchanges)**.** *Let $f, g\colon [0, m+1] \to \mathbb{R}$ as introduced above. If the straight-line homotopy from $-f$ to $-g$, which is an interchange of minima, is relevant, then so is the straight-line homotopy from f to g, which is an interchange of maxima.*

Proof. We prove the contrapositive: that irrelevant interchanges of maxima imply irrelevant interchanges of minima. The interchange of the maxima

$j \neq q$ of f is irrelevant if there is a banana in $\mathrm{Up}(f)$ so that j and q belong to opposite trails, or to sub-bananas rooted on opposite trails of this banana. Let this banana be spanned by a, b. Assuming $j < q$, this implies $j < a < q$ and b is either to the left or the right of the three items. Letting i and p be the lower ends of the bananas spanned by j and q, respectively, we observe that $i < a < p$, $f(a) < f(i)$, and $f(a) < f(p)$.

For the negated function, j, q are minima and i, a, p are maxima satisfying $i, j < a < p, q$, $-f(a) > -f(i)$, and $-f(a) > -f(p)$. If $W(i, j)$ and $W(p, q)$ are both with simple waves, then $W(j, i)$ and $W(q, p)$ are triple-panel windows of $-f$ that are separated by a. It follows that the interchange of minima is irrelevant. Similarly, if one or both of these windows are with short wave, then the separation by a implies that the pairing stays constant, so again the interchange is irrelevant. □

There is a special case to consider when the separating banana is spanned by the special root, and j, q are the maxima with the two largest values. For f, we have $\beta = q$ and for g we have $\beta = j$. In words, the interchange of the maxima j and q does not change the pairing but it causes a replacement of the special root.

CANCELLATIONS. The last step in Scenario B is the cancellation of items j and $j + 1$, which is implemented by removing the two nodes from the up-tree and down-tree. We argue that the implementation is really this easy.

Recall that the value of j during the homotopy from f to g is $h_\lambda(j) = (1-\lambda)f(j)+\lambda g(j)$. By assumption, $f(j-1) < f(j+1)$, so the cancellation happens at $f(j-1) < g(j) < f(j+1)$. Since $f(j) > f(j+1) > g(j)$ and $f(j+2) > f(j+1)$, there exists $0 \leq \mu \leq 1$ such that $f(j+2) > h_\mu(j) > f(j+1)$. At this stage of the homotopy, $j+1$ is a child of j and $\mathsf{dth}(j+1) = j$ in $\mathrm{Up}(f)$. In other words, $j+1$ and j are the upper and lower ends of a pair of empty trails. We can therefore simply remove this pair of trails, while forming a direct link between $\mathsf{dn}(j)$ and $\mathsf{up}(j)$. The situation is symmetric in $\mathrm{Dn}(f)$.

ANTI-CANCELLATIONS. The first step in Scenario A is the anti-cancellation of items j and $j+1$. We describe how to perform the anti-cancellation in the up-tree; the anti-cancellation in the down-tree is symmetric. By assumption, $f(j-1) < g(j) > f(j+1) < f(j+2)$ and $g(j)$ is such that there is no item with value between $f(j+1)$ and $g(j)$. We first identify the maximum b closest to j such that the new minimum $j+1$ lies between j and b, i.e., $j < j+1 < b$. Note that there can be no other critical point between $j+1$ and b.

To insert j, we walk along the path from b to the closest minimum a with $a < j < j+1 < b$, until we find a node q with smaller value than j. The node j is then inserted as a parent of q:

```
if Bth(b) < j < b then q = mid(b) else q = dn(b) endif;
while f(q) > g(j) do q = in(q) endwhile;
if q is a leaf then if q = Bth(b) then insert j as mid(q)
                                  else if q is the global minimum
                                          and q < j then
                                              insert j as mid(q)
                                  else insert j as in(q)
                    endif
               else insert j as up(q)
endif.
```

After inserting j, we construct a banana spanned by $j+1$ and j, which includes setting $\mathsf{in}(j+1) = \mathsf{mid}(j+1) = \mathsf{dth}(j+1) = j$ and $\mathsf{in}(j) = \mathsf{mid}(j) = j+1$.

SLIDES. Consider the case in which the value of a maximum, j, is decreased, and assume that $f(j-1) < f(j+1) > f(j+2)$. Note that $j+2$ is non-critical. If $f(j-1) < g(j) < f(j+1)$, then $j+1$ becomes a maximum and j becomes non-critical. The number of critical items remains the same in f and g, but criticality is transferred from one item in f to a neighboring item in g. As mentioned earlier, this is what we call a *slide*.

The same situation occurs when (1) the value of a minimum increases above the value of a non-critical neighbor, (2) the value of a non-critical item increases above the value of a neighboring maximum, or (3) the value of a non-critical item decreases below the value of a neighboring minimum. A slide has no impact on the structure of the banana-tree and only requires to update the association between items and nodes.

SUMMARY. We summarize the findings in this subsection by stating the running-time for adjusting the value of an item in terms of the number of critical points and the number of changes caused to the augmented persistence diagram. To this end, we define the *accumulated change* along a straight-line homotopy as the total number of changes to the points and arrows in the augmented persistence diagram that occur during the homotopy.

Theorem 3.5.3 (Time to Adjust)**.** *Let $f : [0, m+1] \to \mathbb{R}$ be a generic piecewise linear map with n maxima. The time to adjust the value of an item is $O(\log n + k)$, in which k is the accumulated change during the adjustment, plus $O(k')$ if the adjustment includes an anti-cancellation, in which k' is the difference in the transitive closures of the nesting relation before and after the anti-cancellation.*

Proof. We prove the claimed bound on the running-time by charging most steps to the change in augmented persistence diagrams they cause.

An *interchange of maxima* reduces to a rotation plus possibly a swap in the pairing. The rotation takes $O(1)$ time, which we charge to the changing arrow, and the swap takes $O(1)$ time, which we charge to the two points in the diagram that exchange coordinates. An *interchange of minima* reduces to a swap in the pairing followed by resetting the low-pointers along the path connecting the two maxima involved in the swap. As before, the swap is charged to the two points that exchange coordinates, and each resetting of a low-pointer is charged to the corresponding change in the arrow. It is also possible that the interchange of minima has no effect on the binary tree, namely when the corresponding paths are disjoint. This is detected in $O(1)$

time, and this time is charged to the changes caused by the simultaneous interchange of maxima in the other banana tree. A *cancellation* takes $O(1)$ time, and there are at most two cancellations per value adjustment of an item. In contrast, an *anti-cancellation* takes $O(\log n)$ time to find the maximum, b, closest to the new pair of critical items. The while-loop in the anti-cancellation takes as many iterations as there are nodes on the path from b to the newly inserted node. For each of those nodes an arrow appears in the transitive closure of the nesting relation. Thus, the time to insert the new banana is $O(k')$. The insertion itself takes constant time.

In addition, we take $O(\log n)$ time to update the dictionaries whenever an item changes from critical to non-critical, or the other way round, and there are only $O(1)$ such changes per value adjustment of an item. All this adds up to $O(\log n + k)$ time, if the adjustment does not requires any anti-cancellation, and to $O(\log n + k + k')$ time, if it does. □

3.5.3 Topological Maintenance

Given a collection of linear lists, we next study the maintenance of the augmented persistence diagram subject to cutting and concatenating the lists. Recall that a list $c_1, c_2, \ldots, c_m$ induces a piecewise linear map, $f : [0, m+1] \to \mathbb{R}$, with $f(i) = c_i$ for $1 \leq i \leq m$. The values at $i = 0, m+1$ are added to create the computationally convenient hooks introduced in Section 3.4. To *cut* f, we split the list into $c_1, c_2, \ldots, c_\ell$ and $c_{\ell+1}, c_{\ell+2}, \ldots, c_m$ and let $g : [0, \ell+1] \to \mathbb{R}$ and $h : [\ell, m+1] \to \mathbb{R}$ be the corresponding piecewise linear maps. We need at least two items to construct the hooks, so we require $2 \leq \ell \leq m-1$. We describe the operation for the up-tree, and consider the down-tree only to the extent it provides information to update the up-tree. For ease of reference, we write x for the midpoint between ℓ and $\ell+1$ and set $f(x) = \frac{1}{2}(f(\ell) + f(\ell+1))$.

SPLITTING A BANANA TREE. We introduce terminology before describing the splitting algorithm in three steps. A banana suffers an *injury*, *fatality*, *scare* if x cuts through the in-, mid-, out-panel of the window, respectively; see Figure 3.5.3. A triple-panel window with simple wave is shared by f and $-f$, except that in- and out-panels are exchanged. Hence, a scare in $\mathrm{Up}(f)$ is an injury in $\mathrm{Dn}(f)$, so we exploit the down-tree to find the scares in the up-tree.

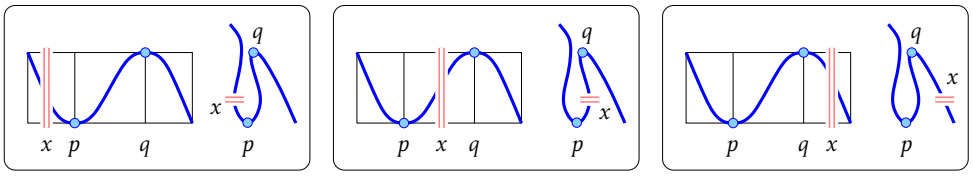

Figure 3.5.3: *From left to right:* an injury, fatality, and scare. Correspondingly, x cuts the banana in its in-trail, in its mid-trail, or after the maximum but still above the value of the minimum.

STEP 1: **Find smallest banana.** To enumerate the bananas that suffer injuries or fatalities, we first find the smallest such banana. To this end, we search the dictionaries to locate the smallest interval between two critical points that satisfy $a < x < b$. Note however that neither b is necessarily the upper end nor a is necessarily the lower end of this banana. Assuming b is a maximum, we use it to find the smallest banana in the up-tree such that x lies between

its lower and upper ends. The search uses a pair of nodes, q, r, maintaining that the upper end of the desired smallest banana has an ancestor that is an interior node of the right trail with upper end q, and r with $f(r) > f(x)$ is a node on this right trail. The iteration advances q and r toward x and halts when these conditions fail. To repeatedly go down the right trail of r, which itself is interior to a right trail, we use the $\mathsf{in}(r)$ pointer. The iteration is slightly different in the first case, when b is interior to a right trail, and in the second case, when b is interior to a left trail:

```
function SmallestBanana(x):
  find a < x < b and assume that b is a maximum;
  if dn(b) < b then q = dth(low(b)); r = dn(b) else q = b; r = mid(b) endif;
  while r ≠ a and f(x) < f(r) do q = r; r = in(r) endwhile;
  return (Bth(q), q).
```

The time is $O(\log n + k)$, in which k is the number of inspected bananas. Each of these bananas causes a change in the augmented persistence diagram, which pays for the visit.

Step 2: **Stack bananas.** After locating the smallest banana in $\mathrm{Up}(f)$ that suffers an injury or fatality, we find the others by traversing the tree upward. In the process, we load three initially empty stacks with the injuries and fatalities. In doing so, we distinguish spanning min-max pair to the left of x, separated by x, to the right of x, and denote the corresponding stacks L_{up}, M_{up}, R_{up}, respectively. On its way up, the algorithm pushes each banana on one of the three stacks, until it encounters the first banana with maximum in the spine.

```
function LoadStacks (x):
  (p, q) = SmallestBanana(x);
  loop case p < x and q < x: Push(L_up, (p, q));
            p < x xor q < x: Push(M_up, (p, q));
            p > x and q > x: Push(R_up, (p, q))
       endcase;
       if q is in spine of Up(f) then exit endif;
       p = low(q); q = dth(p)
  forever.
```

Similarly, we load the initially empty stacks L_{dn}, M_{dn}, R_{dn} with bananas in $\mathrm{Dn}(f)$. When we split the up-tree, it will be convenient to reverse the pairs defining the bananas in the down-tree, and when we split the down-tree, we reverse the pairs defining the bananas in the up-tree. We call a stack with bananas (p_0, q_0) at the bottom to (p_j, q_j) at the top *sorted* if $f(p_j) < \ldots < f(p_0) < f(q_0) < \ldots < f(q_j)$. The stacks satisfy the following properties:

Lemma 3.5.4 (Sorted Stacks)**.** *Let $c_1, c_2, \ldots, c_m$ be a sequence of distinct values, f the thus defined piecewise linear map, x such that $c_2 < x < c_{m-1}$, and L_{up}, M_{up}, R_{up}, L_{dn}, M_{dn}, R_{dn} the stacks as returned by* LoadStacks *holding the bananas with injury, fatality, and scare in $\mathrm{Up}(f)$ and $\mathrm{Dn}(f)$. Then*

(i) all six stacks are sorted;

(ii) all bananas on the stacks are with simple wave, except for the last banana pushed onto $L_{\mathrm{up}}, M_{\mathrm{up}}, R_{\mathrm{up}}$, whose maximum belongs to the spine of $\mathrm{Up}(f)$, and the last banana pushed onto $L_{\mathrm{dn}}, M_{\mathrm{dn}}, R_{\mathrm{dn}}$, whose maximum belongs to the spine of $\mathrm{Dn}(f)$;

(iii) *the set of simple wave bananas on* M_{up} *is the same as that on* M_{dn}*;*

(iv) $L_{\text{up}}, M_{\text{up}}, L_{\text{dn}}$ *can be merged into a single sorted stack, and so can* $R_{\text{up}}, M_{\text{up}}, R_{\text{dn}}$ *and* $L_{\text{up}}, M_{\text{up}}, R_{\text{up}}$ *and* $L_{\text{dn}}, M_{\text{dn}}, R_{\text{dn}}$.

Proof. Property (i) is implied by Condition II on the path-decomposition and the fact that the algorithm visits the bananas from bottom to top. Property (ii) holds because all bananas are with simple wave, other than the ones with maxima in the spine, and except for the respective last ones, no banana pushed onto the stacks have their maxima in the spine. To see Property (iii) recall that a simple wave of f is also a simple wave of $-f$. Hence, simple wave bananas belong to both trees, except that they share the mid-trail but not the in-trail. Each banana on M_{up} and M_{dn} suffers a fatality, which cuts cuts the mid-trail and therefore also the corresponding banana on the other stack. Property (iv) follows from the fact that all bananas on one of these stacks must correspond to nested windows. Note, however, that it is not true that all six stacks can be merged to a sorted stack. The reason is that triple-panel windows of bananas in L_{up} and R_{dn} can overlap without being nested, and so can triple-panel windows of bananas in L_{dn} and R_{up}; see the windows spanned by f, g and h, i in Figure 3.2.2. However, if x cuts through any two such overlapping and not nested windows, then it also separates the critical points that span one from those that span the other. Such pairs neither exist for $L_{\text{up}}, M_{\text{up}}, L_{\text{dn}}$, nor for any of the other three triplets of stacks. □

There are up to two windows with short wave which need special treatment: First, the window of $-f$ with its maximum on the spine of $\text{Dn}(f)$ and x in its in-panel; second, the window of f with its maximum on the spine of $\text{Up}(f)$ and x in neither its in-panel or mid-panel. The former appears in L_{dn} or R_{dn}, but is not a window of $\text{Up}(f)$ and we ignore it when splitting $\text{Up}(f)$. The latter is not loaded into L_{up}, M_{up} or R_{up}, is not a window in $\text{Dn}(f)$ and is thus not loaded onto any stack. However, it may have x in its out-panel, in which case it suffers a scare and should be pushed onto L_{dn} or R_{dn}. We can identify this case by examining $q' = \text{in}(q_j)$, where (p_j, q_j) is the topmost banana. If $(\text{Bth}(q'), q')$ pushed onto L_{dn} or R_{dn} preserves the properties of Lemma 3.5.4 and $f(\text{Bth}(q')) < f(x) < f(q')$, then indeed x is in the out-panel of this window and we place it on L_{dn} or R_{dn} as appropriate when splitting $\text{Up}(f)$. Otherwise, no window with x in its out-panel is missing from the stacks. Symmetrically, when splitting $\text{Dn}(f)$ we ignore in $L_{\text{up}} \cup R_{\text{up}}$ the window with maximum on the spine of $\text{Up}(f)$ and push the missing window in $\text{Dn}(f)$ onto L_{up} or R_{up}.

After pushing the bananas onto the stacks, we remove them from the top. Write $(p, q) = \text{Top}(L_{\text{up}})$ for the topmost banana on L_{up}, with $(p, q) = (\texttt{nil}, \texttt{nil})$ if L_{up} is empty, and similarly for the other stacks. We set $f(\texttt{nil}) = -\infty$. The overall top banana is the one whose maximum has the largest value and is returned by function TopBanana:

```
function TopBanana:
  Stack = nil; (p,q) = (nil,nil);
  for each S ∈ {L_up, R_up, M_up, L_dn, R_dn} do (p',q') = Top(S);
      if f(q') > f(q) then Stack = S; (p,q) = (p',q') endif
  endfor; return (Stack,(p,q)).
```

Step 3: **Split up-tree.** This operation splits the up-tree into two and in the process adds at most four new nodes: a second special root, a minimum and a

maximum if ℓ and $\ell+1$ are non-critical points prior to the cut, and the up-type endpoint of the two new hooks on both sides of the cut. The algorithm uses the loaded stacks to visit the nodes that need repair from top to bottom, i.e. from the largest to the smallest banana as determined by the stacks. The iteration ends when the stacks are empty:

```
function Split(x):
  LoadStacks(x);
  loop (Stack,(p,q)) = TopBanana;
       if (p,q) = (nil,nil) then exit endif;
       case Stack ∈ {L_up, R_up}: DoInjury(p,q);
            Stack = M_up: DoFatality(p,q);
            Stack ∈ {L_dn, R_dn}: DoScare(p,q)
       endcase; Pop(Stack)
  forever.
```

Finally, we add ℓ and $\ell+1$ as new nodes if they become critical in the process, and we do the final adjustment to the values of the new hooks. In each iteration, we have two banana trees, one on the left and the other on the right. An injury transfers part of one tree to the other, and so does a fatality. The latter also adjusts the pairing between the critical points, while a scare only adjusts the pairing. To initialize this setting, we construct a second banana tree consisting of a single banana with two empty trails connecting its special root with a dummy leaf, α. If we cut the original banana tree on the left spine or the special banana, then this new tree becomes the left tree and we define its special root to be at $-\infty$ along the interval. Otherwise, the new tree becomes the right tree and we define the special root of the original tree to be at $-\infty$, swapping in- and mid-trails of the special banana to satisfy the uniqueness conditions. After splitting is complete, we reset the special roots to $-\infty$ and swap the in- and mid-trails of the special banana as needed.For reasons that will become clear shortly, we set $f(\alpha) = f(p_j) - \varepsilon$, in which p_j, q_j are the nodes that span the top banana on the stacks, and $\varepsilon > 0$ is smaller than the difference between the values of any two items. We observe that the top banana in the first iteration either suffers an injury or a fatality. Indeed, if it suffered a scare, then x would cut through its out-panel and therefore lie between q_j and $\mathsf{low}(q_j)$. But then x cuts through the in- or mid-panel of the banana spanned by $\mathsf{low}(q_j)$ and $\mathsf{dth}(\mathsf{low}(q_j))$, and this banana would have been the top banana on the stacks, which is a contradiction. To see the correctness of the splitting operation, we note that Function Split maintains the following invariants:

(i) both banana trees satisfy the uniqueness conditions I.1, I.2, II, III.1, and III.2;

(ii) any node u that is not on any stack and does not have an ancestor on the stacks is in the left tree, if $u < x$, and in the right tree, if $u > x$.

Invariant (ii) implies that once the stacks are empty, all nodes to the left of x are in the left tree and all nodes to the right of x are in the right tree. By invariant (i) the right and left trees are the unique trees representing the maps g and h. Assuming $x < b$ for the root of the banana tree, the first transfer will be from right to left (as in Figure 3.5.4), so we let the existing banana tree be

the right tree and the banana with the two empty trails the left tree. We are now ready to discuss the actions specific to the injury, fatality, and scare of a banana.

An *injury* occurs when x cuts the in-trail of the banana spanned by p, q. To simplify the discussion, assume $Stack = R_{\text{up}}$, so $x < p < q$, as illustrated in Figure 3.5.3, left, and Figure 3.5.4, middle and left. The case $Stack = L_{\text{up}}$ and $q < p < x$ is symmetric.

The injury causes a possibly empty portion of the in-trail to split off and append to the rightmost banana spanned by a spine node of the tree on the left; see again Figure 3.5.4. The cut off portion consists of all interior nodes $j < x$ and their sub-bananas. Note that it is quite possible that x cuts the string somewhere in a sub-banana with upper end on the in-trail. In this case, the sub-banana does not transfer as its upper end is to the right of x, and it is instead subject to a later repair operation. Because the bananas taken from the stacks are sorted, we have $f(i) > f(\alpha)$ for all transferred nodes i. We maintain α as an up-type endpoint that spans a banana with b, as before the operation.

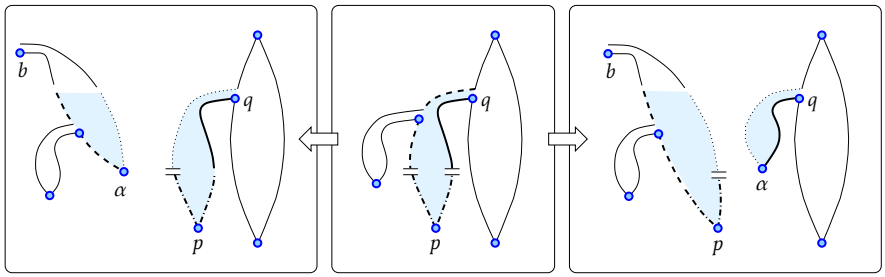

Figure 3.5.4: *Middle:* the banana spanned by p, q suffers an injury (cut in in-trail) or a fatality (cut in mid-trail). *Left:* to process the injury, we move the *dashed* portion and append it to the rightmost spine banana of the left tree. The up-type node, α, is paired with the upper end, b, of the expanded banana. *Right:* to process the fatality, we move the entire in-trail together with the *dash-dotted* portion of the mid-trail. The new up-type endpoint, α, is paired with q in the right tree, and p is paired with the upper end, b, of the expanded banana. Dotted portions of trails are empty.

A *fatality* occurs when x cuts the mid-trail of the banana spanned by p, q. Hence, $Stack = M_{\text{up}}$, and we assume $p < x < q$, which is the case illustrated in Figure 3.5.3, middle, and Figure 3.5.4, middle and right. The case $q < x < p$ is symmetric.

The fatality causes the entire in-trail and a possibly empty portion of the mid-trail to split off and append to the rightmost banana spanned by a spine node of the tree on the left; see again Figure 3.5.4. The transfer includes the minimum, p, and all interior nodes, j, with $j < x$. As in the case of an injury, it is possible that x cuts a sub-banana with upper end on the mid-trail. This sub-banana is not transferred and instead subject to later repair operation. Since p moves from the right to the left, we move α from the left to the right and adjust the pairing accordingly: p spans a banana with b in the left tree, and α spans a banana with q in the right tree. To ensure that the path-decomposition remains correct, we set $f(\alpha) = f(p) - \varepsilon$.

A *scare* occurs when x lies in the out-panel of the triple-panel window spanned by p, q, so x cuts neither trail of the corresponding banana. However, there is a banana spanned by q, p in $\text{Dn}(f)$, and x cuts the in-trail of that

banana. Assume $Stack = L_{\text{dn}}$, so $p < q < x$, which is the case illustrated in Figure 3.5.3, right. The case $Stack = R_{\text{dn}}$ and $x < q < p$ is symmetric.

The scare does not affect the right up-tree, and it affects the left up-tree only indirectly, namely by triggering a change in the pairing. In other words, it preserves the underlying ordered binary tree (First Step of the banana tree construction), but it changes the path-decomposition (Second Step), and therefore also the organization of the bananas (Third Step). This is done by executing an interchange of two minima, namely of p and $\alpha = \mathsf{low}(\mathsf{dth}(p))$. To justify this interchange, we adjust the value of α to $f(\alpha) = f(p) + \varepsilon$.

GLUING TWO BANANA TREES. Concatenating the lists $c_1, c_2, \ldots, c_\ell$ and $c_{\ell+1}, c_{\ell+2}, \ldots, c_m$ is the inverse of cutting $c_1, c_2, \ldots, c_m$ into these two lists. Equivalently, we can think of concatenating $g\colon [0, \ell+1] \to \mathbb{R}$ and $h\colon [\ell, m+1] \to \mathbb{R}$ to get $f = g \cdot h\colon [0, m+1] \to \mathbb{R}$. A triple-panel window with simple wave of g is still a triple-panel window with simple wave of f, and similarly for h and f. Also a triple-panel window whose wave is short at the left end of the domain of g is still a triple-panel window with short wave of f, and similarly for h and the right end of the domain of h. It follows that the only windows that need repair are the global windows of g and h and the windows whose waves are short at the right end of the domain of g or the left end of the domain of h. These windows correspond to the bananas rooted at nodes of the right spine of $\mathrm{Up}(g)$ and the left spine of $\mathrm{Up}(h)$.

In a nutshell, the algorithm visits the nodes in the relevant portions of the two spines from bottom to top. For $\mathrm{Up}(g)$, we get a sequence of nested short wave windows ending with the global window, and we list the corresponding critical points from right to left as $a_0, b_0, a_1, b_1, \ldots, a_i, b_i = \beta_g$. Symmetrically, we list the critical points we get from $\mathrm{Up}(h)$ from left to right as $a'_0, b'_0, a'_1, b'_1, \ldots a'_j, b'_j = \beta_h$. To have a specific setting, we assume that a_0 is a proper minimum of g (an up-type endpoint after removing the hook), while a'_0 is a hook that is an up-type endpoint of h. Because the windows are nested, we have

$$f(a_i) < \ldots < f(a_0) < f(x) < f(b_0) < \ldots < f(b_i); \tag{8}$$
$$f(a'_j) < \ldots < f(a'_1) < f(x) < f(b'_0) < \ldots < f(b'_j), \tag{9}$$

in which the hook, a'_0, has been removed from the second list; see Figure 3.5.5. The two orderings imply that the minimum with largest value is either a_0 or a'_1, and the maximum with smallest value is either b_0 or b'_0. To get a sorted list of nested windows, the algorithm pairs up the lowest maximum with the highest minimum, removes the two, and iterates.

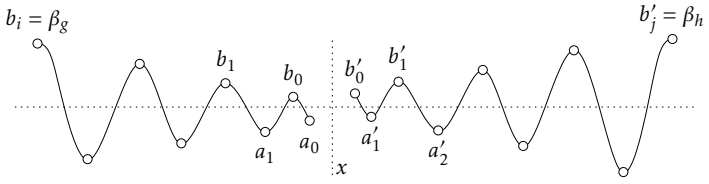

Figure 3.5.5: The simplified graph of g to the *left* of x, which connects the critical points that span bananas in the right spine of $\mathrm{Up}(g)$. The symmetrically simplified graph of h to the *right* of x, which connects the critical points that span bananas in the left spine of $\mathrm{Up}(h)$.

Given a_0, the next maximum and then the next minimum to its left are $b_0 = \mathsf{dth}(a_0)$ and $a_1 = \mathsf{low}(b_0)$. Symmetrically, $a'_1 = \mathsf{low}(b'_0)$ and $b'_1 = \mathsf{dth}(a'_1)$. Assuming the configuration in Figure 3.5.5, namely $f(a_0) < f(x) < f(b'_0)$, a_0 remains a minimum and b'_0 remains a maximum after connecting the two items monotonically. In other cases, a_0 and b'_0 may become non-critical. As a pre-processing step to gluing, we remove up-type and down-type endpoints that become non-critical when we connect the two lists. Whatever the case, the algorithm assumes that a_0 and b'_0 are the first critical points to the left and right of x, and that a_0 is a minimum and b'_0 is a maximum. We write α for the hook at b'_0 and use it as a dummy leaf, like in cutting. For the purposes of this algorithm we define $f(\beta_h) < f(\beta_g)$ for tie-breaking. We also define $\mathsf{low}(\beta_g) = \mathtt{nil}$ and $\mathsf{low}(\beta_h) = \mathtt{nil}$ with $f(\mathtt{nil}) = -\infty$. Again, we define the special root of the left tree, β_g, to be at $-\infty$ along the interval and swap the in- and mid-trails of the special banana of the left tree. This is reset after the gluing is complete.In each iteration, the maximum with lower value is paired with a minimum to its left or its right. If the maximum in the left tree, q, is processed, then the candidate minima are $\hat{p} = \mathsf{low}(q)$, which is on the left of q, and p, which is on the same side of q as $\mathrm{Bth}(q)$. If the maximum in the right tree is processed, then the situation is symmetric. The iteration ends when one of the two trees is empty, i.e., consists only of a special root and the dummy α.

```
function GLUE(a_0, b'_0, β_g, β_h):
  set q = b_0 = dth(a_0); q' = b'_0;
  repeat if f(q) < f(q') then
            p̂ = low(q); if p ≠ α then p = Bth(q) else p = Bth(q') endif;
            case f(p) > f(p̂) and p = Bth(q): UNDOINJURY(p, q);
                 f(p) > f(p̂) and p = Bth(q'): UNDOFATALITY(p, q);
                 f(p̂) > f(p): UNDOSCARE(p̂, q)
            endcase; q = dth(low(q))
         else symmetric cases for f(q) > f(q')
       endif
  until in(β_g) = mid(β_g) = α or in(β_h) = mid(β_h) = α;
  if in(β_g) = mid(β_g) = α then discard β_g, α else discard β_h, α endif.
```

The main three subroutines are inverses of the earlier functions and undo injuries, fatalities, and scares. Function UNDOINJURY extends the short wave banana beginning at q to a simple wave banana by inserting maxima into the in-trail beginning at q; see Figure 3.5.4 but read the change backward, from the left to the middle. Function UNDOFATALITY extends the short wave banana beginning at q to a simple wave banana while also changing the pairing of q; see again Figure 3.5.4 now reading the change from the right to the middle. Finally, function UNDOSCARE adjust the value of the up-type endpoint, $\alpha = \mathrm{Bth}(q)$, to $f(\alpha) = f(\hat{p}) - \varepsilon$, thereby extending the out-panel of the window associated with q and changing the pairing.

SUMMARY. We summarize by stating the running-time for cutting and concatenating lists in terms of the number of critical points and the number of changes to the augmented persistence diagrams.

Theorem 3.5.5 (Time to Cut and Concatenate)**.** *Let $f : [0, m+1] \to \mathbb{R}$ be a generic piecewise linear map with n maxima, and let g and h such that $f = g \cdot h$. The time to*

cut f into g and h is $O(\log n + k)$, in which k is the size of the symmetric difference between $\overrightarrow{\mathsf{Dgm}}(f)$ and $\overrightarrow{\mathsf{Dgm}}(g) \sqcup \overrightarrow{\mathsf{Dgm}}(h)$, and so is the time to concatenate g and h to form f.

Proof. We focus on the algorithm for cutting f at x, and omit the argument for glueing, which is symmetric. It takes $O(\log n)$ time to find the two consecutive critical points that sandwich x between then, and it takes $O(\log n)$ time to cut the dictionaries accordingly. The time needed to split $\mathrm{Up}(f)$ and $\mathrm{Dn}(f)$ is proportional to a constant plus the number of nodes visited during the iteration.

We argue that we can charge each such node, q, to a change in the augmented persistence diagram in such a way that each change is charged at most twice, once by a node in $\mathrm{Up}(f)$ and once by a node in $\mathrm{Dn}(f)$. Let q be a visited maximum in $\mathrm{Up}(f)$, and let p be the minimum with $\mathsf{dth}(p) = q$. Then $W(p,q)$ is a window with simple or short wave cut by x. If x cuts through the mid-panel or the out-panel, then $W(p,q)$ is neither a window of g nor of h, and we charge q to the disappearance of the point $(f(p), f(p))$ from the ordinary subdiagram. So suppose x cuts through the in-panel. If $W(p,q)$ is with simple wave, $W(q,p)$ is a window of $-f$ and x cuts through its out-panel, so we charge q to the disappearance of $(f(q), f(p))$ from the relative subdiagram. Finally, if $W(p,q)$ is a window with short wave and x cuts through its in-panel, then the algorithm has reached the spine of $\mathrm{Up}(f)$. By Lemma 3.4.3, all ancestors of q are of the same type, so we can end the traversal here. Thus, this case causes only constant work which is charged to the cutting operation directly. □

We remark that the $O(\log n + k)$ time bound for cutting and concatenating would not hold if in splitting and gluing we traversed the banana trees all the way to the respective special roots. Let q be the node on the spine where the algorithm halts. The path connecting q to the special root may be arbitrarily long, and none of these nodes corresponds to a change in the augmented persistence diagram. A particular map for which this happens is the *damped sine function*, defined by $f(x) = x \sin x$; see the right half of Figure 3.5.5 for a sketch. We get a triple-panel window for each min-max pair, each with short wave on the left. If we cut f close to 0, then all these windows get insubstantially smaller but remain to be spanned by the same min-max pairs. These changes are not visible in the augmented persistence diagrams of f and the functions created by cutting f at x.

3.6 Discussion

The main contribution of this paper is a dynamic data structure for maintaining the augmented persistence diagram of linear lists. The data structure starts with the Cartesian tree of Vuillemin [30] (see also Aragon and Seidel [92]), which it decomposes into paths and then splits into pairs of parallel trails, arriving at an unconventional representation referred to as banana tree. For the efficiency of the data structure, it is essential to maintain a pair of banana trees, which store complementary information about the list. The most important next step is the implementation of the data structure and its algorithms. With such an implementation, we can deepen our understanding of the persistence of random lists, which in turn can be used as a baseline for our understanding of non-random time series.

The theoretical foundations for our dynamic algorithms are described in [25], where maps on 1-dimensional domains, which include but go beyond intervals are described. Can the banana trees be extended to geometric trees (which allow for bifurcations) and geometric networks (which allow for bifurcations as well as loops) without deterioration of the asymptotic running time? Because of the unconventional representation of trees in terms of bananas (pairs of parallel trails), we finally ask whether there are other dynamic data structure questions that can benefit from this representation.

4

DYNAMICALLY MAINTAINING THE PERSISTENT HOMOLOGY OF TIME SERIES – CORRECTNESS PROOFS

This chapter contains proofs of correctness of the local and topological update algorithms presented in Chapter 3.

4.1 Correctness of Local Maintenance

In this section, we prove the correctness of the local operations and of the algorithms for Scenarios A and B (see Lemmas 4.1.21 and 4.1.22 in Section 4.1.6). We also give details on how to treat changes in value that lead to an up-type item becoming down-type or vice versa. We make extensive use of the homotopy $h_\lambda : [0, m+1] \to \mathbb{R}$ defined in Section 3.5.2 as $h_\lambda(x) = (1-\lambda)f(x) + \lambda g(x)$ for $0 \leq \lambda \leq 1$, where f and g differ in the value of a single item. The change in function value from f to g can be arbitrary and the transformation of $\mathrm{Up}(f)$ into $\mathrm{Up}(g)$ is achieved by a sequence of local operations, c which we define in terms of small changes in the function value of a single item. To make the notion of small change more precise, we define *contiguity* of items in terms of function value.

Definition 4.1.1 (Contiguity)**.** Items p and q are *contiguous in* f if there exists no item $u \neq \{p, q\}$ such that $f(u) \in [\min\{f(p), f(q)\}, \max\{f(p), f(q)\}]$.

Each local operation involving items p and q is then defined as the transformation of $\mathrm{Up}(h_\sigma)$ into $\mathrm{Up}(h_\theta)$, where p and q are contiguous in h_σ and h_θ. We prove the correctness of local operations in terms of such a small change, then show how to extend this to larger changes where items are not necessarily contiguous. This allows us to combine the local operations to form the Scenarios A and B. It is easy to see that when items remain contiguous upon exchanging values then they swap places in the order of items by function value. This is stated in the following lemma.

Lemma 4.1.2 (Local Changes)**.** *Let $0 \leq \sigma < \theta \leq 1$. Let p and q be items that are contiguous in both h_σ and h_θ with $h_\sigma(p) < h_\sigma(q)$ and $h_\theta(p) > h_\theta(q)$. The order of items by function value is the same in h_σ and h_θ except that p and q are swapped.*

In some of the correctness proofs, we analyze how a change in the function value of an item affects the structure of windows and leverage the correspondence between windows and bananas established in Lemma 3.4.1. In other correctness proofs, we use the fact that there is a unique banana tree satisfying the conditions proved in Lemma 3.4.2. To this end we reformulate the uniqueness conditions in terms of individual nodes and their pointers. This requires the following definitions.

Definition 4.1.3 (Ancestor and Descendant)**.** A node p is an *ancestor* of an internal node q if $p = \mathsf{up}^*(q)$ and of a leaf r if $p = \mathsf{up}^*(\mathsf{in}(r))$ or $p = \mathsf{up}^*(\mathsf{mid}(r))$, where up^* denotes a sequence of $\mathsf{up}(\cdot)$-pointers. A node s is a *descendant* of a node t if t is an ancestor of s.

Definition 4.1.4 (Banana subtree)**.** The *banana subtree rooted at a maximum* q is the banana tree consisting of the banana spanned by $\mathrm{Bth}(q)$ and q and all bananas whose maximum has an ancestor other than q on the in-trail or mid-trail starting at $\mathrm{Bth}(q)$ and ending at q.

We now state the new uniqueness conditions as Invariants 4.1.5 to 4.1.7.

Invariant 4.1.5. *For each maximum q of f, except for the special root, it holds that*

1. *if $Bth(q) < q$, then all nodes $u \neq q$ in the banana subtree rooted at q satisfy $u < q$ and all descendants v of $\mathsf{dn}(q)$, including $\mathsf{dn}(q)$, satisfy $v > q$;*
2. *if $Bth(q) > q$, then all nodes $u \neq q$ in the banana subtree rooted at q satisfy $u > q$ and all descendants v of $\mathsf{dn}(q)$, including $\mathsf{dn}(q)$, satisfy $v < q$.*

Invariant 4.1.6. *For each minimum p of f, except for the global minimum, it holds that $f(p) > f(\mathsf{low}(\mathsf{dth}(p)))$.*

Invariant 4.1.7. *For each maximum q of f, except for the special root, it holds that*

1. *$f(\mathsf{up}(q)) > f(q) > f(\mathsf{dn}(q))$;*
2. *if $q \neq \mathsf{in}(\mathsf{up}(q))$, then $\mathsf{up}(q) < q < \mathsf{dn}(q)$ or $\mathsf{dn}(q) < q < \mathsf{up}(q)$;*
3. *if $q = \mathsf{in}(\mathsf{up}(q))$, then either $\mathsf{up}(q) < q$ and $\mathsf{dn}(q) < q$ or $\mathsf{up}(q) > q$ and $\mathsf{dn}(q) > q$.*

We show that there is a unique banana tree satisfying Invariants 4.1.5 to 4.1.7 for a given map f. We first show that a banana tree satisfying Invariants 4.1.5 and 4.1.7 also satisfies Conditions III.1 and III.2.

Lemma 4.1.8 (Invariants imply Banana Conditions)**.** *A banana tree $\mathcal{B}$ that satisfies Invariants 4.1.5 and 4.1.7 also satisfies Conditions* III.1 *and* III.2*.*

Proof. Let p and q be a minimum and a maximum spanning a banana in $\mathcal{B}$. Assume $p < q$; the other case is symmetric. The requirement on function values increasing along the trails from p to q follows immediately from $f(\mathsf{dn}(u)) < f(u) < f(\mathsf{up}(u))$. We now show that the trails are also ordered along the interval. By Invariant 4.1.5, $\mathsf{mid}(q) < q$ and $\mathsf{in}(q) < q$. Write $p = v_0, v_1, \ldots, v_\ell = q$ for the nodes along the mid-trail with $v_1 = \mathsf{mid}(p)$, $v_{\ell-1} = \mathsf{mid}(q)$ and $v_i = \mathsf{up}(v_{i-1})$ for $2 \leq i \leq \ell$ and $v_i = \mathsf{dn}(v_{i+1})$ for $0 \leq i \leq \ell - 2$. If $v_{\ell-1} = \mathsf{mid}(q)$ is a maximum, then by Invariant 4.1.7 and $\mathsf{mid}(q) < q$ it satisfies $\mathsf{dn}(v_{\ell-1}) < v_{\ell-1} < \mathsf{up}(v_{\ell-1}) = q$. If $v_{\ell-2}$ is a maximum, then it in turn satisfies $\mathsf{dn}(v_{\ell-2}) < v_{\ell-2} < \mathsf{up}(v_{\ell-2}) = v_{\ell-1}$ by Invariant 4.1.7. Repeating this argument for all v_i for $1 \leq i \leq \ell - 3$ it follows that $p < v_1 < \cdots < v_\ell$.

Write $p = u_0, u_1, \ldots, u_k$ for the nodes along the in-trail with $u_1 = \mathsf{in}(p)$, $u_{k-1} = \mathsf{in}(q)$ and $u_i = \mathsf{up}(u_{i-1})$ for $2 \leq i \leq k$ and $u_i = \mathsf{dn}(u_{i+1})$ for $0 \leq i \leq k - 2$. By Invariant 4.1.7 and $\mathsf{in}(q) < q$ we have $u_{k-1} < \mathsf{dn}(u_{k-1})$. Following a similar argument as for the v_i we get $\mathsf{dn}(u_i) > u_i > \mathsf{up}(u_i)$ for all $1 \leq i \leq k - 2$ and it follows that $p = u_0 > u_1 > \cdots > u_{k-1}$. Thus, as required by Conditions III.1 and III.2, the trails are ordered along the interval and by function value, i.e., $\mathcal{B}$ satisfies Conditions III.1 and III.2. □

The next lemma states that a path-decomposed binary tree can be transformed into a banana tree satisfying the invariants.

Lemma 4.1.9 (Conditions imply Banana Invariants)**.** *If a path-decomposed binary tree T fulfills the Conditions* I.1, I.2 *and* II*, then the banana tree obtained from T as described in Section* 3.4.2 *satisfies Invariants* 4.1.5 *to* 4.1.7*.*

Proof. Let T be a path-decomposed binary tree satisfying Conditions I.1, I.2 and II and let $\mathcal{B}$ be the banana tree obtained from T as described in Section 3.4.2. We begin by proving that the banana tree $\mathcal{B}$ satisfies Invariant 4.1.7. Recall that $\mathcal{B}$ satisfies Conditions III.1 and III.2 and that by Lemma 3.4.2 it is unique. To see that this implies Invariant 4.1.7 consider any banana spanned by a minimum a and a maximum b. Assume $a < b$; the argument for $a > b$ is symmetric. Let $a = u_0, u_1, \ldots, u_j = b$ be the nodes along the in-trail and $a = v_0, v_1, \ldots, v_k = b$ be the nodes along the mid-trail. By Conditions III.1, III.2 we have

1. $u_i > u_{i+1}$ for all $0 \leq i \leq j-2$ and $f(u_i) < f(u_{i+1})$ for all $0 \leq i \leq j-1$,
2. $v_i < v_{i+1}$ for all $0 \leq i \leq k-1$ and $f(v_i) < f(v_{i+1})$ for all $0 \leq i \leq k-1$.

The pointers $\mathsf{dn}(\cdot)$ and $\mathsf{up}(\cdot)$ are defined such that $\mathsf{up}(u_i) = u_{i+1}$, $\mathsf{dn}(u_i) = u_{i-1}$ for all $1 \leq i \leq j-1$ and $\mathsf{up}(v_i) = v_{i+1}$, $\mathsf{dn}(v_i) = v_{i-1}$ for all $1 \leq i \leq k-1$. Together with Conditions III.1 and III.2 this implies that all maxima q on a trail between a and b satisfy $f(\mathsf{dn}(q)) < f(q) < f(\mathsf{up}(q))$ and, except u_{j-1}, they satisfy $\mathsf{dn}(q) < q < \mathsf{up}(q)$ or $\mathsf{up}(q) < q < \mathsf{dn}(q)$. By Condition III.1 it holds that $\mathsf{dn}(u_{j-1}) = u_{j-2} > u_{j-1}$, and by the assumption that $a < b$ it holds that $b = \mathsf{up}(u_{j-1}) > u_{j-1}$. This shows that Invariant 4.1.7 is satisfied for all u_i with $1 \leq i \leq j-1$ and v_i with $1 \leq i \leq k-1$, i.e, for all nodes internal on a trail from a to b. Since all maxima except the special root are internal to some trail it follows that $\mathcal{B}$ satisfies Invariant 4.1.7.

We now prove that $\mathcal{B}$ satisfies Invariant 4.1.6. We need to show that for every minimum p it holds that $f(p) > f(\mathsf{low}(\mathsf{dth}(p)))$. Assume by contradiction that for some minimum p this is not the case and let $a = \mathsf{low}(\mathsf{dth}(p))$, i.e., we assume $f(a) = f(\mathsf{low}(\mathsf{dth}(p))) > f(p)$. The inequality $f(\mathsf{low}(\mathsf{dth}(p))) > f(p)$ implies that $\mathsf{dth}(p) = q$ is an ancestor of a for which a does not have the smallest value in the subtree of q. Condition II requires that there is a path $P(a,q)$ and this is a contradiction as $q \neq \mathsf{dth}(a)$. It follows that $f(p) > f(\mathsf{low}(\mathsf{dth}(p)))$ for every minimum p and that $\mathcal{B}$ satisfies Invariant 4.1.6.

It remains to show that the banana tree $\mathcal{B}$ also satisfies Invariant 4.1.5. Let q be some maximum except the special root, let Q be the path ending at q and let p be the leaf at the other end of Q. Assume $p < q$; the other case is symmetric. The path Q is contained entirely in a subtree S_1 of q and the construction of $\mathcal{B}$ is such that this subtree becomes the banana subtree rooted at q. By Condition I.1 either for all $s \in S_1 : s < q$ or for all $s \in S_1 : s > q$. Since $p \in S_1$, it follows that all $s \in S_1$ satisfy $s < q$. The subtree S_1 becomes the banana subtree rooted at q in $\mathcal{B}$ and thus the condition that $u < q$ for all u in the banana subtree rooted at q is satisfied. Let S_2 be the other subtree of q in T. By Condition I.1 for all $t \in S_2$ we have $t > q$. The node $\mathsf{dn}(q)$ in $\mathcal{B}$ is one of the nodes in S_2 and thus $\mathsf{dn}(q) > q$. By Conditions III.1 and III.2 all descendants v of $\mathsf{dn}(q)$ on the same trail as $\mathsf{dn}(q)$ satisfy $v > \mathsf{dn}(q)$ and thus $v > q$. All these nodes v are also nodes in S_2, since they are on the same path as $\mathsf{dn}(q)$ and have smaller value. The banana subtree of each v is obtained from the subtree of v in T and thus these banana subtrees consist of nodes in subtrees of S_2. It follows that every descendant t of $\mathsf{dn}(q)$ including $\mathsf{dn}(q)$ satisfies $t > q$, as required by Invariant 4.1.5. This concludes the proof that $\mathcal{B}$ satisfies Invariant 4.1.5. □

We now give an algorithm that turns a banana tree into path-decomposed binary tree.

Lemma 4.1.10 (Merging Trails)**.** *There exists a deterministic algorithm to merge the trails of each banana in a banana tree satisfying Invariants 4.1.5 to 4.1.7, such that the resulting path-decomposed binary tree satisfies Conditions* I.1, I.2 *and* II.

Proof. Let $\mathcal{B}$ be a banana tree satisfying Invariants 4.1.5 to 4.1.7. Define the height of a banana subtree to be the longest simple path from the root of the banana subtree to a leaf following only $\mathsf{in}(\cdot)$, $\mathsf{mid}(\cdot)$ and $\mathsf{dn}(\cdot)$ pointers and with function value decreasing along the path. We show by induction over the height that any banana subtree of $\mathcal{B}$ satisfying Invariants 4.1.5 to 4.1.7 can be merged into a path-decomposed binary tree satisfying Conditions I.1, I.2 and II. Then, since the banana subtree rooted at the special root is equivalent to the banana tree $\mathcal{B}$, it follows that there is a deterministic algorithm to obtain a path-decomposed binary tree satisfying Conditions I.1, I.2 and II from $\mathcal{B}$.

We now give the algorithm for merging banana trees into path-decomposed binary trees. By induction over the height h we show that a banana subtree of $\mathcal{B}$ of height h with root q can be merged into a path-decomposed binary tree $T(q)$ such that Conditions I.1, I.2 and II are satisfied. To achieve this we also show that $\mathrm{Bth}(q) < q$ implies that q has no right child in $T(q)$ and $\mathrm{Bth}(q) > q$ implies that q has no left child in $T(q)$. This is needed to ensure that after assembling the subtrees of smaller height on a path, the nodes in the resulting binary tree are ordered correctly. Note that a banana tree consists of at least two nodes spanning a banana with empty trails, i.e., trails without interior nodes, so the base case is $h = 1$.

Base case: A banana tree of height 1 consists of a single banana spanned by a minimum p and a maximum q with $f(p) < f(q)$. We obtain a path-decomposed binary tree as follows: q becomes the tree with p as left child of q if $p < q$ and right child otherwise. There is a single path between p and q. This tree clearly satisfies Conditions I.1, I.2 and II. Furthermore, since $\mathrm{Bth}(q) = p$ if $p < q$, then q has no right child and if $p > q$ then q has no left child, so the claim holds.

Induction hypothesis: For some $h \geq 1$, for all $1 \leq j \leq h$ a banana subtree of $\mathcal{B}$ of height j can be merged into a path-decomposed binary tree T satisfying Conditions I.1, I.2 and II. Furthermore, $\mathrm{Bth}(q) < q$ implies that q has no right child in T and $\mathrm{Bth}(q) > q$ implies that q has no left child in T.

Induction step: Assume the induction hypothesis for some $h \geq 1$. We show that the claim holds for banana subtrees of height $h+1$. Let q be the root of a banana subtree of $\mathcal{B}$ of height $h+1$ and $p = \mathrm{Bth}(q)$. Banana subtrees rooted at a node on the banana spanned by p and q have height at most h, and by the induction hypothesis these banana subtrees can be merged into path-decomposed binary trees satisfying Conditions I.1, I.2 and II. Write $T(u)$ for the path-decomposed binary tree obtained from the banana subtree rooted at node u.

Let $s_0 = p, s_1, \ldots, s_\ell = q$ be the nodes on the trails between p and q ordered by function value. This ordering is unique since function values are distinct. Consider a node s_i for $1 \leq i \leq \ell - 1$. We show that for all $0 \leq j \leq i-1$, if $\mathrm{Bth}(s_i) < s_i$ then $s_i < s_j$ and $s_i < u$ for any descendant u of s_j, and if $\mathrm{Bth}(s_i) > s_i$ then $s_i > s_j$ and $s_i > u$ for any descendant u of s_j. Assume $\mathrm{Bth}(s_i) < s_j$; the other case is symmetric. By Invariant 4.1.5 we have $s_i < \mathsf{dn}(s_i)$ and for all descendants u of $\mathsf{dn}(s_i)$ it holds that $s_i < u$. Note that the nodes s_j for $0 \leq i-1$ that are on the same trail as s_i are descendants of $\mathsf{dn}(s_i)$ or $\mathsf{dn}(s_i)$. By Conditions III.1 and

III.2 the nodes s_k for $k \leq i-1$ that are on the other trail than s_i satisfy $s_i < s_k$ and $\mathsf{dn}(s_k) < s_j$. For these nodes s_k Invariant 4.1.5 implies $s_k < \mathrm{Bth}(s_k)$ and all descendants v of s_k satisfy $s_k < v$. This proves that for all $0 \leq j \leq i-1$ both $s_i < s_j$ and $s_i < u$ for any descendant of u.

We iteratively assemble the path-decomposed binary trees rooted at the s_i for $1 \leq i \leq \ell - 1$ into a path-decomposed binary tree rooted at q by linking $T(s_i)$ to the tree assembled from the $T(s_j)$ for $j \leq i-1$.

We begin with $i = 1$. Note that $\mathsf{dn}(s_1) = s_0$, since s_1 must be the bottom-most node of the in- or mid-trail between p and b. By Invariant 4.1.5, if $\mathrm{Bth}(s_1) < s_1$ then $\mathsf{dn}(s_1) > s_1$ and if $\mathrm{Bth}(s_1) > s_1$ then $\mathsf{dn}(s_1) < s_1$. Assume $\mathrm{Bth}(s_1) < s_1$; the other case is symmetric. By the induction hypothesis the root s_1 of the tree $T(s_1)$ has no right child in this tree. We make $s_0 = \mathsf{dn}(s_1)$ the right child of s_1 and since $s_0 = \mathsf{dn}(s_1) > s_1$ this ensures that the resulting tree satisfies Condition I.1. By Invariant 4.1.7, $f(\mathsf{dn}(s_1)) < f(s_1)$, so Condition I.2 is also satisfied.

Now consider any i with $1 \leq i \leq \ell - 1$. In iteration $i-1$ we have already linked the $T(s_j)$ for $1 \leq j \leq i$ into a tree T_{i-1} satisfying Conditions I.1 and I.2. The process to combine T_{i-1} with $T(s_i)$ is similar to that for $i = 1$. By Invariant 4.1.5 we have either $\mathrm{Bth}(s_i) < s_i < \mathsf{dn}(s_i)$ or $\mathrm{Bth}(s_i) > s_i > \mathsf{dn}(s_i)$. Assume again that $\mathrm{Bth}(s_i) < s_i < \mathsf{dn}(s_i)$; the other case is symmetric. The nodes in T_{i-1} are exactly the nodes s_j for $j \leq i-1$ and their descendants, so as discussed above, for all $t \in T_{i-1}$ it holds that $\mathrm{Bth}(s_i) < s_i < t$. By the induction hypothesis s_i has no right child in $T(s_i)$. We can thus attach T_{i-1} as the right subtree of s_i in $T(s_i)$ and the resulting tree T_i satisfies Condition I.1. The tree T_i also satisfies Condition I.2, as T_{i-1} satisfies Condition I.2 and $f(s_i) > f(s_{i-1})$.

The algorithm finishes by making $T_{\ell-1}$ the left subtree of q if $p < q$ and the right subtree otherwise, which yields the tree $T(q)$. Furthermore, we make $p = s_0, s_1, \ldots, s_\ell = q$ a path in $T(q)$. Since all nodes u in $T_{\ell-1}$ are in the banana subtree rooted at q they satisfy $u < q$ if $p < q$ and $u > q$ if $p > q$. Thus $T(q)$ satisfies Condition I.1. Condition I.2 is also satisfied since $T_{\ell-1}$ satisfies this condition and $f(q) > f(s_{\ell-1})$. Note also that $T(q)$ has no right child if $p := \mathrm{Bth}(q) < q$ and no left child if $p := \mathrm{Bth}(q) > q$, since the banana subtree rooted at q satisfies Invariant 4.1.5.

It remains to show that $T(q)$ satisfies Condition II. There can be no minimum $a \neq p$ in the banana subtree rooted at q with $f(a) < f(p)$, as otherwise there would exist a minimum c in the banana subtree rooted at q violating $f(\mathsf{low}(\mathsf{dth}(c)) < f(c)$ required by Invariant 4.1.6. This implies that p has the smallest value in the banana subtree rooted at q. It is connected to the root q by a path, as required by Condition II. Since all $T(s_i)$ for $1 \leq i \leq \ell - 1$ satisfy Condition II by the induction hypothesis we now only need to show that the path ending at each s_i satisfies Condition II. Write t_i for the other end of the path ending at s_i. We need to show that s_i is the nearest ancestor to t_i such that s_i has a descendant with smaller value than t_i in $T(q)$. Since $f(p) < f(t_i)$, as discussed above, s_i is indeed such an ancestor, and since t_i is connected to the root of $T(s_i)$ by a path there exists no other such ancestor in $T(s_i)$. Thus, Condition II holds. This concludes the induction and the proof of the lemma. □

Finally, we state our result on uniqueness of banana trees.

Corollary 4.1.11 (Invariants Determine the Banana Tree)**.** *Given a linear list of distinct values there is a unique banana tree satisfying Invariants 4.1.5 to 4.1.7.*

Proof. By Lemma 3.4.2 there exists a unique path-decomposed binary tree for the linear list that satisfies Conditions I.1, I.2 and II. By Lemmas 4.1.9

and 4.1.10 it follows that there is also a unique banana tree that satisfies Invariants 4.1.5 to 4.1.7. □

4.1.1 Slides

A *slide* occurs when an internal critical item becomes non-critical and its non-critical neighbor becomes critical. We distinguish two cases, based on whether the critical item is a maximum or a minimum.

MAX SLIDE: Let p be a non-critical item, q a neighbor of p and a maximum. The value of p increases above the value of q or the value of q decreases below the value of p.

MIN SLIDE: Let p be a non-critical item, q a neighbor of p and a minimum. The value of p decreases below the value of q or the value of q increases above the value of p.

In both cases, the number of critical items remains unchanged. If the order of critical items by function value is unaffected by the slide apart from q being replaced by p, then the banana tree can be updated by replacing the formerly critical item with the new critical item in the tree, which changes the label of the node associated with q to refer to p instead. We now prove that this correctly maintains the up-tree if the slide is caused by a sufficiently small change in value.

Lemma 4.1.12 (Max-Slide)**.** *Let p and q be neighboring items. Let $\sigma \in [0,1)$ be such that in h_σ the item p is non-critical, q is a maximum and p, q are contiguous. Let $\theta \in (\sigma,1]$ be such that in h_θ the item p is a maximum, q is non-critical and p, q are contiguous. The tree $Up(h_\theta)$ is obtained from $Up(h_\sigma)$ by replacing q with p.*

Proof. By Lemma 4.1.2, the order of maxima by function value is the same in h_σ and h_θ, with q replaced by p. The maps h_λ are defined such that they differ in exactly one fixed item j, so either $j = p$ or $j = q$. Neither p nor q is a minimum in h_σ or h_θ, so all minima have equal value in h_σ and h_θ. It then follows that the minimum s paired with q in h_σ is paired with p in h_θ. That is, if $W(s,q)$ is a window in h_σ, then $W(s,p)$ is a window in h_θ. Furthermore, $W(s,p)$ is nested into the same window in h_θ as $W(s,q)$ is in h_σ. No other window is affected by the change in value, since the order of critical items by function value is the same other than q being replaced by p. By Lemma 3.4.1, s and q span a banana in h_σ and s and p span a banana in h_θ. Thus, replacing q with p in the up-tree for h_σ yields the up-tree for h_θ, as claimed. □

Lemma 4.1.13 (Min-Slide)**.** *Let p and q be neighboring items. Let $\sigma \in [0,1)$ be such that in h_σ the item p is non-critical, q is a minimum and p and q are contiguous. Let $\theta \in (\sigma,1]$ be such that in h_θ the item p is a minimum, q is non-critical and p and q are contiguous. The tree $Up(h_\theta)$ is obtained from $Up(h_\sigma)$ by replacing q with p.*

Proof. By Lemma 4.1.2, the order of minima by function value is the same in h_σ and h_θ, with q replaced by p. Since the values of maxima are equal in h_σ and h_θ it then follows that the maximum s paired with q in h_σ is paired with p in h_θ. That is, if $W(q,s)$ is a window in h_σ, then $W(p,s)$ is a window in h_θ. Furthermore, $W(p,s)$ is nested into the same window in h_θ as $W(s,q)$ is in h_σ. No other window is affected by the change in value, since the order

of critical items by function value is the same other than q being replaced by p. By Lemma 3.4.1, q, s span a banana in h_σ and p, s span a banana in h_θ. Thus, replacing q with p in the up-tree for h_σ yields the up-tree for h_θ, as claimed. □

4.1.2 Cancellations

Let p be a minimum, q be a maximum and a neighbor of p, such that there is a window $W(p,q)$ without any windows nested into it, i.e., such that p and q span a banana in the banana tree whose trails contain no other critical items. A cancellation occurs when the value of p increases above the value of q or the value of q decreases below the value of p. Then, both p and q become non-critical and the window spanned by p and q disappears. To update the up-tree we remove the banana spanned by p and q by connecting $\mathsf{up}(q)$ to $\mathsf{dn}(q)$ and discarding p and q.

Lemma 4.1.14 (Cancellation)**.** *Let p, q be neighboring items. Let $\sigma \in [0,1)$ be such that in h_σ the item p is a minimum, q is a maximum and p and q are contiguous and paired in h_σ. Let $\theta \in (\sigma,1]$ be such that in h_θ the items p and q are non-critical. Then nodes p and q span a banana in $Up(h_\sigma)$ that does not contain any other critical items and removing this banana yields $Up(h_\theta)$.*

Proof. Let $\gamma \in (\sigma,\theta]$ be such that in h_γ the items p and q are non-critical and contiguous. By Lemma 4.1.2, the order of items by function value is the same in h_σ and h_γ, except that p and q are swapped. In particular, the order of critical items by function value is the same in h_σ and h_γ. Since p, q are non-critical in h_γ and in h_θ it follows that the order of critical items remains the same. Thus, the order of critical items in h_σ and h_θ is identical.

Since p and q are contiguous in h_σ, there is no critical item with value between $h_\sigma(p)$ and $h_\sigma(q)$. As p and q are neighbors this implies that p and q are paired in h_σ: consider the process to obtain the pairing described in Section 3.2.2; the component born at p in h_σ dies at q, since the component on the other side of q must be born at a lower value. Furthermore, there are no windows nested into $W(p,q)$, since there is no critical item with value in $[h_\sigma(p),h_\sigma(q)]$. All other windows in h_σ are also windows in h_θ, since the critical items have the same order by function value.

As $W(p,q)$ is a window of h_σ, by Lemma 3.4.1 the nodes p and q span a banana in $\mathrm{Up}(h_\sigma)$. There are no windows nested into $W(p,q)$ and thus the trails between p and q are empty. Since p and q are non-critical in h_θ they are not present in $\mathrm{Up}(h_\theta)$. Thus, removing the banana spanned by p and q from $\mathrm{Up}(h_\sigma)$ yields $\mathrm{Up}(h_\theta)$, as claimed. □

4.1.3 Slides and Cancellations Involving Endpoints

When the criticality of the neighbor of an endpoint changes it can also change the endpoint from up-type to down-type or vice versa. Likewise, a change in the criticality of an endpoint leads to a change in the criticality of its neighbor. The operations needed to maintain the up-tree in these cases are similar to cancellations and slides. Let $q \in \{1,m\}$ be an endpoint and a be its neighbor. In the following we assume $a < q$, i.e., that q is the right endpoint of the interval. The case $q < a$ is symmetric. Let $0 \le \sigma < \theta \le 1$ such that

$h_\sigma(q) > h_\sigma(a)$, $h_\theta(q) < h_\theta(a)$ or $h_\sigma(q) < h_\sigma(a)$, $h_\theta(q) > h_\theta(a)$. There are four cases:

1. $h_\sigma(q) > h_\sigma(a)$, $h_\theta(q) < h_\theta(a)$, a is a minimum in h_σ and non-critical in h_θ,
2. $h_\sigma(q) > h_\sigma(a)$, $h_\theta(q) < h_\theta(a)$, a is non-critical in h_σ and a maximum in h_θ,
3. $h_\sigma(q) < h_\sigma(a)$, $h_\theta(q) > h_\theta(a)$, a is a maximum in h_σ and non-critical in h_θ,
4. $h_\sigma(q) < h_\sigma(a)$, $h_\theta(q) > h_\theta(a)$, a is non-critical in h_σ and a minimum in h_θ.

We now give the algorithm for the case $a < q$.

```
1  cancel-or-slide-endpoint(a, q):
2    if q changes from down-type to up-type then
3      if a becomes non-critical then
4        remove q and Bth(q); replace a by q
5      else if a becomes a maximum then
6        let p = Bth(q); replace q by a; replace p by q
7      endif
8    else if q changes from up-type to down-type then
9      if a becomes non-critical then
10       replace q by the hook; replace a by q
11     else if a becomes a minimum then
12       replace q by a; insert q as in(a); add a banana between q and its hook
13     endif;
14   endif.
```

Lemma 4.1.15 (Changes Involving Endpoints)**.** *Let q be an endpoint, a be its neighbor and let $\check{q}$ be the hook neighboring q. Let $0 \leq \sigma < \theta \leq 1$ such that*

1. *either $h_\sigma(q) > h_\sigma(\check{q}) > h_\sigma(a)$, $h_\theta(q) < h_\theta(\check{q}) < h_\theta(a)$,*
2. *or $h_\sigma(q) < h_\sigma(\check{q}) < h_\sigma(a)$, $h_\theta(q) > h_\theta(\check{q}) > h_\theta(a)$,*

where q and $\check{q}$ are contiguous and $\check{q}$ and a are contiguous in h_σ and h_θ. Applying the algorithm `cancel-or-slide-endpoint`(a,q) *to the up-tree $Up(h_\sigma)$ yields the up-tree $Up(h_\theta)$.*

Proof. Assume $a < q$; the other case is symmetric. We show how the change in function value from h_σ to h_θ affects the windows in each of the cases and that the algorithm correctly updates the bananas to match the windows of h_θ. Recall that the maps h_λ are defined such that they differ in a single fixed item. Thus, we can assume that either $h_\theta(q) = h_\sigma(q)$ or $h_\theta(a) = h_\sigma(a)$ and for all $u \neq q, a$ it holds that $h_\theta(u) = h_\sigma(u)$.

$h_\sigma(q) > h_\sigma(a)$, $h_\theta(q) < h_\theta(a)$, a IS NON-CRITICAL IN h_θ (LINE 4): There is a component in h_σ born at $\check{q}$ that dies at q and a component born at a that dies at an item b. The component born at $\check{q}$ merges into the component born at a. These components correspond to two windows $W(\check{q}, q)$ and $W(a, b)$, where the former is nested into the latter. In h_θ there are no components born at $\check{q}$ or a, as they are no longer homological critical points. Instead, there is a component born at q, which corresponds to a window $W(q, \cdot)$. As q, $\check{q}$ and $\check{q}$, a are contiguous in h_σ and h_θ the order

of items by function value in h_σ and h_θ differs only in that q, $\check{q}$ and a are reversed by Lemma 4.1.2. Since q neighbors a it follows that the component born at q in h_θ dies at b, i.e., there is a window $W(q,b)$ in h_θ. The algorithm reflects this change in windows: the banana spanned by $\check{q}$ and q corresponding to $W(\check{q},q)$ is removed; the banana spanned by a and b corresponding to $W(a,b)$ is replaced by a banana spanned by q and b corresponding to $W(q,b)$.

$h_\sigma(q) > h_\sigma(a)$, $h_\theta(q) < h_\theta(a)$, a IS A MAXIMUM IN h_θ (LINE 6): Going from h_σ to h_θ can be seen as relabeling q such that it represents item a, relabeling $\check{q}$ such that it represents q and removing the item formerly associated with a. Changing the function values to reflect the values in h_θ does not affect the ordering of items, due to contiguity of q, $\check{q}$ and $\hat{q}$, a. The corresponding change in the up-tree is to relabel the node representing q to represent item a, and relabeling the node representing the hook to represent item q, as the algorithm does in Line 6.

$h_\sigma(q) < h_\sigma(a)$, $h_\theta(q) > h_\theta(a)$, a IS NON-CRITICAL IN h_θ (LINE 10): Similarly to the previous case the change from h_σ to h_θ can be seen as insertion of an item $a^* < a$ next to a and contiguous with q in h_σ, then relabeling q to represent $\check{q}$, a to represent q and a^* to represent a. Adjusting the values of the items to reflect the values in h_θ does not affect the ordering of items, due to contiguity of q, $\check{q}$ and $\check{q}$, a. The corresponding change in the up-tree is to relabel the node representing q to represent $\check{q}$, and relabel the node representing a to represent q, as the algorithm does in Line 10.

$h_\sigma(q) < h_\sigma(a)$, $h_\theta(q) > h_\theta(a)$, a IS A MINIMUM IN h_θ (LINE 12): This is the reverse of the first case: there is a component born at q in h_θ and it is replaced by a component born at a and another component born at $\check{q}$ in h_θ. Reverse to the case above the window $W(q,b)$ is replaced by the window $W(a,b)$ and a new window $W(\check{q},q)$ appears. The change from $W(q,b)$ to $W(a,b)$ is reflected by replacing q by a in the up-tree. The new window $W(\check{q},q)$ must be nested into $W(a,b)$ as $h_\theta(a) < h_\theta(q) < h_\theta(b)$, since a, $\check{q}$ and q are contiguous in h_θ. Furthermore, $W(\check{q},q)$ must be nested into the in-panel, as $b < a$. The nesting of $W(\check{q},q)$ into the in-panel of $W(a,b)$ is reflected in the algorithm by adding a banana spanned by $\check{q}$ and q into the in-trail of the banana spanned by a and b.

The algorithm updates the bananas to reflect the windows and their nesting hierarchy in h_θ, which implies by Lemma 3.4.1 that the new up-tree is $\mathrm{Up}(h_\theta)$. □

4.1.4 Anti-Cancellations

Let p and q be neighboring non-critical items. If the value of one changes such that both become critical, a new window is introduced. This is the reverse of a cancellation and we call it an *anti-cancellation*.

Denote by h_σ and h_θ the map before and after the anti-cancellation, respectively, and assume that in both h_σ and h_θ the items p and q are contiguous. Furthermore, assume that p becomes a minimum and q a maximum. To change $\mathrm{Up}(h_\sigma)$ into $\mathrm{Up}(h_\theta)$ we need to introduce a banana spanned by p and q. We recall the algorithm for the case $p < q$; the case $p > q$ is symmetric. The algorithm first identifies the maximum b closest to p such that $b < p < q$, i.e.,

such that p becomes the critical item next to b. It then walks down the path from b to its successor until it reaches the first node t such that $h_\theta(t) < h_\theta(q)$. The node q is then inserted into this path above t, and a new banana spanned by p and q is attached at q. To insert a node q between some nodes a and b, where a and b are neighbors on the same trail and $f(a) < f(b)$, we set $\mathsf{up}(q) = b$, $\mathsf{dn}(q) = a$ and make the pointer from a to b and the pointer from b to a point to q instead.

```
1  anticancel(p,q):
2    find the maximum b closest to p such that b < p < q
3    if b < Bth(b) then t = mid(b) else t = dn(b) endif;
4    while h_θ(t) > h_θ(q) do t = in(t) endwhile;
5    if t is a leaf then if t = Bth(b) then insert q between t and mid(t)
6                                      else  if t is the global minimum
7                                                and t < q then
8                                                insert q between t and mid(t)
9                                      else insert q between t and in(t)
10                        endif;
11                 else insert q between t and up(t) into the trail of t
12   endif.
13   Set in(p) = mid(p) = q, in(q) = mid(q) = p, dth(p) = q, low(q) = low(dn(q)).
```

Lemma 4.1.16 (Anti-Cancellation)**.** *Let p and q be neighboring items. Let $\sigma \in [0,1)$ such that in h_σ both p and q are non-critical, $h_\sigma(p) > h_\sigma(q)$, and p and q are contiguous. Let $\theta \in (\sigma, 1]$ such that in h_θ the item p is a minimum, q is a maximum, and p and q are contiguous. Algorithm* `anticancel`(p,q) *applied to the up-tree $\mathrm{Up}(h_\sigma)$ yields the up-tree $\mathrm{Up}(h_\theta)$.*

Proof. We prove the lemma for the case where $p < q$; the other case is symmetric. The tree $\mathrm{Up}(h_\sigma)$ satisfies Invariants 4.1.5 to 4.1.7. We show that the tree obtained by `anticancel` also satisfies these invariants for h_θ, which by Corollary 4.1.11 implies that it is the unique up-tree for h_θ.

The algorithm first finds the maximum b closest to p such that $b < p < q$. Since p and q are non-critical in h_σ, the item p cannot be an endpoint. Furthermore, $h_\sigma(p) > h_\sigma(q)$ and thus this maximum b exists. After the while-loop terminates t satisfies $h_\theta(t) \leq h_\theta(q)$. Since p and q are contiguous in h_θ it holds that $h_\theta(t) \neq h_\theta(q)$ and thus $h_\theta(t) < h_\theta(q)$ and $h_\theta(t) < h_\theta(p)$.

The node q is inserted above t such that $\mathsf{dn}(q) = t$. As $\mathsf{low}(q)$ is set to $\mathsf{low}(\mathsf{dn}(q)) = \mathsf{low}(t)$, we have $h_\theta(\mathsf{low}(q)) < h_\theta(p)$ and it follows that $h_\theta(\mathsf{low}(q)) = h_\theta(\mathsf{low}(\mathsf{dth}(p))) < h_\theta(p)$. Since the $\mathsf{low}(\cdot)$ and $\mathsf{dth}(\cdot)$ pointers of all other nodes are unchanged between $\mathrm{Up}(h_\sigma)$ and the new up-tree, the new up-tree satisfies Invariant 4.1.6.

We now show that Invariant 4.1.5 holds in the new up-tree. Let t_0 be the node t as initialized in the first if-statement and t_i be the node t after the i-th iteration of the loop. Denote by I the last iteration, such that t_I is the node t after the loop terminates. Note that all t_i for $i \geq 0$ are present in $\mathrm{Up}(h_\sigma)$. As Invariant 4.1.5 holds for $\mathrm{Up}(h_\sigma)$, the initialization of t guarantees $b < t_0$. Furthermore, all t_i for $i \geq 1$ are descendants of t_0 and thus $b < t_i$ for all $i \geq 0$. By definition of b and since p and q are neighbors it also holds that $q < t_i$ for all $i \geq 0$ and thus $b < p < q < t_i$ for all $i \geq 0$.

To continue the proof of Invariant 4.1.5 we need the following claim:

Claim 4.1.17. *The while-loop terminates. For all $0 \leq i \leq I-1$ it holds that t_i is not a leaf, $Bth(t_i) < t_i$ and $t_{i+1} < t_i$.*

Proof. There exists a minimum a such that $b < p < q < a$ and such that $h_\sigma(a) < h_\sigma(q)$ and such that there are no critical items u with $q < u < a$. This follows from a similar argument as for the existence of b: q is non-critical in h_σ and thus cannot be an endpoint and $h_\sigma(q) < h_\sigma(p)$, so the first critical item greater than q must be a minimum. Note that a must be either t_0 or a descendant of t_0 and that in the latter case it is the leftmost descendant of t_0 by Invariant 4.1.5. As the algorithm modifies the up-tree after the while-loop terminated, we only argue about $\mathrm{Up}(h_\sigma)$ in this proof. Recall that this up-tree satisfies Invariants 4.1.5 to 4.1.7 and that by Lemma 4.1.8 it also satisfies Conditions III.1 and III.2.

By Invariant 4.1.5 $b = \mathsf{up}(t_0) < t_0$ and by Invariants 4.1.5 and 4.1.7 $\mathsf{in}(t_0) < t_0 < \mathsf{dn}(t_0)$, unless t_0 is a leaf. The while-loop assigns $t_1 = \mathsf{in}(t_0)$ and by Conditions III.1, III.2 and Invariant 4.1.5 this is the leftmost node among the other nodes on the banana with maximum and their descendants. Invariants 4.1.5 and 4.1.7 again $\mathsf{in}(t_1) < t_1$, unless t_1 is a leaf. This argument can be repeated for $\mathsf{in}(t_1), \mathsf{in}(\mathsf{in}(t_1)), \ldots$ until we reach a leaf. Thus, by following $\mathsf{in}(\cdot)$-pointers from t_0 we reach the leftmost leaf that is a descendant of t_0, which is a. As $h_\theta(a) = h_\sigma(a) < h_\sigma(q) \leq h_\theta(q)$, there exists a node $\mathsf{in}^*(t_0)$ with $h_\theta(\mathsf{in}^*(t_0)) < h_\theta(q)$ and hence the while-loop terminates. If the loop terminates in a leaf, then t_I is a leaf and all t_i with $0 \leq i \leq I-1$ are not leaves. If the loop terminates at an internal node, then all t_i for $0 \leq i \leq I$ are not leaves. We have also seen that $\mathsf{in}(t_i) < t_i$ for all $0 \leq i \leq I$ and this implies $Bth(t_i) < t_i$ by Invariant 4.1.5. This proves the claim. □

We resume the proof of Lemma 4.1.16. Claim 4.1.17 implies that the t_i are ordered such that $t_0 > t_1 > \cdots > t_I$. For all $0 \leq i \leq I-1$ the nodes p and q are inserted into the banana subtree rooted at t_i, as either $q = \mathsf{in}(t_i)$ or q is a descendant of $\mathsf{in}(t_i)$. Since $Bth(t_i) < t_i$ and $p < q < t_i$ for $0 \leq i \leq I-1$, it follows that no t_i for $0 \leq i \leq I-1$ violates Invariant 4.1.5. Recall that q becomes $\mathsf{up}(t_I)$; this implies that neither the banana subtree rooted at t_I nor the descendants of $\mathsf{dn}(t_I)$ change and thus t_I satisfies the condition of Invariant 4.1.5 in the new tree. Finally, b also satisfies the condition of Invariant 4.1.5, as $b < p < q < t_0$ and p and q are inserted into the subtree of b that contains t_0. It follows that there is no node that violates the condition of Invariant 4.1.5 and thus the new up-tree satisfies Invariant 4.1.5.

We use the same claim to show that Invariant 4.1.7 is satisfied after inserting q and p. There are two cases: $t_I \neq t_0$ and $t_I = t_0$. In the first case, $q = \mathsf{in}(t_{I-1})$, $t_I = \mathsf{dn}(q)$ and $t_{I-1} = \mathsf{up}(q)$. As shown above $t_I > q$ and $t_{I-1} > q$. Furthermore, since the while-loop did not terminate for $t = t_{I-1}$ we have $h_\theta(q) < h_\theta(t_{I-1})$. It also holds that $h_\theta(t_I) < h_\theta(q)$, as the while-loop terminated for $t = t_I$. Thus, q satisfies the conditions in Invariant 4.1.7. In the case $t_I = t_0$, $t_I = \mathsf{dn}(q)$, $\mathsf{up}(q) = b$ and either $q = \mathsf{mid}(b)$ or $q = \mathsf{dn}(b)$. We have $b < q < t_I$, i.e., $\mathsf{up}(q) < q < \mathsf{dn}(q)$. As in the other case $h_\theta(t_I) < h_\theta(q)$. It is also true that $h_\theta(q) < h_\theta(b)$, which follows from p and q being contiguous in h_θ and the definition of b. Thus, q again satisfies the conditions in Invariant 4.1.7. For t_I and t_{I-1} the conditions in Invariant 4.1.7 follow from Claim 4.1.17 and the fact that $h_\theta(t_I) < h_\theta(q) < h_\theta(t_{I-1})$. No other node can violate Invariant 4.1.7 and thus the new up-tree satisfies Invariant 4.1.7.

We have shown that the tree constructed by applying anticancel(p,q) to Up(h_σ) yields an up-tree that satisfies Invariants 4.1.5 to 4.1.7 for h_θ. By Corollary 4.1.11 this is the unique up-tree for h_θ. □

4.1.5 Interchanges

INTERCHANGES OF MAXIMA. An interchange of maxima occurs when the value of a maximum j increases above the value of a maximum q or the value of q decreases below the value of j. This only has structural consequences on the up-tree if $q = \mathsf{up}(j)$, as this is the only case where one of the invariants, namely Invariant 4.1.7, is violated.

Denote by h_σ and h_θ the map before and after the interchange, such that $h_\sigma(j) < h_\sigma(q)$ and $h_\theta(j) > h_\theta(q)$. Assume $q < j$, which is the situation depicted in Figure 3.5.2. The case $j < q$ is symmetric. Let i and p be such that $j = \mathsf{dth}(i)$ and $q = \mathsf{dth}(p)$ in Up(h_σ). There are three cases: $j = \mathsf{dn}(q)$, $j = \mathsf{in}(q)$ and $j = \mathsf{mid}(q)$. If $j = \mathsf{in}(q)$ or $j = \mathsf{mid}(q)$, then $p = \mathsf{low}(j)$ and by Invariant 4.1.6 $h_\sigma(i) > h_\sigma(p)$. If $j = \mathsf{dn}(q)$ we further distinguish two cases depending on the order of $h_\sigma(i)$ and $h_\sigma(p)$. We now give the algorithm for processing an interchange of maximum j with maximum $q = \mathsf{up}(j)$ for the case $q < j$.

```
1  max-interchange(j,q):
2    Let i = Bth(j), p = Bth(q);
3    if j = in(q) then
4      remove j from its trail; insert j as up(q)
5    else if j = mid(q) then
6      exchange j and q;
7      remove q from its trail; insert q as dn(j);
8      swap in- and mid-trail of i and q
9    else if j = dn(q) then
10     if h_σ(i) < h_σ(p) then
11       remove q from its trail; insert q as in(j)
12     else
13       exchange j and q;
14       remove q from its trail; insert q as mid(j);
15       swap in- and mid-trail of i and q
16     endif;
17   endif.
```

The next lemma states that this algorithm correctly maintains the up-tree.

Lemma 4.1.18 (Max-interchange)**.** *Let j and q be two items. Let $\sigma \in [0,1)$ such that in h_σ the items j and q are maxima with $q = \mathsf{up}(j)$ and such that j and q are contiguous in h_σ. Let $\theta \in (\sigma,1]$ such that in h_θ the items j and q are maxima with $h_\theta(j) > h_\theta(q)$ and such that j and q are contiguous in h_θ. Applying* max-interchange(j,q) *to the up-tree Up(h_σ) yields the up-tree Up(h_θ).*

Proof. The tree Up(h_σ) satisfies Invariants 4.1.5 to 4.1.7. Recall that by Lemma 4.1.8 this up-tree also satisfies Conditions III.1 and III.2. We show that the tree obtained by max-interchangealso satisfies these invariants for h_θ, which by Corollary 4.1.11 implies that it is the unique up-tree for h_θ.

Write I_q for the nodes internal to the in-trail beginning at q and M_q for the nodes internal to the mid-trail beginning at q in Up(h_σ); similarly, write I_p, I_i,

I_j for the nodes internal to the in-trails and M_p, M_i, M_j for the nodes internal to the mid-trails at p, i, j.

There are three cases in `max-interchange`: $j = \mathsf{in}(q)$, $j = \mathsf{mid}(q)$ and $j = \mathsf{dn}(q)$. As $q = \mathsf{up}(j)$ there is no other case to consider. The case $j = \mathsf{dn}(q)$ is further divided into two cases based on the values of $\mathrm{Bth}(j)$ and $\mathrm{Bth}(q)$. We describe for each case how the trails change from $\mathrm{Up}(h_\sigma)$ to the new tree and show that Invariants 4.1.5 and 4.1.7 hold in the new tree. Afterwards we prove Invariant 4.1.6.

Case 1 ($j = \mathsf{in}(q)$): Assume $j < q$; the other case is symmetric. Note that $i = \mathrm{Bth}(j) < j$ and $p = \mathrm{Bth}(q) < q$. The in-trail and mid-trail between i and j remain the in-trail and mid-trail between i and j, and the in-trail and mid-trail between p and q remain the in-trail and mid-trail between p and q, with the exception that j is removed from the in-trail between p and q. In the new tree the nodes internal to the in-trail between i and j are thus I_j and the nodes internal to the mid-trail between i and j are M_j. Similarly, the nodes internal to the in-trail between p and q are $I_q \setminus \{j\}$ and the nodes internal to the mid-trail between p and q are M_q.

To see that Invariant 4.1.5 holds we analyze the subtrees of j and q to show that they satisfy the condition of Invariant 4.1.5 in the new up-tree. For any other node the condition holds in the new tree as the nodes in the respective subtrees do not change. First, consider the node q. The node $\mathsf{dn}(q)$ is the same in $\mathrm{Up}(h_\sigma)$ and in the new tree. The algorithm also does not modify the descendants of $\mathsf{dn}(q)$. Thus, for all descendants u of $\mathsf{dn}(q)$ including $\mathsf{dn}(q)$ we have $q < u$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$, as $p = \mathrm{Bth}(q) < q$. The banana subtree rooted at q in the new tree differs from that in $\mathrm{Up}(h_\sigma)$ only by the banana subtree rooted at j, which the algorithm removes. Thus, for all nodes v in the banana subtree rooted at q it holds that $v < q$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$, as $p = \mathrm{Bth}(q) < q$. It follows that q satisfies the condition for Invariant 4.1.5. Now consider the node j. The algorithm does not change the banana subtree rooted at j and thus for all v in this subtree we have $v < j$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$, as $i = \mathrm{Bth}(j) < j$. The other subtree of j, i.e., the descendants of $\mathsf{dn}(j)$ differ significantly: in $\mathrm{Up}(h_\sigma)$ these were the descendants of nodes on the in-trail below j; in the new tree they are the descendants of q. For the descendants u_1 of $\mathsf{dn}(q)$ it is easy to see that $j < u_1$, as $j < q < u_1$, which follows from the discussion for q and the assumption that $j < q$. For the descendants u_2 of nodes on the banana spanned by p and q in the new tree it follows from Conditions III.1 and III.2 and Invariant 4.1.5 that $j < u_2$: the node j is the topmost node on the in-trail, and with $j < q$ this implies that the other maxima t on the banana satisfy $j < t$; nodes u_3 in the banana subtrees rooted at each t are either descendants of $\mathsf{dn}(j)$, which implies $j < u_3$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$, or t is on the mid-trail between p and q, which by $p < q$ and Invariant 4.1.5 also implies $j < t < u_3$. It follows that all descendants u of $\mathsf{dn}(j) = q$, including $\mathsf{dn}(j)$ satisfy $j < u$. Thus, the condition of Invariant 4.1.5 holds for j.

We now prove that Invariant 4.1.7 holds by again analyzing j and q; for the remaining nodes the conditions of Invariant 4.1.7 follow directly from Invariant 4.1.7 for $\mathrm{Up}(h_\sigma)$. Let $u_q = \mathsf{up}(q)$ and $d_q = \mathsf{dn}(q)$ in $\mathrm{Up}(h_\sigma)$. In the new tree $d_q = \mathsf{dn}(q)$, $u_q = \mathsf{up}(j)$ and $j = \mathsf{up}(q)$. By the assumption $j < q$ and Invariant 4.1.7 for $\mathrm{Up}(h_\sigma)$ we have $j < q < d_q$ in the new tree, i.e., $\mathsf{up}(q) < q < \mathsf{dn}(q)$. If $q = \mathsf{in}(u_q)$ in $\mathrm{Up}(h_\sigma)$ then $q < u_q$ and in the new tree

$j = \mathsf{in}(u_q)$ and $j < u_q$. Thus, if $j = \mathsf{in}(\mathsf{up}(j)) = \mathsf{in}(u_q)$ in the new tree then $j < \mathsf{up}(j)$ and $j < \mathsf{dn}(j) = q$. Otherwise, if $q \neq \mathsf{in}(u_q)$ in $\mathrm{Up}(h_\sigma)$ then $u_q < q$ and $u_q < j$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. In the new tree this implies $j \neq \mathsf{in}(\mathsf{up}(j)) = \mathsf{in}(u_q)$ and $u_q = \mathsf{up}(j) < j < \mathsf{dn}(j) = q$. The inequalities $h_\theta(\mathsf{up}(j)) > h_\theta(j) > h_\theta(\mathsf{dn}(j)) = h_\theta(q)$ and $h_\theta(j) = h_\theta(\mathsf{up}(q)) > h_\theta(q) > h_\theta(\mathsf{dn}(q))$ follow directly from the fact that j and q are contiguous in h_θ and from Invariant 4.1.7 for $\mathrm{Up}(h_\sigma)$. Thus, j and q satisfy the conditions of Invariant 4.1.7 in the new tree and it follows that Invariant 4.1.7 holds in the new tree.

CASE 2 ($j = \mathsf{mid}(q)$): Assume $j < q$; the other case is symmetric. Note that $i = \mathrm{Bth}(j) > j$ and $p = \mathrm{Bth}(q) < q$. The nodes q and j exchange their partners, i.e., in the new up-tree $\mathrm{Bth}(q) = i$ and $\mathrm{Bth}(j) = p$. The in-trail and mid-trail between i and j become the mid- and in-trail between i and q (notice that the trails change from in to mid and vice versa); the in-trail and mid-trail between p and q become the in- and mid-trail between p and q, with j removed from the mid-trail. That is, in the new tree the nodes internal to the in-trail between i and q are the nodes M_j and the nodes internal to the mid-trail between i and q are the nodes I_j. The nodes internal to the in-trail between p and j are the nodes I_q and the nodes internal to the mid-trail between p and j are the nodes $M_q \setminus \{j\}$.

The proof of Invariant 4.1.7 is identical to that of Case 1. To see that Invariant 4.1.5 holds we again analyze the subtrees of j and q to show that they satisfy the condition of Invariant 4.1.5. As in the previous case the remaining nodes satisfy the condition in the new tree, as their subtrees are unchanged by the algorithm. For the node q observe that $\mathsf{dn}(q)$ and its descendants are the same in $\mathrm{Up}(h_\sigma)$ and the new tree. Furthermore, the nodes in the banana subtree rooted at q in the new tree are also in the banana subtree rooted at q in $\mathrm{Up}(h_\sigma)$. The condition of Invariant 4.1.5 thus holds for q by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. Now we consider the node j. The descendants of $\mathsf{dn}(j)$ are (1) the descendants of $\mathsf{dn}(q)$, including $\mathsf{dn}(q)$, and (2) the node q along with the banana subtree rooted at q. For the descendants u_1 of $\mathsf{dn}(q)$, including $\mathsf{dn}(q)$, it holds that $j < u_1$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. The banana subtree rooted at q in the new tree is the banana subtree rooted at j in $\mathrm{Up}(h_\sigma)$, and for nodes u_2 in this banana subtree $j < u_2$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$, since $j < i = \mathrm{Bth}(j)$. With the assumption $j < q$ it follows that the descendants u of $\mathsf{dn}(j)$ including $\mathsf{dn}(j)$ satisfy $j < u$ in the new tree. The nodes v in the banana subtree j in the new tree are descendants of maxima $t \neq j$ on a trail between p and q. By Conditions III.1, III.2 and Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$ these nodes satisfy $v < j$: the node j is the topmost node of the right trail of the banana spanned by p and q; the nodes t_1 on the left trail satisfy $t_1 < j$ by Conditions III.1 and III.2 and the nodes v_1 in the banana subtrees rooted at t_1 satisfy $v_1 < t_1 < j$ by Invariant 4.1.5; the nodes t_2 on the right trail below j are descendants of $\mathsf{dn}(j)$, and they along with their descendants v_2 satisfy $t_2 < j$ and $v_2 < j$ by Invariant 4.1.5. It follows that j satisfies the condition of Invariant 4.1.5. Since j and q satisfy the condition of Invariant 4.1.5 it follows that Invariant 4.1.5 holds in the new tree.

CASE 3.1 ($j = \mathsf{dn}(q)$ AND $h_\sigma(i) < h_\sigma(p)$): Assume $q < j$; the other case is symmetric. Note that $i = \mathrm{Bth}(j) < j$ and $p = \mathrm{Bth}(q) < q$. The in-trails and

mid-trails remain the same, with the exception that q is added to the in-trail between i and j. That is, in the new tree the nodes internal to the in-trail between i and j are $I_j \cup \{q\}$, the nodes internal to the mid-trail between i and j are M_j, the nodes internal to the in-trail between p and q are I_q and the nodes internal to the mid-trail between p and q are M_q.

To see that Invariant 4.1.5 holds we again analyze the subtrees of j and q, as in the previous cases. The banana subtree rooted at q is the same in $\mathrm{Up}(h_\sigma)$ and the new tree. The set of descendants of $\mathsf{dn}(q)$ including $\mathsf{dn}(q)$ in the new tree is a subset of that in $\mathrm{Up}(h_\sigma)$. Thus, the condition of Invariant 4.1.5 holds for q in the new tree by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. For the node j the $\mathsf{dn}(j)$ and its descendants do not change. The banana subtree rooted at j in the new tree is equal to that in $\mathrm{Up}(h_\sigma)$, with the addition of q and the banana subtree rooted at q. By assumption $q < j$ and by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$ the nodes v_1 in the banana subtree rooted at q satisfy $v_1 < q$ and thus $v_1 < j$. The remaining nodes v_2 in the banana subtree of j satisfy $v_2 < j$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. Thus, for all nodes v in the banana subtree rooted at j we have $v < j$. It follows that j satisfies the condition of Invariant 4.1.5 and that the new up-tree satisfies Invariant 4.1.5.

We now show that Invariant 4.1.7 holds. Let $i_j = \mathsf{in}(j)$, $u_q = \mathsf{up}(q)$ in $\mathrm{Up}(h_\sigma)$. Observe that q is inserted at the top of the left trail between i and j, i.e., $q = \mathsf{in}(j)$ in the new tree. By Invariant 4.1.5 and the assumption that $q < j$ it follows that $q < \mathsf{up}(q) = j$ and $q < \mathsf{dn}(q)$. Note that $u_q = \mathsf{up}(j)$ in the new tree. The inequality $j < \mathsf{dn}(j)$ holds in $\mathrm{Up}(h_\sigma)$ since j is on a left trail, and holds in the new tree since $\mathsf{dn}(j)$ does not change. If $q = \mathsf{in}(u_q)$ in $\mathrm{Up}(h_\sigma)$, then $q < u_q$ and by Invariant 4.1.5 $j < u_q$. In the new tree $j = \mathsf{in}(u_q)$ and thus $j < \mathsf{up}(j) = u_q$ and $j < \mathsf{dn}(j)$. Otherwise, if $q \neq \mathsf{in}(u_q)$ in $\mathrm{Up}(h_\sigma)$, then $u_q < q < j$. Thus, in the new tree $u_q = \mathsf{up}(j) < j < \mathsf{dn}(j)$. The inequalities $h_\theta(\mathsf{up}(q)) > h_\theta(q) > h_\theta(\mathsf{dn}(q))$ and $h_\theta(\mathsf{up}(j)) > h_\theta(j) > h_\theta(\mathsf{dn}(j))$ follow directly from Invariant 4.1.7 for $\mathrm{Up}(h_\sigma)$ and the fact that j and q are contiguous in h_θ.

CASE 3.2 ($j = \mathsf{dn}(q)$ AND $h_\sigma(i) > h_\sigma(p)$): Assume $q < j$; the other case is symmetric. Note that $i = \mathrm{Bth}(j) < j$ and $p = \mathrm{Bth}(q) < q$. The nodes q and j exchange their partners, i.e., in the new up-tree $\mathrm{Bth}(q) = i$ and $\mathrm{Bth}(j) = p$. The in-trail and mid-trail between i and j become the mid-trail and in-trail between i and p (notice that the trails change from in to mid and vice versa); the in-trail and mid-trail between p and q become the in-trail and mid-trail between p and j. Note that q is added to the mid-trail between p and j. Thus, in the new tree the nodes internal to the in-trail between p and j are I_q, the nodes internal to the mid-trail between p and j are $M_q \cup \{q\}$, the nodes internal to the in-trail between i and q are M_j and the nodes internal to the mid-trail between i and q are I_q.

The proof for Invariant 4.1.7 is identical to that of Case 3.1. The proof for Invariant 4.1.5 is similar to that of Case 3.1 and we point out the differences. The banana subtree rooted at q in the new tree is the banana subtree rooted at j in $\mathrm{Up}(h_\sigma)$. Since the nodes u in this subtree were descendants of $\mathsf{dn}(q)$ in $\mathrm{Up}(h_\sigma)$ it follows from Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$ that $q < u$. The descendants of $\mathsf{dn}(q)$ including $\mathsf{dn}(q)$ are nodes formerly in the banana subtree rooted at q, and these nodes v satisfy $v < q$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. As in the previous case, the descendants of $\mathsf{dn}(j)$ including $\mathsf{dn}(j)$ are unchanged. The banana subtree rooted at j in

the new tree is the banana subtree rooted at j in $\mathrm{Up}(h_\sigma)$ combined with the banana subtree rooted at q in $\mathrm{Up}(h_\sigma)$. Invariant 4.1.5 follows by the same argument as above.

It remains to prove that Invariant 4.1.6 holds in the new tree. Observe that $\mathsf{low}(\cdot)$ pointers only change for q and j. Consequently, we need to show that p and i satisfy $h_\theta(\mathsf{low}(\mathsf{dth}(p)) < h_\theta(p)$ and $h_\theta(\mathsf{low}(\mathsf{dth}(i))) < h_\theta(i)$. We consider the three cases above:

CASE 1 ($j = \mathsf{in}(q)$): We have $\mathsf{low}(j) = p$ in $\mathrm{Up}(h_\sigma)$ and $\mathsf{low}(j) = \mathsf{low}(q)$ in the new tree. Recall that $j = \mathsf{dth}(i)$ and $q = \mathsf{dth}(p)$ in both $\mathrm{Up}(h_\sigma)$ and in the new tree. Since $\mathrm{Up}(h_\sigma)$ satisfies Invariant 4.1.6, we know that $h_\sigma(i) > h_\sigma(\mathsf{low}(j)) = h_\sigma(p) > h_\sigma(\mathsf{low}(q))$. As $h_\sigma(u) = h_\theta(u)$ for $u \neq j, q$, it holds that $h_\theta(i) > h_\theta(\mathsf{low}(j)) = h_\theta(\mathsf{low}(q))$ and $h_\theta(p) > h_\theta(\mathsf{low}(q))$.

CASE 2 ($j = \mathsf{mid}(q)$): We have $\mathsf{low}(j) = p$ in $\mathrm{Up}(h_\sigma)$ and $\mathsf{low}(j) = \mathsf{low}(q)$ in the new tree. In the new tree $\mathsf{dth}(i) = q$ and $\mathsf{dth}(p) = j$. It holds that $h_\sigma(i) > h_\sigma(p) > h_\sigma(\mathsf{low}(q))$ by Invariant 4.1.6 for $\mathrm{Up}(h_\sigma)$ and thus $h_\theta(i) > h_\theta(\mathsf{low}(q))$. Similarly $h_\theta(p) = h_\sigma(p) > h_\sigma(\mathsf{low}(q)) = h_\theta(\mathsf{low}(q)) = h_\theta(\mathsf{low}(j))$, i.e., $h_\theta(p) > h_\theta(\mathsf{low}(j))$.

CASE 3 ($j = \mathsf{dn}(q)$): We distinguish the two cases $h_\sigma(i) < h_\sigma(p)$ and $h_\sigma(i) > h_\sigma(p)$.

$h_\sigma(i) < h_\sigma(p)$: We have $\mathsf{low}(q) = \mathsf{low}(j)$ in $\mathrm{Up}(h_\sigma)$ and $\mathsf{low}(j) \neq \mathsf{low}(q) = i$ in the new tree. Note that $\mathsf{low}(j)$ is the same in both trees. In the new tree it holds that $h_\theta(p) = h_\sigma(p) > h_\sigma(\mathsf{low}(q)) = h_\sigma(i) = h_\theta(i)$, i.e., $h_\theta(p) > h_\theta(i)$. As $\mathsf{low}(j)$ does not change and $\mathsf{dth}(i) = j$ in both trees, $h_\theta(i) > h_\theta(\mathsf{low}(j))$ in the new tree.

$h_\sigma(i) > h_\sigma(p)$: We have $\mathsf{low}(q) = \mathsf{low}(j)$ in $\mathrm{Up}(h_\sigma)$ and $\mathsf{low}(j) \neq \mathsf{low}(q) = p$ in the new tree. In the new tree we also have $\mathsf{dth}(i) = q$ and $\mathsf{dth}(p) = j$. Note again that $\mathsf{low}(j)$ is the same in both trees. In the new tree it holds that $h_\theta(p) = h_\sigma(p) > h_\sigma(\mathsf{low}(j)) = h_\theta(\mathsf{low}(j))$, since in $\mathrm{Up}(h_\sigma)$ $h_\sigma(p) > h_\sigma(\mathsf{low}(q))$ and $\mathsf{low}(j) = \mathsf{low}(q)$. Furthermore, in the new tree $h_\theta(i) = h_\sigma(i) > h_\sigma(p) = h_\theta(p)$.

In all cases p and i satisfy the required condition and thus Invariant 4.1.6 is satisfied in the new tree. We have shown that the new up-tree obtained by applying `max-interchange`to $\mathrm{Up}(h_\sigma)$ satisfies Invariants 4.1.5 to 4.1.7, which implies that it is indeed the unique up-tree for h_θ. □

INTERCHANGES OF MINIMA. Let i be a minimum and $p = \mathsf{low}(\mathsf{dth}(i))$. In an up-tree satisfying Invariant 4.1.6 the function value of i is greater than the function value of p. If this relation changes such that the function value of p becomes greater than the function value of i, Invariant 4.1.6 is violated. To fix this violation we perform an *interchange of minima*.

Let $\sigma \in [0, 1)$ such that $h_\sigma(i) > h_\sigma(\mathsf{low}(\mathsf{dth}(i))) = h_\sigma(p)$ and such that i and p are contiguous. Let $\theta \in (\sigma, 1]$ such that $h_\theta(i) < h_\theta(p)$ and such that i and p are contiguous. Write $j = \mathsf{dth}(i)$ and $q = \mathsf{dth}(p)$. We give the detailed algorithm for the case $q < p$; the other case is symmetric.

```
1  min-interchange(i, p):
2    if low(dth(i)) ≠ p then exit endif.
3    Let j = dth(i), q = dth(p).
4    Let u_j = up(j), d_j = dn(j), i_j = in(j), m_j = mid(j).
5    From q down along the trail not containing j find
6      1. first node s⁻ such that h_σ(s⁻) < h_σ(j)
7      2. last node s⁺ such that h_σ(s⁺) > h_σ(j).
8    Set dn(j) ← m_j, mid(j) ← d_j, in(j) ← s⁻, up(j) ← s⁺.
9    if q < j < p then
10     Swap in(i) and mid(i);
11     if i_j = i then mid(i_j) = u_j else up(i_j) = u_j endif;
12     if u_j = q then mid(u_j) = i_j else dn(u_j) = i_j endif;
13     if s⁺ = q then in(s⁺) = j else dn(s⁺) = j endif;
14     if s⁻ = p then in(s⁻) = j else up(s⁻) = j endif
15   else if q < p < j
16     Swap in(p) and mid(p);
17     if i_j = i then in(i_j) = u_j else up(i_j) = u_j endif;
18     if u_j = q then in(u_j) = i_j else dn(u_j) = i_j endif;
19     if s⁺ = q then mid(s⁺) = j else dn(s⁺) = j endif;
20     if s⁻ = p then in(s⁻) = j else up(s⁻) = j endif
21   endif.
22   Set dth(i) = q, dth(p) = j.
23   Set low(u) = i for all u ≠ q between u_j and q and j and q, including u_j, j.
```

Lemma 4.1.19 (Min-interchange)**.** *Let p and i be items. Let $\sigma \in [0,1)$ such that p and i are minima and contiguous in h_σ, $h_\sigma(i) > h_\sigma(p)$ and $p = \mathsf{low}(\mathsf{dth}(i))$ in $Up(h_\sigma)$. Let $\theta \in (\sigma, 1]$ such that p and i are minima and contiguous in h_θ, and $h_\theta(i) < h_\theta(p)$. Applying* `min-interchange`$(i,p)$ *to $Up(h_\sigma)$ yields the up-tree for h_θ.*

Proof. The algorithm defines $j = \mathsf{dth}(i)$, $q = \mathsf{dth}(p)$. Assume $q < p$; the other case is symmetric. We show that by applying `min-interchange`to $\mathrm{Up}(h_\sigma)$, which satisfies Invariants 4.1.5 to 4.1.7, we obtain an up-tree satisfying Invariants 4.1.5 to 4.1.7 for h_θ. By Corollary 4.1.11 this is the unique up-tree for h_θ, $\mathrm{Up}(h_\theta)$. Note that by definition of h_λ the maps h_σ and h_θ differ in either the value of i or the value of p, and are equal in the values of all other items.

In Line 5 finds two nodes on the trail between p and q that does not contain j: s^- with $h_\sigma(s^-) < h_\sigma(j)$ and s^+ with $h_\sigma(s^+) > h_\sigma(j)$, with s^- being the highest such node and s^+ being the lowest such node along the trail. Since by Invariant 4.1.7 nodes are ordered by function value along trails, s^- and s^+ are neighbors along the trail.

It may be the case that s^- is a node internal to the trail or that $s^- = p$, and similarly s^+ is either internal to the trail or $s^+ = q$. Similar statements hold for the nodes defined in Line 4:

- $u_j = \mathsf{up}(j)$ is either internal to the trail between p and q or $u_j = q$,
- $d_j = \mathsf{dn}(j)$ is either internal to the trail between p and q or $d_j = p$,
- $i_j = \mathsf{in}(j)$ is either internal to the in-trail between i and j or $i_j = i$,
- $m_j = \mathsf{mid}(j)$ is either internal to the mid-trail between i and j or $m_j = i$.

The if-statement in Line 9 distinguishes two cases: $q < j < p$ and $q < p < j$. The cases correspond to j being on the mid-trail between p and q and j being on the in-trail between p and q. As $j = \mathsf{dth}(i)$ and $p = \mathsf{low}(\mathsf{dth}(i))$ the node j must be in one of the trails between p and q. Thus, these two cases are the only cases that can occur.

We first prove Invariant 4.1.7. Recall that by Lemma 4.1.8, $\mathrm{Up}(h_\sigma)$ satisfies Conditions III.1 and III.2. We consider each node for which the $\mathsf{up}(\cdot)$ or $\mathsf{dn}(\cdot)$ pointer changes individually.

NODE j: in the new tree $\mathsf{up}(j) = s^+$ and $\mathsf{dn}(j) = m_j$. By definition of h_λ it holds that $h_\sigma(s^+) = h_\theta(s^+)$ and $h_\sigma(j) = h_\theta(j)$. As $h_\sigma(s^+) > h_\sigma(j)$ it follows that $h_\theta(\mathsf{up}(j)) > h_\theta(j)$. If $m_j \neq i$ then $h_\sigma(m_j) = h_\theta(m_j)$; if $m_j = i$ then $h_\sigma(m_j) \geq h_\theta(m_j)$ as either $h_\sigma(i) = h_\theta(i)$ or $h_\theta(i) < h_\theta(p) = h_\sigma(p)$. Thus, $h_\theta(m_j) < h_\theta(j)$. It follows that $h_\theta(\mathsf{dn}(j)) < h_\theta(j) < h_\theta(\mathsf{up}(j))$. For the other conditions of Invariant 4.1.7 we consider the two cases of Line 9 separately:

- $q < j < p$: the node j is on the mid-trail between p and q, which by Conditions III.1 and III.2 implies that $j < \mathsf{dn}(j)$ in $\mathrm{Up}(h_\sigma)$. It follows that $m_j < j$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$, since in that tree m_j is in the banana tree rooted at j. Thus, in the new tree $m_j = \mathsf{dn}(j) < j$. If $s^+ = q$, then $q = s^+ = \mathsf{up}(j) < j$. Since in this case $j = \mathsf{in}(s^+)$ (see Line 13) in the new tree j satisfies point 3 of Invariant 4.1.7. If $s^+ \neq q$, then $j < s^+$ by Conditions III.1 and III.2, as s^+ is in the in-trail between p and q. Thus $\mathsf{dn}(j) < j < \mathsf{up}(j)$ in the new tree and hence j satisfies point 2 of Invariant 4.1.7.
- $q < p < j$: the node j is on the in-trail between p and q, which by Conditions III.1 and III.2 implies that $\mathsf{dn}(j) < j$ in $\mathrm{Up}(h_\sigma)$. It follows that $j < m_j$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. Thus, in the new tree $j < \mathsf{dn}(j) = m_j$. If $s^+ = q$, then $q = s^+ = \mathsf{up}(j) < j$. Since in this case $j = \mathsf{mid}(s^+)$ (see Line 19) in the new tree j satisfies point 2 of Invariant 4.1.7. If $s^+ \neq q$, then $s^+ < j$ by Conditions III.1 and III.2, as s^+ is in the mid-trail between p and q. Thus, $\mathsf{up}(j) < j < \mathsf{dn}(j)$ in the new tree and hence j satisfies point 2 of Invariant 4.1.7.

NODE i_j: we only need to consider the case where $i_j \neq i$, as i is a minimum and does not affect Invariant 4.1.7. Since $\mathsf{dn}(i_j)$ is the same in $\mathrm{Up}(h_\sigma)$ and in the new tree, by Invariant 4.1.7 it holds that $h_\sigma(\mathsf{dn}(i_j)) < h_\sigma(i_j)$. If $\mathsf{dn}(i_j) \neq i$ then $h_\sigma(\mathsf{dn}(i_j)) = h_\theta(\mathsf{dn}(i_j))$; otherwise if $\mathsf{dn}(i_j) = i$ then $h_\theta(\mathsf{dn}(i_j)) \leq h_\sigma(\mathsf{dn}(i_j))$ by the assumptions on i and p. Hence, we have $h_\theta(\mathsf{dn}(i_j)) < h_\sigma(i_j) = h_\theta(i_j)$. In the new tree $\mathsf{up}(i_j) = u_j$. By Invariant 4.1.7 for $\mathrm{Up}(h_\sigma)$ and the definition of h_λ it holds that $h_\theta(u_j) = h_\sigma(u_j) > h_\sigma(j) > h_\sigma(i_j) = h_\theta(i_j)$. Thus, $h_\theta(\mathsf{dn}(i_j)) < h_\theta(i_j) < h_\theta(\mathsf{up}(i_j))$. For the other conditions of Invariant 4.1.7 we consider the two cases of Line 9 separately:

- $q < j < p$: j is on the mid-trail and thus by Conditions III.1 and III.2 $j < \mathsf{dn}(j)$ in $\mathrm{Up}(h_\sigma)$. By Invariant 4.1.5 it follows that in $\mathrm{Up}(h_\sigma)$ $i_j = \mathsf{in}(j) < j$ and by Invariant 4.1.7 $i_j < \mathsf{dn}(i_j)$. Also by Invariant 4.1.7 $u_j = \mathsf{up}(j) < j$ in $\mathrm{Up}(h_\sigma)$. As i_j and j are in the same subtree of u_j in $\mathrm{Up}(h_\sigma)$ Invariant 4.1.5 implies that $u_j < i_j$ since $u_j < j$. Thus, in the new tree $u_j = \mathsf{up}(i_j) < i_j < \mathsf{dn}(i_j)$.
- $q < p < j$: j is on the in-trail and thus by Conditions III.1 and III.2 $\mathsf{dn}(j) < j$ in $\mathrm{Up}(h_\sigma)$. By Invariant 4.1.5 and Invariant 4.1.7 it follows that in $\mathrm{Up}(h_\sigma)$ $j = \mathsf{up}(i_j) < i_j$ and $\mathsf{dn}(i_j) < i_j$. If $u_j = q$ then by Invariant 4.1.7

$u_j = \mathsf{up}(j) < j$ in $\mathrm{Up}(h_\sigma)$ and thus in the new tree $u_j = \mathsf{up}(j) < i_j$; in this case $i_j = \mathsf{in}(\mathsf{up}(i_j))$ in the new tree and hence i_j satisfies point 3 of Invariant 4.1.7. If $u_j \neq q$ then by Invariant 4.1.7 $u_j = \mathsf{up}(j) > j$ in $\mathrm{Up}(h_\sigma)$. As j and i_j are in the same subtree of u_j in $\mathrm{Up}(h_\sigma)$ Invariant 4.1.5 implies that $u_j > i_j$. It follows that in the new $\mathsf{dn}(i_j) < i_j < \mathsf{up}(i_j) = u_j$ and hence i_j satisfies point 2 of Invariant 4.1.7.

Node u_j: we only need to consider the case where $u_j \neq q$, as in the other case $\mathsf{up}(u_j)$ and $\mathsf{dn}(u_j)$ are not changed. Note that only $\mathsf{dn}(u_j)$ changes and that $\mathsf{up}(u_j)$ is the same in $\mathrm{Up}(h_\sigma)$ and the new tree. By Invariant 4.1.5 and the definition of h_λ we have $h_\theta(u_j) = h_\sigma(u_j) < h_\sigma(\mathsf{up}(u_j)) = h_\theta(\mathsf{up}(u_j))$. As shown for i_j we have $h_\theta(i_j) = h_\theta(\mathsf{dn}(u_j)) < h_\theta(u_j)$ in the new tree and thus $h_\theta(\mathsf{dn}(u_j)) < h_\theta(u_j) < h_\theta(\mathsf{up}(u_j))$. We now consider the two cases of Line 9:

- $q < j < p$: as shown for node i_j it holds that $u_j < \mathsf{dn}(u_j) = i_j$ in the new tree. As j is on the mid-trail between p and q in $\mathrm{Up}(h_\sigma)$ so is u_j and thus $\mathsf{up}(u_j) < u_j$. Hence, $\mathsf{up}(u_j) < u_j < \mathsf{dn}(u_j)$ in the new tree.
- $q < p < j$: recall that $u_j \neq q$. We have shown for node i_j that $\mathsf{dn}(u_j) = i_j < u_j$. As j is on the in-trail between p and q in $\mathrm{Up}(h_\sigma)$ so is u_j and thus either $q = \mathsf{up}(u_j) < u_j$ or $\mathsf{up}(u_j) > u_j$. In the first case u_j satisfies point 3 of Invariant 4.1.7 and point 2 of Invariant 4.1.7 in the new tree.

Node s^+: we only need to consider the case where $s^+ \neq q$, as in the other case $\mathsf{up}(s^+)$ and $\mathsf{dn}(s^+)$ are not changed. This implies that in the new tree $\mathsf{dn}(s^+) = j$. The details are similar to those for node j. We have seen there that $h_\theta(s^+) > h_\theta(j)$ and thus in the new tree $h_\theta(s^+) > h_\theta(\mathsf{dn}(s^+))$. As the $\mathsf{up}(s^+)$ is the same in the new tree as in $\mathrm{Up}(h_\sigma)$ it follows that $h_\theta(\mathsf{up}(s^+)) > h_\theta(s^+)$ also in the new tree. Now consider the two cases of Line 9:

- $q < j < p$: as above this implies $j = \mathsf{dn}(s^+) < s^+$. If $\mathsf{up}(s^+) = q$ then $s^+ = \mathsf{in}(\mathsf{up}(s^+))$, $\mathsf{up}(s^+) < s^+$ and hence s^+ satisfies point 3 of Invariant 4.1.7. If $\mathsf{up}(s^+) \neq q$ then $s^+ < \mathsf{up}(s^+)$ by Conditions III.1 and III.2 and hence s^+ satisfies point 2 of Invariant 4.1.7.
- $q < p < j$: as above this implies $j = \mathsf{dn}(s^+) > s^+$. As s^+ is on the mid-trail from p to q in $\mathrm{Up}(h_\sigma)$ it follows that $s^+ > \mathsf{up}(s^+)$. Thus, s^+ satisfies point 2 of Invariant 4.1.7.

Node s^-: we only need to consider the case where $s^- \neq p$, as p has a minimum and does not affect Invariant 4.1.7. This implies that in the new tree $\mathsf{up}(s^-) = j$. The pointer $\mathsf{dn}(s^-)$ remains unchanged and thus $h_\theta(\mathsf{dn}(s^-)) < h_\theta(s^-)$. As $h_\theta(s^-) = h_\sigma(s^-) < h_\sigma(j) = h_\theta(j)$ we also have $h_\theta(s^-) < h_\theta(\mathsf{up}(s^-))$. Since $s^- = \mathsf{in}(j) = \mathsf{in}(\mathsf{up}(s^-))$ in the new tree it remains to show that $\mathsf{up}(s^-) = j$ and $\mathsf{dn}(s^-)$ are on the same side of s^- in both cases of Line 9.

- $q < j < p$: j is on the mid-trail and s^- is on the in-trail between p and q, and thus by Conditions III.1 and III.2 $j < s^-$. These conditions also imply that $\mathsf{dn}(s^-) < s^-$.
- $q < p < j$: j is on the in-trail and s^- is on the mid-trail between p and q, and thus by Conditions III.1 and III.2 $s^- < j$. These conditions also imply that $s^- < \mathsf{dn}(s^-)$

In both cases s^- satisfies point 3 of Invariant 4.1.7.

All nodes for which the $\mathsf{up}(\cdot)$ or $\mathsf{dn}(\cdot)$ pointer changed satisfy the conditions of Invariant 4.1.7 in the new tree, and thus the new tree satisfies Invariant 4.1.7.

We now show that the new tree also satisfies Invariant 4.1.5. For any maximum, if the contents of the subtrees of that maximum are unchanged, then the condition of Invariant 4.1.5 continues to hold for this maximum. That is, for a maximum b, if the set of descendants of $\mathsf{dn}(b)$ is the same in $\mathrm{Up}(h_\sigma)$ and in the new tree, and if the set of nodes in the banana subtree rooted at b is the same in $\mathrm{Up}(h_\sigma)$ and in the new tree, and if $\mathrm{Bth}(b) < b$ (or $\mathrm{Bth}(b) > b$) in $\mathrm{Up}(h_\sigma)$ and in the new tree, then b satisfies the condition of Invariant 4.1.5 in the new tree, since it also satisfied it in $\mathrm{Up}(h_\sigma)$. This is the case for most of the nodes:

- $\mathrm{Bth}(q) = p$ in $\mathrm{Up}(h_\sigma)$ and $\mathrm{Bth}(q) = i$ in the new tree, but both $q < p$ and $q < i$. The set of nodes in either subtree of q remains unchanged.
- For any node on the trail from u_j to q (except u_j and q) and any node on the trail from s^+ to q (excluding s^+ and q) the set of nodes in either subtree remains unchanged, and so does the $\mathrm{Bth}(\cdot)$ of these nodes.
- The same holds for any descendant of j in $\mathrm{Up}(h_\sigma)$, for s^- and any descendant of s^-.
- For any node not in the banana subtree of q the contents of the subtrees and $\mathrm{Bth}(\cdot)$ is unchanged.

Thus, we only need to show that u_j, j and s^+ do not violate the condition of Invariant 4.1.5. Furthermore, we only need to consider the case where $u_j \neq q$ and $s^+ \neq q$.

Node u_j: the banana subtree of u_j is the same in $\mathrm{Up}(h_\sigma)$ and in the new tree, as is $\mathrm{Bth}(u_j)$. The node $\mathsf{dn}(u_j) = i_j$ in the new tree is a descendant of j in $\mathrm{Up}(h_\sigma)$ and $j = \mathsf{dn}(u_j)$ in $\mathrm{Up}(h_\sigma)$. The descendants of i_j are descendants of $\mathsf{dn}(u_j)$ in both $\mathrm{Up}(h_\sigma)$ and the new tree. As u_j does not gain any descendants, it follows that u_j satisfies the condition of Invariant 4.1.5 in the new tree.

Node s^+: the banana subtree of s^+ is the same in $\mathrm{Up}(h_\sigma)$ and in the new tree, as is $\mathrm{Bth}(s^+)$. Through the change of $\mathsf{dn}(s^+)$ to j, $\mathsf{dn}(s^+)$ gains new descendants in the new tree. These are the nodes on the trail between p and j, along with their banana subtrees, and the nodes on the trail from i to j. By Invariant 4.1.5 and Conditions III.1 and III.2 for $\mathrm{Up}(h_\sigma)$ these are all on the same side of s^+ as $\mathsf{dn}(s^+)$ in $\mathrm{Up}(h_\sigma)$. It follows that s^+ satisfies the condition of Invariant 4.1.5 in the new tree.

Node j: The two cases $q < j < p$ and $q < p < j$ are symmetric. We give the proof for the first case and assume $q < j < p$. The pointers $\mathsf{mid}(j)$ and $\mathsf{dn}(j)$ are swapped going from $\mathrm{Up}(h_\sigma)$ to the new tree (see the assignment in Line 8). The banana subtree rooted at j in the new tree consists of descendants of s^+ and the descendants of $\mathsf{dn}(j)$ in $\mathrm{Up}(h_\sigma)$. By Invariant 4.1.5 and Conditions III.1 and III.2 for $\mathrm{Up}(h_\sigma)$, each node v in this set satisfies $j < v$. Note that $\mathrm{Bth}(j) = p$ in the new tree and thus $j < \mathrm{Bth}(j)$ by assumption. The descendants of $\mathsf{dn}(j)$ in the new tree are descendants of nodes on the mid-trail of j in $\mathrm{Up}(h_\sigma)$. Each node t in this set satisfies $t < j$ by Invariant 4.1.5 for $\mathrm{Up}(h_\sigma)$. Thus, in the new tree j satisfies the condition of Invariant 4.1.5.

We have shown that all nodes satisfy the condition of Invariant 4.1.5 and thus the new up-tree satisfies Invariant 4.1.5.

It remains to show that the new up-tree also satisfies Invariant 4.1.6. Invariant 4.1.6 can be violated only by minima for which $\mathsf{low}(\mathsf{dth}(\cdot))$ changes. This changes for i, p and any node u with $\mathsf{low}(\mathsf{dth}(\cdot)) = p$ in $\mathrm{Up}(h_\sigma)$.

- In the new tree $\mathsf{low}(\mathsf{dth}(p)) = i$, and $h_\theta(p) > h_\theta(i)$ by definition of h_θ.
- As the algorithm sets $\mathsf{dth}(i) = q$ (see Line 22) in the new tree $\mathsf{low}(\mathsf{dth}(i)) = \mathsf{low}(q)$. Since either $h_\sigma(i) = h_\theta(i)$ or $h_\sigma(p) = h_\theta(p)$ and since i and p are contiguous in both h_σ and h_θ, $h_\sigma(p) > h_\sigma(\mathsf{low}(q))$ implies that $h_\theta(i) > h_\sigma(\mathsf{low}(q)) = h_\theta(\mathsf{low}(q)) = h_\theta(\mathsf{low}(\mathsf{dth}(i)))$.
- The nodes u for which we set $\mathsf{low}(\mathsf{dth}(u)) = i$ have $\mathsf{low}(\mathsf{dth}(u)) = p$ in $\mathrm{Up}(h_\sigma)$. They satisfy $h_\sigma(u) = h_\theta(u) > h_\sigma(\mathsf{low}(\mathsf{dth}(p))$, and contiguity of i and p in h_σ they satisfy $h_\theta(u) > h_\sigma(i)$. As $h_\theta(i) \leq h_\sigma(i)$, it follows that $h_\theta(u) > h_\theta(i) = h_\theta(\mathsf{low}(\mathsf{dth}(u))$.

All minima satisfy the condition of Invariant 4.1.6 and thus the new tree satisfies Invariant 4.1.6. We have shown that the new tree satisfies Invariants 4.1.5 to 4.1.7 for h_θ, which concludes the proof that it is the unique up-tree for h_θ. □

4.1.6 Correctness of Scenarios A and B

So far, we have described the local operations in terms of small changes in function value, where the involved items are contiguous in function value before and after the operation. However, Scenarios A and B described in Section 3.5.2 also need to deal with larger changes in value, involving not necessarily contiguous items. We start by describing the conditions under which a change in the value of a maximum (or a minimum by Lemma 3.5.2) does not affect the corresponding banana tree. This is stated formally in the following lemma.

Lemma 4.1.20 (Unaffected Banana Tree)**.** *Let j be a maximum. Let h_σ and h_θ be maps that differ only in the value of j such that all other items have the same criticality. If it holds that* $\max\{h_\sigma(\mathsf{in}(j)), h_\sigma(\mathsf{mid}(j)), h_\sigma(\mathsf{dn}(j))\} < h_\theta(j) < h_\sigma(\mathsf{up}(j))$ *then $\mathrm{Up}(h_\sigma)$ and $\mathrm{Up}(h_\theta)$ are identical.*

Proof. By Invariants 4.1.5 to 4.1.7, we know that $\max\{h_\sigma(\mathsf{in}(j)), h_\sigma(\mathsf{mid}(j)), h_\sigma(\mathsf{dn}(j))\} < h_\sigma(j) < h_\sigma(\mathsf{up}(j))$. The up-tree $\mathrm{Up}(h_\sigma)$ satisfies Invariant 4.1.5 for the map h_θ, since the order of items along the interval is independent of h_σ and h_θ. It also satisfies Invariant 4.1.6 for h_θ, as $h_\sigma(u) = h_\theta(u)$ for all minima u in h_σ and h_θ, and the set of minima is the same in h_σ and h_θ. Finally, it also satisfies Invariant 4.1.7, which follows from $\max\{h_\sigma(\mathsf{in}(j)), h_\sigma(\mathsf{mid}(j)), h_\sigma(\mathsf{dn}(j))\} < h_\theta(j) < h_\sigma(\mathsf{up}(j))$ and the fact that the order of items along the interval is independent of h_σ and h_θ. By Corollary 4.1.11 there is a unique banana tree for h_θ that satisfies Invariants 4.1.5 to 4.1.7, and thus $\mathrm{Up}(h_\sigma) = \mathrm{Up}(h_\theta)$. □

We now show that the algorithms given for Scenarios A and B indeed transform $\mathrm{Up}(f)$ into $\mathrm{Up}(g)$, where the maps f and g differ in the value of item j. Recall that in Scenario A item j is non-critical in f and $f(j) < g(j)$, and that in Scenario B item j is a maximum in f and decreases its value until it is non-critical in g. For Scenarios A and B we assume that j is not an endpoint,

i.e., $j \in [2, m-1]$ and we give an additional algorithm for updating the value of an endpoint below.

We restate the algorithm for Scenario A, where we now also account for updates to items that lead to an endpoint changing from down-type to up-type (see Line 2).

```
1   if g(j) > f(j+1) then
2     if j+2 is a hook then cancel or slide endpoint j+1 to j
3                                   in Up(f) and Dn(f)
4     else if f(j+1) < f(j+2) then anti-cancel j and j+1
5                                       in Up(f) and Dn(f)
6                                   else slide j+1 to j in Up(f) and Dn(f)
7     endif;
8     set q = up(j) in Up(f);
9     while f(q) < g(j) do interchange j and q in Up(f) and Dn(f);
10                              q = up(j) in Up(f)
11    endwhile;
12  endif.
```

The following lemma states that this algorithm correctly updates $\mathrm{Up}(f)$ to obtain $\mathrm{Up}(g)$.

Lemma 4.1.21 (Correctness of Scenario A)**.** *Let f and g be maps with $f(j) < g(j)$ and $f(i) = g(i)$ for all $i \neq j$, and $f(j-1) < f(j) < f(j+1)$. Applying the algorithm for Scenario A to $\mathrm{Up}(f)$ yields $\mathrm{Up}(g)$.*

Proof. We show how the up-tree $\mathrm{Up}(f)$ is modified throughout the algorithm following the homotopy h_λ until the up-tree becomes $\mathrm{Up}(g)$. If $g(j) < f(j+1)$ then item j is non-critical in f and g, $\mathrm{Up}(f) = \mathrm{Up}(g)$, and the algorithm does not change the up-tree. Assume $g(j) > f(j+1)$ for the remainder of the proof. By this assumption there exists an h_{θ_0} with $0 < \theta_0 \leq 1$ such that j and $j+1$ are contiguous in h_{θ_0}. If $j+2$ is a hook, then $j+1$ is down-type in f and up-type in h_{θ_0} and by Lemma 4.1.15 applying `cancel-or-slide-endpoint`$(j, j+1)$ yields $\mathrm{Up}(h_{\theta_0})$. If $j+2$ is not a hook and $f(j+1) < f(j+2)$, then j and $j+1$ are non-critical in f and a maximum and minimum, respectively, in h_{θ_0}. By Lemma 4.1.16 executing an anti-cancellation between j and $j+1$ yields the up-tree $\mathrm{Up}(h_{\theta_0})$. If $j+2$ is not a hook and $f(j+1) > f(j+2)$, then $j+1$ is a maximum in f, non-critical in h_{θ_0} and j is a maximum in h_{θ_0}. By Lemma 4.1.12 executing a slide from $j+1$ to j yields the up-tree $\mathrm{Up}(h_{\theta_0})$.

Let q_i be $\mathsf{up}(j)$ at the beginning of the i-th iteration of the loop in Line 9. We show by induction that after every iteration i of the loop either (1) the current up-tree corresponds to a map $h_{\theta_{i+1}}$ where q_i and j are contiguous and $h_{\theta_{i+1}}(q_i) < h_{\theta_{i+1}}(j)$ or (2) the current up-tree is $\mathrm{Up}(g)$. For the base case $i = 1$, if $f(q_1) > g(j)$, then the up-tree $\mathrm{Up}(h_{\theta_0}) = \mathrm{Up}(g)$ by Lemma 4.1.20. Otherwise, there exists a $\theta_1 \in (\theta_0, 1]$ such that $h_{\theta_1}(q_1) < h_{\theta_1}(j)$ and q_1 and j are contiguous in h_{θ_1}. By Lemma 4.1.20 and Lemma 4.1.18 interchanging q_1 and j yields $\mathrm{Up}(h_{\theta_1})$. Now assume that at the beginning of some iteration i the current up-tree is $\mathrm{Up}(h_{\theta_{i-1}})$, with $h_{\theta_{i-1}}(j) \in [\max\{f(\mathsf{in}(j)), f(\mathsf{mid}(j)), f(\mathsf{dn}(j))\}, f(q_i)]$. If $f(q_i) > g(j)$, then the up-tree $\mathrm{Up}(h_{\theta_i}) = \mathrm{Up}(g)$ by Lemma 4.1.20. Else, there exists a $\theta_{i+1} \in (\theta_i, 1]$ such that $h_{\theta_{i+1}}(q_i) < h_{\theta_{i+1}}(j)$ and q_i and j are contiguous in $h_{\theta_{i+1}}$. Again, interchanging q_i and j yields the up-tree $\mathrm{Up}(h_{\theta_{i+1}})$.

There exists an iteration I where $f(q_I) > g(j)$, at the latest when q_I is the special root, and the loop terminates. Then, by Lemma 4.1.20 the $\mathrm{Up}(h_{\theta_{I-1}}) = \mathrm{Up}(g)$. □

The following algorithm updates the banana trees in Scenario B, now including updates that change up-type items to down-type items (see Line 5).

```
1  loop q = arg max{f(dn(j)), f(in(j)), f(mid(j))} in Up(f);
2    if f(q) > f(j+1) then interchange q and j in Up(f) and Dn(f)
3                     else exit endif
4  forever;
5  if j+2 is a hook then cancel or slide endpoint j+1 to j
6                        in Up(f) and Dn(f)
7  else if f(j+1) < f(j+2) then cancel j with j+1 in Up(f) and Dn(f)
8                          else slide j to j+1 in Up(f) and Dn(f)
9  endif.
```

The next lemma states that this algorithm correctly updates $\mathrm{Up}(f)$ to obtain $\mathrm{Up}(g)$.

Lemma 4.1.22 (Correctness of Scenario B)**.** *Let f and g be maps with $f(j) > g(j)$ and $f(i) = g(i)$ for all $i \neq j$, and $f(j-1) < f(j) > f(j+1)$. Applying the algorithm for Scenario B to $\mathrm{Up}(f)$ yields $\mathrm{Up}(g)$.*

Proof. The proof is analogous to that of Lemma 4.1.21: after each iteration i of the loop, there is a θ_i such that q_i and j are contiguous in h_{θ_i} and the current up-tree is $\mathrm{Up}(h_{\theta_i})$. Let $\mathrm{Up}(h_\gamma)$ be the up-tree after the loop, where in h_γ the items j and q are contiguous in value. If $j+2$ is a hook, then $j+1$ is an up-type endpoint in h_γ and a down-type endpoint in g, since $g(j) < g(j+1)$. Then, by Lemma 4.1.15 and the fact that non-critical items are not represented in the up-tree applying `cancel-or-slide-endpoint`$(j, j+1)$ to $\mathrm{Up}(h_\gamma)$ yields $\mathrm{Up}(g)$. Similarly, if $j+2$ is not a hook and $f(j+1) < f(j+2)$, then j and $j+1$ are non-critical in g as $f(j-1) < g(j) < f(j+1) < f(j+2)$, and canceling j with $j+1$ yields $\mathrm{Up}(g)$ by Lemma 4.1.14. Finally, if $j+2$ is not a hook and $f(j+1) > f(j+2)$, then $j+1$ was non-critical in f and is a maximum in g, and sliding j to $j+1$ yields $\mathrm{Up}(g)$ by Lemma 4.1.12. □

To conclude, we state an algorithm for updating the banana trees under the change of value of the endpoint m. The algorithm for the other endpoint of the interval is symmetric. We assume that m is up-type in f and down-type in g, i.e., $f(m) < f(m-1) < g(m)$. There is also the reverse case, in which m is down-type in f and up-type in g.

```
1  Set q = dn(m) in Dn(f);
2  while q is a maximum in −f and f(q) < g(m) do
     interchange m and q in Up(f) and Dn(f);
3    q = dn(m) in Dn(f)
4  endwhile;
5  if f(m−1) < g(m) then cancel or slide endpoint m to m−1 endif;
6  Set q = up(m) in Up(f);
7  while f(q) < g(m) do interchange m and q in Up(f) and Dn(f);
8                       q = up(m) in Up(f)
9  endwhile.
```

In the first loop we exploit the coupling of interchanges between the up-tree and down-tree: we find the maximum with which to interchange m in $\mathrm{Dn}(f)$, and perform the corresponding interchange of minima in $\mathrm{Up}(f)$. The purpose

of this loop is to prepare the banana tree for switching m from an up-type to a down-type item, as described in Section 4.1.3. The value of the hook $m+1$ is defined to have value $f(m+1) = m+\varepsilon$ and $g(m+1) = g(m)-\varepsilon$, i.e., its value is adjusted alongside that of m, which justifies interchanging m only with $\mathsf{dn}(m)$ in the first loop, rather than with $\mathsf{in}(m)$, $\mathsf{mid}(m)$ and $\mathsf{dn}(m)$ as we have done in Scenario B. For completeness we state in the next lemma that this algorithm correctly maintains the up-tree; we omit the proof, which is analogous to that of Lemmas 4.1.21 and 4.1.22.

Lemma 4.1.23 (Local Change at an Endpoint)**.** *Let f and g be maps differing only in the value of the endpoint m, such that $f(m) < f(m-1) < g(m)$. Applying the previous algorithm to $Up(f)$ yields $Up(g)$.*

4.2 Correctness of Topological Maintenance

4.2.1 Splitting Banana Trees

In this section we prove that the algorithm Split described in Section 3.5.3 correctly splits a banana tree. The proof will be by induction over the iterations of the loop in Split, with the base case in Lemma 4.2.1 and the induction step in Lemma 4.2.2. We first prove these two lemmas and then state the result in Theorem 4.2.3.

Given a list of items, its associated map f and an item ℓ, the algorithm `Split` computes from $\mathrm{Up}(f)$ two new up-trees $\mathrm{Up}(g)$ and $\mathrm{Up}(h)$, where the former contains the items up to ℓ and the latter the remaining items. We define x to be the midpoint of ℓ and $\ell+1$ with $f(x) = \frac{1}{2}(f(\ell)+f(\ell+1))$. We write $Stack_i$, p_i, q_i for $Stack$, p, q returned by TopBanana in the i-th iteration of Split. The up-trees $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$ denote the left and right up-trees after the i-th iteration, with initial up-trees $\mathrm{Up}(g_0)$, $\mathrm{Up}(h_0)$. One of the initial up-trees consists of the new special root and the dummy leaf α. We assume that this is the tree $\mathrm{Up}(h_0)$ with special root β_h and that $\mathrm{Up}(g_0) = \mathrm{Up}(f)$. This is equivalent to assuming that the top banana on the stacks is on the right spine of $\mathrm{Up}(f)$, where the top banana is the banana closest to the root such that x lies in some panel of the corresponding window. The other case where the top banana is on the left spine of $\mathrm{Up}(f)$ is symmetric. We further define α such that it is the rightmost item of $\mathrm{Up}(g_i)$ if it is in $\mathrm{Up}(g_i)$ and the leftmost item of $\mathrm{Up}(h_i)$ if it is in $\mathrm{Up}(h_i)$. Finally, let I be the total number of bananas on the stack, i.e., the number of iterations of Split.

Lemma 4.2.1 (Base Case)**.** *At the beginning of the first iteration it holds that*

1. *for all nodes u in $Up(g_0)$ and all nodes v in $Up(h_0)$ we have $u < v$,*
2. *$Up(g_0)$ and $Up(h_0)$ satisfy Invariants 4.1.5 to 4.1.7,*
3. *any node u that is not on any stack and does not have an ancestor on any stack is in $Up(g_0)$ if $u < x$ and in $Up(h_0)$ if $u > x$,*
4. *$f(\alpha) < f(x)$, $f(q_j) < f(\mathsf{in}(\alpha))$ and $f(q_j) < f(\mathsf{mid}(\alpha))$ at the beginning of the iteration for all $j \in [1, I]$ with $(p_j, q_j) \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$,*
5. *all bananas $(p_j, q_j) \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$ for $j \in [1, I]$ are in $Up(g_0)$ if α is in $Up(h_0)$ and in $Up(h_1)$ if α is in $Up(g_1)$,*

6. *if* $Stack_1 \in \{L_{\mathrm{dn}}, R_{\mathrm{dn}}\}$ *then* q_1 *is in the same tree as* α,
7. *the in-trail between* α *and* $\mathsf{dth}(\alpha)$ *is empty and* α *is on the spine.*

Proof. Assume that $\mathrm{Up}(g_0) = \mathrm{Up}(f)$. Claim 1 holds by definition of α and the special root of $\mathrm{Up}(h_0)$. The up-tree $\mathrm{Up}(f) = \mathrm{Up}(g_0)$ is the input to the algorithm and satisfies Invariants 4.1.5 to 4.1.7 by assumption. The topmost banana (p_1, q_1) has q_1 on the right spine of $\mathrm{Up}(g_0)$ and $q_1 < x$, which implies that nodes u that are not descendants of q_1 satisfy $u < q_1 < x$. These are precisely the nodes in $\mathrm{Up}(g_0)$ that have no ancestor on any stack. The nodes in $\mathrm{Up}(h_0)$ also have no ancestor on any stack, but $\alpha > x$ by definition, and so is the special root of $\mathrm{Up}(h_0)$. This implies claim 3. The inequality $f(\alpha) < f(x)$ follows from $f(\alpha) < f(p_1) < f(x)$, where the last inequality holds since x is in the in-panel or mid-panel of $W(p_1, q_1)$. The pointers $\mathsf{in}(\alpha)$ and $\mathsf{mid}(\alpha)$ both point to the special root of $\mathrm{Up}(h_0)$, which by definition has greater value than all q_j for $j \in [1, I]$ with $q_j \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$. Thus, all three inequalities of claim 4 hold. To see that claim 5 holds simply note that all nodes on the stacks are in $\mathrm{Up}(g_0)$ and α is in $\mathrm{Up}(h_0)$. Claim 6 holds trivially, as the topmost banana (p_1, q_1) cannot be on L_{dn} or R_{dn}: this would put x in the in-panel or mid-panel of the banana that (p_1, q_1) is nested in, which contradict (p_1, q_1) being the topmost banana. Finally, α is the only other node in $\mathrm{Up}(h_0)$ besides the special root, and thus both trails between α and $\mathsf{dth}(\alpha)$ are empty and α is on the spine. □

Lemma 4.2.2 (Inductive Step)**.** *If at the beginning of any iteration* i *of* SPLIT *it holds that*

1. *for all nodes* u *in* $Up(g_{i-1})$ *and all nodes* v *in* $Up(h_{i-1})$ *we have* $u < v$,
2. $Up(g_{i-1})$ *and* $Up(h_{i-1})$ *satisfy Invariants 4.1.5 to 4.1.7,*
3. *any node* u *that is not on any stack and does not have an ancestor on any stack is in* g_{i-1} *if* $u < x$ *and in* h_{i-1} *if* $u > x$,
4. $f(\alpha) < f(x)$, $f(q_j) < f(\mathsf{in}(\alpha)$ *and* $f(q_j) < f(\mathsf{mid}(\alpha))$ *at the beginning of the iteration for all* $j \in [i, I]$ *with* $(p_j, q_j) \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$,
5. *all bananas* $(p_j, q_j) \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$ *for* $j \in [i, I]$ *are in* $Up(g_i)$ *if* α *is in* $Up(h_i)$ *and in* $Up(h_i)$ *if* α *is in* $Up(g_i)$,
6. *if* $Stack_i \in \{L_{\mathrm{dn}}, R_{\mathrm{dn}}\}$ *then* q_i *is in the same tree as* α,
7. *the in-trail between* α *and* $\mathsf{dth}(\alpha)$ *is empty and* α *is on the spine,*

then after the i*-th iteration*

1. *for all nodes* u *in* $Up(g_i)$ *and all nodes* v *in* $Up(h_i)$ *we have* $u < v$,
2. $Up(g_i)$ *and* $Up(h_i)$ *satisfy Invariants 4.1.5 to 4.1.7,*
3. *any node* u *that is not on any stack and does not have an ancestor on any stack is in* g_i *if* $u < x$ *and in* h_i *if* $u > x$,
4. $f(\alpha) < f(x)$, $f(q_j) < f(\mathsf{in}(\alpha))$ *and* $f(q_j) < f(\mathsf{mid}(\alpha))$ *for all* $j \in [i+1, I]$ *with* $(p_j, q_j) \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$,
5. *all bananas* $(p_j, q_j) \in L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$ *for* $j \in [i+1, I]$ *are in* $Up(g_i)$ *if* α *is in* $Up(h_i)$ *and in* $Up(h_i)$ *if* α *is in* $Up(g_i)$,

6. *if* TopBanana *returns* $Stack_{i+1} \in \{L_{\mathrm{dn}}, R_{\mathrm{dn}}\}$ *then* q_{i+1} *is in the same tree as* α,
7. *the in-trail between* α *and* $\mathsf{dth}(\alpha)$ *is empty and* α *is on the spine.*

Proof. There are in total six cases:

1. $Stack = L_{\mathrm{up}}$,
2. $Stack = R_{\mathrm{up}}$,
3. $Stack = M_{\mathrm{up}}$ and $q_i < x < p_i$,
4. $Stack = M_{\mathrm{up}}$ and $p_i < x < q_i$,
5. $Stack = L_{\mathrm{dn}}$,
6. $Stack = R_{\mathrm{dn}}$.

The first two cases are symmetric, as are the two cases with $Stack = M_{\mathrm{up}}$ and the last two cases. We prove the claim for cases 1, 3 and 5. In the following we assume that the seven conditions hold.

Case 1 ($Stack_i = L_{\mathrm{up}}$)**:** Split executes DoInjury. By definition of L_{up} $q_i < p_i < x$. The node q_i is on a spine, as otherwise the banana it is nested in would contain x in its in- or mid-panel. This is not possible, since this banana would then be on the stacks above (p_i, q_i). Furthermore, $q_i < p_i$ implies that q_i is on a right spine and it is thus in the left tree. The fourth claim follows immediately from the fourth assumption, as the value of α remains unchanged and L_{up}, R_{up}, M_{up} can be merged into a sorted stack by Lemma 3.5.4.

Nodes j with $j > x$ on the in-trail between p_i and q_i are removed and inserted between α and $\mathsf{mid}(\alpha)$. These nodes and the nodes in their banana subtrees are the rightmost nodes of $\mathrm{Up}(g_i)$, which implies claim 1. Since we do not modify the in-trail between α and $\mathsf{dth}(\alpha)$ and it is assumed to be empty at the beginning of the iteration, this trail is also empty after the iteration, which proves claim 7.

Denote by j^- the lowest and highest among the moved nodes in terms of function value, respectively. Conditions III.1 and III.2 hold in $\mathrm{Up}(g_{i-1})$ and $\mathrm{Up}(h_{i-1})$, and thus the nodes j form a contiguous section of the in-trail with $j^+ = \mathsf{in}(q_i)$. Write a for the highest node on the in-trail between p_i and q_i that is not moved. Both the in-trail originally containing the nodes j and the mid-trail between α and $\mathsf{dth}(\alpha)$ are right trails. By the assumption that nodes in $\mathrm{Up}(g_i)$ are less than nodes in $\mathrm{Up}(h_i)$ it follows that $\alpha = \mathsf{dn}(j^-) < j^- < \cdots < j^+ < \mathsf{up}(j^+)$. The item x is in the in-panel of the window $W(p_i, q_i)$ corresponding to the banana (p_i, q_i) in $\mathrm{Up}(g_{i-1})$, and by Conditions III.1, III.2 and Invariant 4.1.7 $x < j^-$ also implies $f(x) < f(j^-)$. This, together with the fourth assumption implies that $f(\alpha) = f(\mathsf{dn}(j^-)) < f(x) < f(j^-) < \cdots < f(j^+) < f(\mathsf{up}(j^+))$. Thus, Invariant 4.1.7 holds in the new trees.

Since nodes pointed to by $\mathsf{low}(\mathsf{dth}(\cdot))$ only change for $\mathrm{Bth}(j)$ where j is one of the moved nodes, namely from p_i to α, Invariant 4.1.6 holds in the new trees by the assumption that $f(\alpha) < f(p_i)$.

Invariant 4.1.5 continues to hold in the tree that originally contained the nodes j, since nodes are only removed from subtrees. The nodes j themselves only gain α as a descendant of $\mathsf{dn}(j)$ or as $\mathsf{dn}(j)$ itself. In addition, $\alpha < j$ by definition of α and $j < \mathrm{Bth}(j)$ by definition of right trail. Thus, the nodes j also satisfy the condition of Invariant 4.1.5. The nodes j are inserted as

descendants of $\mathsf{dn}(t)$, where $t = \mathsf{mid}(\alpha)$ before the insertion. Since $j < \mathsf{dn}(t)$ by assumption Item 1, the condition of Invariant 4.1.5 continues to holds for t, and also holds for the remaining nodes of this tree. It follows that both trees satisfy Invariant 4.1.5 after the i-th iteration.

The nodes u in $\mathrm{Up}(h_{i-1})$ that are not on any stack and do not have an ancestor on any stack satisfy $u > x$. The nodes j and the nodes in their banana subtrees are also greater than x. Thus, $\mathrm{Up}(h_{i-1})$ contains no node u that is not on any stack and has no ancestor on any stack with $u < x$. In the left tree $\mathrm{Up}(g_i)$ the node a and the nodes in its subtree are the rightmost nodes of a, since a is on the right spine. Other nodes in $\mathrm{Up}(g_i)$ are thus less than x. Note that $a < x$ by definition of a. The node a is either a minimum, a maximum and not on any stack, or a maximum and on some stack. In the first two cases all nodes in $\mathrm{Up}(g_i)$ are less than x. In the third case, all nodes except possibly descendants of a are less than x, and descendants of a have an ancestor on the stack. Thus, claim 3 holds.

To see that claim 5 holds, recall that the bananas in L_{up}, R_{up} and M_{up} are nested into each other. Call the topmost banana on these stacks (p_ℓ, q_ℓ). It must be nested into (p_i, q_i) and is thus on the in-trail between p_i and q_i in $\mathrm{Up}(g_i)$, as otherwise x would be in the mid-panel of $W(p_i, q_i)$, which contradicts $Stack_i = L_{\mathrm{up}}$. This implies $q_i < p_i$ and $q_i < x$. It follows that q_i remains in the left tree, while α remains in the right tree. This proves claim 5.

We now prove claim 6. Assume that $Stack_{i+1} = L_{\mathrm{dn}}$. Then, by definition $p_{i+1} < q_{i+1}$. Thus, q_{i+1} cannot be on the in-trail between p_i and q_i. It can also not be on any other trail, since then $q_{i+1} < p_i < x$, which implies that x is not in the out-panel of $W(p_{i+1}, q_{i+1})$. It follows that p_{i+1} and q_{i+1} cannot be in the left tree. Now assume that $Stack_{i+1} = R_{\mathrm{dn}}$. Then, by definition $q_{i+1} < p_{i+1}$. If q_{i+1} is in $\mathrm{Up}(g_i)$, then $q_{i+1} < x$, as all maxima greater than x have been moved to $\mathrm{Up}(h_i)$. This contradicts $W(p_{i+1}, q_{i+1})$ having x in its out-panel, which implies that q_{i+1} must be in the right tree. In both cases, q_{i+1} is in the right tree, which is the same tree as α.

Case 3 ($Stack_i = M_{\mathrm{up}}$ and $q_i < x < p_i$): Split executes DoFatality. By definition of M_{up} $q_i < x < p_i$. This case is similar to the case $Stack_i = L_{\mathrm{up}}$ and we mainly point out the differences. As above, q_i is on the left spine of $\mathrm{Up}(g_{i-1})$. Since x is in a panel of the window (p_i, q_i) it holds that $f(p_i) < f(x)$. The fourth claim then follows immediately from the third assumption, as $f(\alpha) < f(p_i)$ after the iteration and L_{up}, R_{up}, M_{up} can be merged into a sorted stack by Lemma 3.5.4.

All nodes internal to the in-trail between p_i and q_i are moved to the right tree, as are the nodes j with $j > x$ on the mid-trail between p_i and q_i. The node α replaces p_i as $\mathsf{Bth}(q_i)$. As q_i is on a spine the nodes that are moved to the right tree are the rightmost nodes of $\mathrm{Up}(g_{i-1})$, and together with assumption 1 this implies claim 1, i.e., that the nodes in $\mathrm{Up}(h_i)$ are all greater than the nodes in $\mathrm{Up}(g_i)$. The in-trail between α and $\mathsf{dth}(\alpha)$ in $\mathrm{Up}(g_i)$ is the in-trail between α and q_i, which is empty as all nodes on the in-trail beginning at q_i have been moved to $\mathrm{Up}(h_i)$.

Invariants 4.1.5 and 4.1.7 hold in $\mathrm{Up}(g_i)$ as $f(\alpha) < f(p_i) < f(q_i)$ by assumption 4 and because the order of maxima and nodes in their subtrees is unchanged. By the assumption that $f(q_i) < f(\mathsf{in}(\alpha))$ and $f(q_i) < f(\mathsf{mid}(\alpha))$, where $\mathsf{in}(\alpha)$ and $\mathsf{mid}(\alpha)$ are the pointers in $\mathrm{Up}(h_{i-1})$, the nodes j that are moved to the right tree satisfy $f(j) < f(\mathsf{up}(j))$. Those nodes j that are on a left trail in $\mathrm{Up}(g_{i-1})$ are on a left trail in $\mathrm{Up}(h_i)$, and those on a right trail are also on a right trail in $\mathrm{Up}(h_i)$. It follows that $\mathrm{Up}(h_i)$ also satisfies Invariants 4.1.5

and 4.1.7. In $\mathrm{Up}(g_i)$, only for nodes with $\mathsf{low}(\mathsf{dth}(\cdot)) = p_i$ does this pointer change, namely to α, and $f(\alpha)$ is set to $f(p_i) - \varepsilon$. These nodes satisfy the condition of Invariant 4.1.6, as does α itself. In $\mathrm{Up}(h_i)$ the node p_i satisfies the condition of Invariant 4.1.6, as $f(p_i) > f(\alpha)$. The nodes that are moved to the right tree also satisfy this condition, since $\mathsf{low}(\mathsf{dth}(\cdot))$ remains unchanged. Other nodes u with $\mathsf{low}(\mathsf{dth}(u)) = p_i$ in $\mathrm{Up}(g_i)$ were already on the mid-trail between α and $\mathsf{dth}(\alpha)$ in $\mathrm{Up}(h_{i-1})$. If $f(\mathsf{low}(\mathsf{dth}(u))) < f(p_i)$ then the windows $W(u, \mathsf{dth}(u))$ would have x in their out-panel, which places them in L_{dn} or R_{dn}. But by assumption 4 $f(\mathsf{dth}(u)) > f(q_i)$ which implies that they would have been processed in an earlier iteration. This is a contradiction and thus $f(\mathsf{low}(\mathsf{dth}(u))) > f(p_i)$, i.e., they satisfy the condition of Invariant 4.1.6. For no other node of $\mathrm{Up}(h_i)$ does the node $\mathsf{low}(\mathsf{dth}(\cdot))$ change from what it was in $\mathrm{Up}(h_{i-1})$, so Invariant 4.1.6 holds in $\mathrm{Up}(h_i)$. This proves claim 2.

For claim 3 observe that nodes u in $\mathrm{Up}(g_i)$ satisfy $u < x$, as any nodes with $u > x$ are moved to h_i. Nodes v with $v < x$ in $\mathrm{Up}(h_i)$ have as ancestor the highest node that is moved from the mid-trail; if such nodes v exist, then this highest node itself must be in either R_{up} or M_{up}, and thus the only nodes in $\mathrm{Up}(h_i)$ with $v < x$ have an ancestor on some stack. This highest node can also be the only next node in $L_{\mathrm{up}} \cup R_{\mathrm{up}} \cup M_{\mathrm{up}}$, so claim 5 follows.

To see that claim 6 holds note that the next banana with x in the out-panel of the corresponding window can only be on the mid-trail between α and q_i in $\mathrm{Up}(g_i)$, similar to the case $Stack_i = L_{\mathrm{up}}$.

Case 5 ($Stack_i = L_{\mathrm{dn}}$): SPLIT executes DOSCARE, which sets $f(\alpha) = f(p_i) + \varepsilon$ and then executes an interchange of minima between α and p_i. The nodes q_i and α are in the same tree by assumption, and they must be in $\mathrm{Up}(g_{i-1})$, as x is in the out-panel of $W(p_i, q_i)$ and the in-trail between α and $\mathsf{dth}(\alpha)$ is assumed to be empty. Going from $\mathrm{Up}(g_{i-1})$ to $\mathrm{Up}(g_i)$ and from $\mathrm{Up}(h_{i-1})$ to $\mathrm{Up}(h_i)$ does not change the set of nodes contained in each tree. In fact, $\mathrm{Up}(h_{i-1}) = \mathrm{Up}(h_i)$. Claims 1 and 5 follow immediately from the respective assumption. The set of nodes internal to the in-trail between α and $\mathsf{dth}(\alpha)$ after the interchange of minima is a subset of the nodes internal to the in-trail between α and $\mathsf{dth}(\alpha)$ before the iteration. The latter is empty by assumption 7. Furthermore, α is on the spine before iteration i by 7 and it remains on the spine through the interchange, so the claim 7 holds.

We now show claim 2. Note that q_i must be on the mid-trail between $\mathsf{dth}(\alpha)$ and α, i.e., $\mathsf{low}(q_i) = \alpha$, as otherwise x would not be in the out-panel of $W(p_i, q_i)$. We first show that there is no other minimum with value between $f(\alpha)$ and $f(p_i)$ whose $\mathsf{dth}(\cdot)$ pointer points to a maximum on the banana spanned by α and $\mathsf{dth}(\alpha)$. The in-trail between α and $\mathsf{dth}(\alpha)$ is empty by assumption. Let s be a minimum with $f(\alpha) < f(s) < f(p_i)$ and with $\mathsf{dth}(s)$ on the mid-trail between α and $\mathsf{dth}(\alpha)$. The banana $(s, \mathsf{dth}(s))$ has x in its out-panel and $s < \mathsf{dth}(s)$, and thus must be on the stack L_{dn}. It must be on L_{dn} below (p_i, q_i), as otherwise it would have been processed before and $f(\alpha) > f(s)$ at the beginning of the iteration. Since L_{dn} is sorted by Lemma 3.5.4 this implies $f(s) > f(p_i)$, which is a contradiction. It follows that there is no other minimum that could be interchanged with α in place of p_i. Setting the value of α to $f(p_i) + \varepsilon$ leads to a violation of Invariant 4.1.6, but this is fixed by the interchange of minima. Thus, $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$ satisfy Invariants 4.1.5 to 4.1.7.

To prove claim 3 we consider the nodes which no longer have an ancestor on any stack or are no longer on any stack themselves. The nodes no longer on any stack are p_i and q_i, which were taken of L_{dn}. These satisfy $p_i < x$ and $q_i < x$

by definition of L_{dn}. The remaining nodes to consider are nodes with ancestor q_i. Descendants u of q_i that were in the banana subtree of q_i in $\text{Up}(g_{i-1})$ satisfy $u < q_i$ by Invariant 4.1.5 for $\text{Up}(g_{i-1})$. The remaining descendants have an ancestor on the mid-trail between α and q_i in the new tree $\text{Up}(g_i)$. The window spanned by this ancestor and its birth has x in its out-panel and is thus on L_{dn}.

The interchange of minima does not modify $\text{mid}(\alpha)$, but does modify $\text{in}(\alpha)$. More precisely, $\text{in}(\alpha) = q_i$ after the interchange. By assumption 4 and Lemma 3.5.4 it follows for all $j \in [i+1, I]$ with $(p_j, q_j) \in L_{\text{up}} \cup R_{\text{up}} \cup M_{\text{up}}$ that $f(q_j) < f(\text{in}(\alpha))$ and $f(q_j) < f(\text{mid}(\alpha))$. By Lemma 3.5.4 and for all $j \in [i+1, I]$ with $(p_j, q_j) \in L_{\text{up}} \cup R_{\text{up}} \cup M_{\text{up}}$ it holds that $f(q_j) < f(q_i)$, which implies $f(q_j) < f(\text{in}(\alpha))$. The value ε is defined to be smaller than the difference between the values of any two items, and thus there is no item with value in $[f(p_i), f(\alpha)]$. Because x is in the out-panel of $W(p_i,,)q_i$ it holds that $f(p_i) < f(x)$ and thus $f(\alpha) < f(x)$.

For the sixth claim we first prove that $Stack_{i+1} \neq R_{\text{dn}}$. If $Stack_{i+1} \in R_{\text{dn}}$, then $x < q_{i+1} < p_{i+1}$ and x is in the out-panel of $W(p_{i+1}, q_{i+1})$ by definition of R_{dn}. This implies that q_{i+1} cannot be on the same trail as q_i and in order for x to be in the out-panel of $W(p_{i+1}, q_{i+1})$ it must in fact be either in the in-trail between α and $\text{dth}(\alpha)$, or in a right spine of the right subtree. The former is impossible, as the in-trail between α and $\text{dth}(\alpha)$ is empty. Now assume that q_{i+1} is on a right trail in the right up-tree. Then $p' = \text{low}(q_{i+1}) < q_{i+1}$ and $\text{dth}(p')$ must be on the left spine of the right tree, as otherwise x would not be in the out-panel of $W(p_{i+1}, q_{i+1})$. This implies that $p' < x < \text{dth}(p')$, which in turn implies that $(p', \text{dth}(p')) \in M_{\text{up}}$. As M_{up} and R_{dn} are sorted and $f(p') < f(p_{i+1})$ by Invariant 4.1.6, this implies that $Stack_{i+1}$ cannot be R_{dn}. Thus, if $Stack_{i+1} \in \{L_{\text{dn}}, R_{\text{dn}}\}$, then $Stack_{i+1} = L_{\text{dn}}$. In this case $p_{i+1} < q_{i+1} < x$, which implies that q_{i+1} must be in the left tree and thus in the same tree as α.

This concludes the proof of the inductive step. □

Theorem 4.2.3 (Correctness of SPLIT)**.** *Given a list of m items and corresponding map f, and an item ℓ with $2 \leq \ell \leq m-1$, SPLIT splits the up-tree $\text{Up}(f)$ into two up-trees $\text{Up}(g)$ and $\text{Up}(h)$, where $\text{Up}(g)$ contains nodes u with $u \leq \ell$ and $\text{Up}(h)$ contains the nodes $u \geq \ell$.*

Proof. Recall that we defined x to be the midpoint between ℓ and $\ell + 1$. The proof is by induction over the iterations of the loop in SPLIT, taking Lemma 4.2.1 as the base case and Lemma 4.2.2 as the induction step. This proves the invariants (i) and (ii) introduced in Section 3.5.3, which imply the theorem. □

4.2.2 Gluing Two Banana Trees

In this section we prove that the algorithm GLUE presented in Section 3.5.3 correctly glues two banana trees $\text{Up}(g)$ and $\text{Up}(h)$ into $\text{Up}(f)$, where $f = g \cdot h$. First, we show how to pre-process the functions g and h to obtain the case where g ends in an up-type item on the right, h begins with a down-type item on the left and this endpoint of g has lower value than the endpoint of h. Write ℓ for the rightmost item of g and ℓ' for the leftmost item of h. We assume $f(\ell) < f(\ell')$. In the case $f(\ell) > f(\ell')$ we obtain the symmetric case to the one described in section Section 3.5.3 and the algorithm is symmetric. There are four cases depending on whether ℓ and ℓ' are up-type or down-type items: (1) ℓ and ℓ' are up-type; (2) ℓ is up-type and ℓ' is down-type; (3) ℓ is down-type

and ℓ' is up-type; (4) ℓ and ℓ' are down-type. Case (2) is the one described in Section 3.5.3 and we show how to reduce the other cases to this case.

(1) $\rightarrow$ (2): ℓ' becomes non-critical in $g \cdot h$ and we replace it by the dummy leaf α.

(3) $\rightarrow$ (2): ℓ and ℓ' become non-critical in $g \cdot h$. Since ℓ is down-type it is paired with a hook in $\mathrm{Up}(g)$ and we simply remove it along with the hook. As in the previous case we replace ℓ' by the dummy leaf α.

(4) $\rightarrow$ (2): ℓ becomes non-critical in $g \cdot h$. As in the previous ℓ is paired with a hook in $\mathrm{Up}(g)$ and we remove it along with the hook.

Whenever ℓ' is a down-type item it is paired with a hook and we replace the hook by the dummy leaf α. In all cases we begin with the situation described in Section 3.5.3.

For the remainder of the section we assume that we are in case (2), and we now prove that the algorithm Glue yields the tree $\mathrm{Up}(f)$. The proof will be by induction over the iterations of the loop, with the base case and induction step in Lemmas 4.2.4 and 4.2.5, respectively. We combine the two and show that the algorithm terminates correctly in Theorem 4.2.6. Write $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$ for the left and right tree after iteration i, respectively, with $\mathrm{Up}(g_0) = \mathrm{Up}(g)$ and $\mathrm{Up}(h_0) = \mathrm{Up}(h)$. Define α to be greater than the other nodes of $\mathrm{Up}(g_i)$ and less than the other nodes of $\mathrm{Up}(h_i)$. Furthermore, we write p_i, $\hat{p}_i$, q_i, q_i' for p, $\hat{p}$, q, q' in the i-th iteration.

Lemma 4.2.4 (Base Case). *At the beginning of the first iteration it holds that*

1. *$Up(g_0)$ and $Up(h_0)$ satisfy Invariants 4.1.5 to 4.1.7,*
2. *the in-trail between α and $\mathsf{dth}(\alpha)$ is empty, $\alpha \in \{Bth(q_1), Bth(q_1')\}$, $f(\alpha) > f(\hat{p}_1)$, $f(\alpha) > f(p_1)$,*
3. *α is the last node on the right spine of $Up(g_0)$ or on the left spine of $Up(0)$, q_1 is on the right spine of $Up(g_0)$ and q_1' is on the left spine of $Up(h_0)$*
4. *for all nodes u in $Up(g_1)$ and all nodes v in $Up(h_1)$ we have $u < v$,*
5. *$\textsc{UndoInjury}(p_1, q_1)$ being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1))$ and $\textsc{UndoInjury}(p_1, q_1')$ being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1'))$,*
6. *$\textsc{UndoFatality}(p_1, q_1)$ being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1'))$ and $\textsc{UndoFatality}(p_1, q_1')$ being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1))$,*
7. *$\textsc{UndoScare}(\hat{p}_1, q_1)$ being called implies $Bth(q_1) = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1'))$ and $\textsc{UndoScare}(\hat{p}_1, q_1')$ being called implies $Bth(q_1') = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1))$.*

Proof. The trees $\mathrm{Up}(g_0)$ and $\mathrm{Up}(h_0)$ are the input to the algorithm, i.e., $\mathrm{Up}(g)$ and $\mathrm{Up}(h)$, and thus satisfy Invariants 4.1.5 to 4.1.7. Since α is a hook, the in-trail between α and $\mathsf{dth}(\alpha)$ is empty. By definition $\alpha = \mathrm{Bth}(b_0') = \mathrm{Bth}(q_1')$. The inequalities $f(\alpha) > f(p_i)$ and $f(\alpha) > f(\hat{p}_1)$ also follow from the assumptions on g and h. The nodes $q_1 = b_0$ and $q_1' = b_0'$ are on the right spine of $\mathrm{Up}(g_0)$ and the left spine of $\mathrm{Up}(h_0)$, respectively. The node $\alpha = \mathrm{Bth}(q_1')$ is thus also on the

spine of $\mathrm{Up}(h_0)$ and since it is a leaf it is the last node on the spine. Claim 4 holds by the assumption that items in g are all less than items in h.

We now prove claims 5, 6 and 7 If $f(q_1) < f(q_1)'$, then $p_1 = a_0$ and by Invariant 4.1.6 $f(p_1) > f(\hat{p}_1)$. Thus UndoInjury(p_1, q_1) is executed. Since $\mathrm{Bth}(q_1) = a_0$ is an up-type item it also holds that $\mathsf{in}(q_1) = a_0$. As by definition $f(\alpha) > f(a_0)$ the inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1))$ holds. This implies claim 5. If $f(q_1) > f(q_1')$, then $p_1 = a_0$ and $p_1 \neq \mathrm{Bth}(q_1')$. Either UndoFatality$(p_1, q_1')$ or UndoScare$(\hat{p}_1, q_1)$ is executed. The inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_1))$ holds as we have just shown and this proves claims 6 and 7.

We have shown that all claims hold at the beginning of the first iteration and this proves the lemma. □

Lemma 4.2.5 (Inductive Step). *If at the beginning of the i-th iteration the following conditions hold*

1. *$Up(g_{i-1})$ and $Up(h_{i-1})$ satisfy Invariants 4.1.5 to 4.1.7,*
2. *the in-trail between α and $\mathsf{dth}(\alpha)$ is empty, $\alpha \in \{Bth(q_i), Bth(q_i')\}$, $f(\alpha) > f(\hat{p}_i)$, $f(\alpha) > f(p_i)$,*
3. *α is the last node on the right spine of $Up(g_{i-1})$ or on the left spine of $Up(h_{i-1})$, q_i is on the right spine of $Up(g_{i-1})$ and q_i' is on the left spine of $Up(h_{i-1})$*
4. *for all nodes u in $Up(g_i)$ and all nodes v in $Up(h_i)$ we have $u < v$,*
5. *UndoInjury(p_i, q_i) being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$ and UndoInjury(p_i, q_i') being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i'))$,*
6. *UndoFatality(p_i, q_i) being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i'))$ and UndoFatality(p_i, q_i') being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$,*
7. *UndoScare$(\hat{p}_i, q_i)$ being called implies $Bth(q_i) = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i'))$ and UndoScare$(\hat{p}_i, q_i')$ being called implies $Bth(q_i') = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$,*

then at the beginning of the next iteration, if it exists, it holds that

1. *$Up(g_i)$ and $Up(h_i)$ satisfy Invariants 4.1.5 to 4.1.7,*
2. *the in-trail between α and $\mathsf{dth}(\alpha)$ is empty, $\alpha \in \{Bth(q_{i+1}), Bth(q_{i+1}')\}$, $f(\alpha) > f(\hat{p}_{i+1})$, $f(\alpha) > f(p_{i+1})$,*
3. *α is the last node on the right spine of $Up(g_i)$ or on the left spine of $Up(h_i)$, q_{i+1} is on the right spine of $Up(g_i)$ and q_{i+1}' is on the left spine of $Up(h_i)$*
4. *for all nodes u in $Up(g_{i+1})$ and all nodes v in $Up(h_{i+1})$ we have $u < v$,*
5. *UndoInjury(p_{i+1}, q_{i+1}) being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}))$ and UndoInjury(p_{i+1}, q_{i+1}') being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}'))$,*
6. *UndoFatality(p_{i+1}, q_{i+1}) being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}'))$ and UndoFatality(p_{i+1}, q_{i+1}') being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}))$,*
7. *UndoScare$(\hat{p}_{i+1}, q_{i+1})$ being called implies $Bth(q_{i+1}) = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}'))$ and UndoScare$(\hat{p}_{i+1}, q_{i+1}')$ being called implies $Bth(q_{i+1}') = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}))$.*

Proof. Assume that the conditions hold. We prove the claims for $f(q_i) < f(q'_i)$ by considering separately the three cases that the algorithm distinguishes. The case $f(q_i) > f(q'_i)$ is symmetric.

Case 1 ($f(p_i) < f(\hat{p}_i)$ and $p_i = \mathrm{Bth}(q_i)$): UndoInjury(p_i, q_i) is called, which removes nodes j with $f(j) < f(q_i)$ from the mid-trail between α and $\mathsf{dth}(\alpha)$ and inserts them into the in-trail between p_i and q_i, specifically between the nodes $\mathsf{in}(q_i)$ and q_i.

We first prove that $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$ satisfy Invariants 4.1.5 to 4.1.7. By assumption 1 these invariants holds for $\mathrm{Up}(g_{i-1})$ and $\mathrm{Up}(h_{i-1})$. To see that they also hold for $\mathrm{Up}(h_i)$, observe that removing maxima along with its banana subtree does not affect the invariants. For $\mathrm{Up}(g_i)$ we first show Invariant 4.1.7. Write j^- and j^+ for the lowest and highest of the nodes are moved, respectively. That is, in $\mathrm{Up}(g_{i-1})$ $j^- = \mathsf{mid}(\alpha)$ and j^+ is the highest node on the mid-trail between α and $\mathsf{dth}(\alpha)$ such that $f(j^+) < f(q_i)$. Write j_q for $\mathsf{in}(q_i)$ in $\mathrm{Up}(g_{i-1})$. In $\mathrm{Up}(g_i)$ these nodes are inserted such that $j^- = \mathsf{up}(j_q)$ and $j^+ = \mathsf{in}(q_i)$. By the assumption that $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$ (assumption 5) we thus have $f(\mathsf{dn}(j^-)) < f(j^-)$ in $\mathrm{Up}(g_i)$. Furthermore, $f(j^+) < f(\mathsf{up}(j^+)) = f(q_i)$. By assumption 4 and Invariant 4.1.7 for $\mathrm{Up}(h_{i-1})$ it holds in $\mathrm{Up}(g_i)$ that $\mathsf{dn}(j^-) < j^- < \mathsf{up}(j^-) < \cdots < \mathsf{dn}(j^+) < j^+$ and $\mathsf{up}(j^+) = q_i < j^+$. Thus Invariant 4.1.7 holds in $\mathrm{Up}(g_i)$. To see that Invariant 4.1.5 is satisfied note that the j that are inserted into $\mathrm{Up}(g_i)$ satisfy $q_i < j$ by assumption 4. Since q_i is on the right spine $q_i < p_i = \mathrm{Bth}(q_i)$ and thus the condition of Invariant 4.1.5 holds for q_i and its ancestors. The nodes j gain descendants u in the same subtree $\mathsf{dn}(j)$, and these nodes u satisfy $u < j$, again by assumption 4. Thus, these nodes j also satisfy the condition for Invariant 4.1.5. For no other node do the subtrees change and thus Invariant 4.1.5 holds in $\mathrm{Up}(g_i)$. Finally, we prove Invariant 4.1.6 by analyzing the nodes for which the nodes at $\mathsf{dth}(\mathsf{low}(\cdot))$ changes. These are the nodes v with $\mathsf{dth}(v) = j$ for some moved node j. They originally have $\mathsf{dth}(\mathsf{low}(v)) = \alpha$ and this changes to $\mathsf{dth}(\mathsf{low}(v)) = p_i$. Assumption 2 that $f(\alpha) > f(p_i)$ thus implies $f(\mathsf{dth}(\mathsf{low}(v))) > f(\alpha) > f(p_i)$. This implies Invariant 4.1.6. In the remainder of the proof for this case we assume Invariants 4.1.5 to 4.1.7 for $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$.

The in-trail between α and $\mathsf{dth}(\alpha)$ is empty at the beginning of the i-th iteration, is not modified by UndoInjury and is thus also empty at the beginning of the next iteration. As $q'_{i+1} = q'_i$ and $\mathsf{dth}(\alpha) = q'_i$ is unchanged it holds that $\alpha = \mathrm{Bth}(q'_{i+1})$ at the beginning of the next iteration. If in the next iteration $f(q_{i+1}) < f(q'_{i+1})$, then $p_{i+1} = \mathrm{Bth}(q_{i+1}) = \mathsf{low}(q_i) = \hat{p}_i$ and $\hat{p}_{i+1} = \mathsf{low}(q_{i+1})$. By Invariant 4.1.6 this implies $f(\hat{p}_{i+1}) < f(p_{i+1}) < f(p_i) < \alpha$, where the last inequality follows from assumption 2. Otherwise, if in the next iteration $f(q_{i+1}) > f(q'_{i+1})$, then $p_{i+1} = \mathrm{Bth}(q_{i+1}) = \mathsf{low}(q_i)$ (since $\mathrm{Bth}(q'_{i+1}) = \alpha$) and $\hat{p}_{i+1} = \mathsf{low}(q'_{i+1}) = \mathsf{low}(q'_i)$. By Invariant 4.1.6 and assumption 2 this implies $f(\hat{p}_{i+1}) < f(\alpha)$ and $f(p_{i+1}) < f(p_i) < f(\alpha)$. This proves claim 2. Claim 3 follows immediately from the assumption 3: α was the last node on the left spine of $\mathrm{Up}(h_{i-1})$ at the beginning of the iteration, and the removal of nodes from its mid-trail does not change that; $q_{i+1} = \mathsf{dth}(\mathsf{low}(q_i))$, which is on the right spine of $\mathrm{Up}(g_i)$ since q_i is on the right spine of $\mathrm{Up}(g_{i-1})$ and $q'_{i+1} = q'_i$ is on the left spine of $\mathrm{Up}(h_i)$ because q'_i is on the left spine of $\mathrm{Up}(h_{i-1})$.

Since α is the last node on the left spine of $\mathrm{Up}(h_{i-1})$ and its in-trail is empty, the nodes that are moved to $\mathrm{Up}(g_i)$ are the leftmost nodes of $\mathrm{Up}(h_{i-1})$ other than α. Assumption 4 and the definition of α thus imply the claim 4.

We now prove claims 5, 6 and 7. Recall that $q'_{i+1} = q_{i+1}$ and $q'_{i+1} = \mathsf{dth}(\mathsf{low}(q_i))$. In total there are six cases, but only three can occur:

(i) $f(q_{i+1}) < f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p_{i+1} = \mathrm{Bth}(q_{i+1})$
$\Longrightarrow$ call to UndoInjury(p_{i+1}, q_{i+1})

(ii) $f(q_{i+1}) > f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p_{i+1} = \mathrm{Bth}(q_{i+1})$
$\Longrightarrow$ call to UndoFatality(p_{i+1}, q'_{i+1})

(iii) $f(q_{i+1}) > f(q'_{i+1})$, $f(\hat{p}_{i+1}) > f(p_{i+1})$
$\Longrightarrow$ call to UndoScare$(\hat{p}_{i+1}, q'_{i+1})$

Since $p_{i+1} = \mathrm{Bth}(q_{i+1}) \neq \alpha$, if $f(q_{i+1}) < f(q_{i+1})'$, then UndoFatality will not be called. By Invariant 4.1.6 it also holds that $f(\hat{p}_{i+1}) < f(p_{i+1})$, which implies that UndoScare will not be called. Finally, $\mathrm{Bth}(q'_{i+1}) = \alpha$ implies that if $f(q_{i+1}) > f(q'_{i+1})$, then UndoInjury will not be called. We analyze the three possible cases separately:

(i) $f(q_{i+1}) < f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p_{i+1} = \mathrm{Bth}(q_{i+1})$ and the function UndoInjury(p_{i+1}, q_{i+1}) is called. We have shown above that q_{i+1} is on the spine, as is q_i, and thus $q_{i+1} = \mathsf{dth}(\mathsf{low}(q_i))$ implies $\mathsf{in}(q_{i+1}) = q_i$. Since $f(\mathsf{mid}(\alpha))$ was not moved to $\mathrm{Up}(g_i)$ and $\mathsf{dth}(\alpha) = f(q'_i) > f(q_i)$ it follows that $f(\mathsf{mid}(\alpha)) > f(q_i) = f(\mathsf{in}(q_{i+1}))$.

(ii) $f(q_{i+1}) > f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p_{i+1} = \mathrm{Bth}(q_{i+1})$ and the function UndoFatality(p_{i+1}, q'_{i+1}) is called. The inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}))$ holds by the same argument as in the previous case.

(iii) $f(q_{i+1}) > f(q'_{i+1})$, $f(\hat{p}_{i+1}) > f(p_{i+1})$ and UndoScare$(\hat{p}_{i+1}, q'_{i+1})$ is called. Since $q'_{i+1} = q'_i$ and $\alpha = \mathrm{Bth}(q'_i)$ it holds that $\alpha = \mathrm{Bth}(q'_{i+1})$. The inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}))$ holds by the same argument as in the other cases.

In all three cases the claims hold.

Case 2 ($f(p_i) < f(\hat{p}_i)$ and $p_i = \mathrm{Bth}(q'_i)$): UndoFatality(p_i, q_i) is called, which takes on the in-trail between p_i and q'_i and the nodes j with $f(j) < f(q_i)$ on the mid-trail between p_i and q'_i and swaps them with $\alpha = \mathrm{Bth}(q_i)$, such that the nodes formerly on the in-trail extend the mid-trail of q_i and the nodes j extend the in-trail of q_i.

We first prove that $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$ satisfy Invariants 4.1.5 to 4.1.7. By assumption 1 these invariants hold for $\mathrm{Up}(g_{i-1})$ and $\mathrm{Up}(h_{i-1})$. To see that they also hold for $\mathrm{Up}(h_i)$, observe that replacing a part of the banana (p_i, q'_i) by α does not affect Invariants 4.1.5 and 4.1.7. By assumption 2 $f(\alpha) > f(p_i)$ and thus $f(\alpha) > f(p_i) > f(\mathsf{low}(\mathsf{dth}(\alpha))$, where the second inequality follows from Invariant 4.1.6 for $\mathrm{Up}(h_{i-1})$. As $\mathsf{low}(\mathsf{dth}(\cdot))$ changes for no other node Invariant 4.1.6 also holds for $\mathrm{Up}(h_i)$. Write j^+ for the topmost node on the mid-trail between p_i and q'_i in $\mathrm{Up}(h_{i-1})$ that is moved to $\mathrm{Up}(g_i)$, and k^+ for the topmost node on the in-trail between p_i and q'_i in $\mathrm{Up}(h_{i-1})$. As the entire in-trail is moved to the left tree $k^+ = \mathsf{in}(q'_i)$ in $\mathrm{Up}(h_{i-1})$. Write m_α for the node $\mathsf{mid}(\alpha)$ in $\mathrm{Up}(g_{i-1})$. In $\mathrm{Up}(g_i)$ $\mathsf{dn}(m_\alpha) = k^+$. By assumption 6 we have $f(m_\alpha) > f(k^+)$, and by definition of j^+ we have $f(j^+) < f(q_i)$. Furthermore, by assumption 4 it holds that $m_\alpha < k^+$ and $q_i < j^+$. Since these are the only node where $\mathsf{up}(\cdot)$ and $\mathsf{dn}(\cdot)$ pointers change it follows that Invariant 4.1.7 holds in $\mathrm{Up}(g_i)$. To see that Invariant 4.1.5 is satisfied note that the nodes u that are inserted into $\mathrm{Up}(g_i)$ satisfy $u > v$ for any node v that is already in this tree from the previous iteration. With $q_i < \mathrm{Bth}(q_i)$ as q_i is on the right spine and $s < \mathsf{dn}(s)$ for any node on the mid-trail beginning at q_i Invariant 4.1.5 follows. The nodes for which $\mathsf{low}(\mathsf{dth}(\cdot))$ changes are those nodes t with $\mathsf{dth}(t)$ on the mid-trail beginning at q_i and the node p_i. It changes from α to p_i.

By assumption 2 $f(p_i) < f(\alpha)$, so $f(\mathsf{low}(\mathsf{dth}(t))) > f(\alpha) > f(p_i)$. Furthermore, for p_i we have $\mathsf{low}(\mathsf{dth}(p_i)) = \hat{p}_i$ and they satisfy $f(p_i) > f(\hat{p})$. This implies Invariant 4.1.6 for $\mathrm{Up}(g_i)$ and concludes the proof of claim 1. In the remainder of the proof for this case we assume Invariants 4.1.5 to 4.1.7 for $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$.

After the iteration $\mathsf{dth}(\alpha) = q_i'$. As the nodes on the in-trail beginning at q_i' are removed, this trail is empty. Since $q_{i+1}' = q_i'$ it holds that $\alpha = \mathrm{Bth}(q_{i+1}')$. For the inequalities $f(\alpha) > f(p_{i+1})$ and $f(\alpha) > f(\hat{p}_{i+1})$ we refer to the proof for the previous case. It follows that claim 2 holds. The node q_i' is on the left spine of $\mathrm{Up}(h_{i-1})$ and thus $q_{i+1}' = q_i'$ is on the left spine of $\mathrm{Up}(h_i)$. Since $\alpha = \mathrm{Bth}(q_{i+1}')$ in $\mathrm{Up}(h_i)$ it also follows that α is on the left spine of $\mathrm{Up}(h_i)$. Similarly, in $\mathrm{Up}(g_{i-1})$ and $\mathrm{Up}(g_i)$ the node q_i is on the right spine and thus $q_{i+1} = \mathsf{dth}(\mathsf{low}(q_i))$ is also on the right spine. This proves claim 3. For the proof of claim 4 we again refer to the previous case, since the proof is similar.

We now prove claims 5, 6 and 7. Again, in the next iteration only three out of six cases can occur:

(i) $f(q_{i+1}) < f(q_{i+1}')$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p = \mathrm{Bth}(q_{i+1})$
$\Longrightarrow$ call to UndoInjury(p_{i+1}, q_{i+1})

(ii) $f(q_{i+1}) > f(q_{i+1}')$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p = \mathrm{Bth}(q_{i+1})$
$\Longrightarrow$ call to UndoFatality(p_{i+1}, q_{i+1}')

(iii) $f(q_{i+1}) > f(q_{i+1}')$, $f(\hat{p}_{i+1}) > f(p_{i+1})$
$\Longrightarrow$ call to UndoScare$(\hat{p}_{i+1}, q_{i+1}')$

Note that these are the same cases as in Case 1. The proof that the remaining three cases cannot occur is identical. The remainder of the proof is also similar to the proof in case 1:

(i) $f(q_{i+1}) < f(q_{i+1}')$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p = \mathrm{Bth}(q_{i+1})$ and the function UndoInjury(p_{i+1}, q_{i+1}) is called. Again, q_{i+1} and q_i are on the spine and thus $\mathsf{in}(q_{i+1}) = q_i$. $f(\mathsf{mid}(\alpha))$ is one of the nodes that have not been moved from the mid-trail, and thus $f(\mathsf{mid}(\alpha)) > f(q_i) = f(\mathsf{in}(q_{i+1}))$.

(ii) $f(q_{i+1}) > f(q_{i+1}')$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p = \mathrm{Bth}(q_{i+1})$ and the function UndoFatality(p_{i+1}, q_{i+1}') is called. The inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1}))$ holds by the same argument as in the previous case.

(iii) $f(q_{i+1}) > f(q_{i+1}')$, $f(\hat{p}_{i+1}) > f(p_{i+1})$ and UndoScare$(\hat{p}_{i+1}, q_{i+1}')$ is called. As before $\alpha = \mathrm{Bth}(q_i') = \mathrm{Bth}(q_{i+1}')$. The inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_{i+1})$ holds as shown in the other cases.

This proves claims 5, 6 and 7.

Case 3 ($f(\hat{p}_i) < f(p_i)$): UndoScare$(\hat{p}_i, q_i)$ is called, which sets $f(\alpha) = f(\hat{p}_i) - \varepsilon$ and then performs an interchange of minima. Note that $\alpha = \mathrm{Bth}(q_i)$, as $f(p_i) < f(\hat{p}_i)$, which by Invariant 4.1.6 for $\mathrm{Up}(g_i)$ cannot be if $p_i = \mathrm{Bth}(q_i)$. Thus, $\alpha \neq \mathrm{Bth}(q_i')$ and by assumption 2 this implies $\alpha = \mathrm{Bth}(q_i)$. Claim 4 follows directly from assumption 4, as the contents of the trees do not change. Invariants 4.1.5 to 4.1.7 immediately follow for $\mathrm{Up}(g_i)$ from the correctness of the interchange of minima. As $\mathrm{Up}(h_i) = \mathrm{Up}(h_{i-1})$ these invariants hold $\mathrm{Up}(h_i)$ by assumption 1. In the remainder of the proof for this case we assume Invariants 4.1.5 to 4.1.7 for $\mathrm{Up}(g_i)$ and $\mathrm{Up}(h_i)$.

The in-trail between α and $\mathsf{dth}(\alpha)$ is empty in $\mathrm{Up}(g_{i-1})$ by assumption 2. As $\mathsf{dth}(\alpha) = q_i$ is on a spine the in-trail remains empty after the interchange of minima. The interchange also results in $\mathsf{dth}(\alpha) = q_{i+1}$ in $\mathrm{Up}(g_i)$. If

$f(q_{i+1}) < f(q_{i+1})'$, then $p_{i+1} = \mathrm{Bth}(q'_{i+1}) = \mathrm{Bth}(q'_i) = p_i$. As $f(p_i) < f(\hat{p}_i)$ and ε is defined to be sufficiently small it follows that $f(p_{i+1}) < f(\alpha)$. Furthermore, $f(\hat{p}_{i+1}) < f(\alpha)$ by Invariant 4.1.6. Otherwise, if $f(q_{i+1}) > f(q_{i+1})'$, then $p_{i+1} = \mathrm{Bth}(q'_{i+1}) = \mathrm{Bth}(q'_i) = p_i$, which we have just shown to satisfy $f(p_{i+1}) < f(\alpha)$. By Invariant 4.1.6 $f(\hat{p}_{i+1}) < f(p_{i+1})$, which implies $f(\hat{p}_{i+1}) < f(\alpha)$. This proves claim 2.

The node q_{i+1} is on the spine since it is already on the spine in $\mathrm{Up}(g_{i-1})$ and the interchange of minima does not change its ancestors. The node α remains on the spine since its in-trail remains empty and $q_{i+1} = \mathsf{dth}(\alpha)$ is on the left spine in $\mathrm{Up}(g_{i+1})$. Since it is also a leaf it is the last node on the spine. Finally, q'_{i+1} is on the spine since $q'_{i+1} = q'_i$. This proves 3.

It remains to prove claims 5, 6 and 7. Again, in the next iteration only three out of six cases can occur:

(i) $f(q_{i+1}) < f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p_{i+1} = \mathrm{Bth}(q'_{i+1})$
$\Longrightarrow$ call to UndoFatality(p_{i+1}, q_{i+1})

(ii) $f(q_{i+1}) < f(q'_{i+1})$, $f(\hat{p}_{i+1}) > f(p_{i+1})$
$\Longrightarrow$ call to UndoScare$(\hat{p}_{i+1}, q_{i+1})$

(iii) $f(q_{i+1}) > f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i_1})$ and $p_{i+1} = \mathrm{Bth}(q'_{i+1})$
$\Longrightarrow$ call to UndoInjury(p_{i+1}, q'_{i+1})

Since $\mathrm{Bth}(q_{i+1}) = \alpha$ we cannot have $p_{i+1} = \mathrm{Bth}(q_{i+1})$ and thus if $f(q_{i+1}) < f(q'_{i+1})$ then UndoInjury is not called. If $f(q_{i+1}) > f(q'_{i+1})$ then $f(p_{i+1}) > f(\hat{p}_{i+1})$ by Invariant 4.1.6, and so UndoScare is not called. UndoFatality is not called in this case as $p_{i+1} = \mathrm{Bth}(q'_{i+1})$. We now analyze the three possible cases:

(i) $f(q_{i+1}) < f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i+1})$ and $p_{i+1} = \mathrm{Bth}(q'_{i+1})$ and the function UndoFatality(p_{i+1}, q_{i+1}) is called. The node $\mathsf{mid}(\alpha)$ is the same in $\mathrm{Up}(g_{i-1})$ and $\mathrm{Up}(g_i)$. Furthermore $\mathsf{in}(q_{i+1})' = \mathsf{in}(q'_i)$, as $\mathrm{Up}(h_i) = \mathrm{Up}(h_{i-1})$. Thus, the assumption that $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q'_i)$ implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q'_{i+1}))$.

(ii) $f(q_{i+1}) < f(q'_{i+1})$, $f(\hat{p}_{i+1}) > f(p_{i+1})$ and UndoScare$(\hat{p}_{i+1}, q_{i+1})$ is called. We have shown above that $\mathsf{dth}(\alpha) = q_{i+1}$ which implies $\alpha = \mathrm{Bth}(q_{i+1})$.

(iii) $f(q_{i+1}) > f(q'_{i+1})$, $f(p_{i+1}) > f(\hat{p}_{i_1})$ and $p_{i+1} = \mathrm{Bth}(q'_{i+1})$ and the function UndoInjury(p_{i+1}, q'_{i+1}) is called. The inequality $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q'_{i+1}))$ holds by the same argument as in the first case.

This concludes the proof of the inductive step. □

Theorem 4.2.6 (Correctness of Glue)**.** *Given two lists of items and corresponding maps g and h Glue applied to $Up(g)$ and $Up(h)$ yields the up-tree $Up(f) = Up(g \cdot h)$.*

Proof. We show by induction over the iterations of Glue that at the beginning of the i-th iteration it holds that

1. $\mathrm{Up}(g_{i-1})$ and $\mathrm{Up}(h_{i-1})$ satisfy Invariants 4.1.5 to 4.1.7,
2. the in-trail between α and $\mathsf{dth}(\alpha)$ is empty, $\alpha \in \{\mathrm{Bth}(q_i), \mathrm{Bth}(q'_i)\}$, $f(\alpha) > f(\hat{p}_i)$, $f(\alpha) > f(p_i)$,

3. α is the last node on the right spine of $\mathrm{Up}(g_{i-1})$ or on the left spine of $\mathrm{Up}(h_{i-1})$, q_i is on the right spine of $\mathrm{Up}(g_{i-1})$ and q_i' is on the left spine of $\mathrm{Up}(h_{i-1})$
4. for all nodes u in $\mathrm{Up}(g_i)$ and all nodes v in $\mathrm{Up}(h_i)$ we have $u < v$,
5. UndoInjury(p_i, q_i) being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$ and UndoInjury(p_i, q_i') being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i'))$,
6. UndoFatality(p_i, q_i) being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i'))$ and UndoFatality(p_i, q_i') being called implies $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$,
7. UndoScare$(\hat{p}_i, q_i)$ being called implies $\mathrm{Bth}(q_i) = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i'))$ and UndoScare$(\hat{p}_i, q_i')$ being called implies $\mathrm{Bth}(q_i') = \alpha$, $f(\mathsf{mid}(\alpha)) > f(\mathsf{in}(q_i))$.

Lemma 4.2.4 proves the base case for the first iteration $i = 1$, and the induction step is in Lemma 4.2.5. We assume these claims for all iterations.

We now show that there exists an iteration after which one tree is empty. Assume that the global maximum of h has greater value than the global maximum of g and both trees have nodes other than the special root and α. The case where the global maximum of g has greater value than that of h is symmetric. Consider any iteration I in which $f(q_I') > f(u)$ for any $u \neq \beta_g$ in $\mathrm{Up}(g_{I-1})$ and $q_I = \beta_g$. Since in some iteration $q_I' = \beta_h$ this iteration exsists. In this iteration $f(q_I') < f(q_I)$, either since q_I is a special root and q_I' is not, or since $f(\beta_h) < f(\beta_g)$ by definition. We distinguish two cases: $q_I' \neq \beta_h$ and $q_I = \beta_h$.

$q_I' \neq \beta_h$: There are three cases to consider: (1) $\mathrm{Bth}(q_I') \neq \alpha$ and $p_I = \mathrm{Bth}(q_I')$; (2) $\mathrm{Bth}(q_I') = \alpha$ and $f(p_I) > f(\hat{p}_I)$; (3) $\mathrm{Bth}(q_I') = \alpha$ and $f(\hat{p}_I) > f(p_I)$. In case (1) $f(p_I) > f(\hat{p}_I)$ by Invariant 4.1.6 and thus UndoInjury(p_I, q_I') is called. In case (2) UndoFatality(p_I, q_I') is called. In case (3) UndoScare$(\hat{p}_I, q_I')$ is called. If case (1) occurs, then $\alpha = \mathrm{Bth}(q_I) = \mathrm{Bth}(\beta_g)$, the in-trail between α and β_g is empty, and all nodes from the mid-trail between α and β_g are moved to $\mathrm{Up}(h_I)$. This leaves $\mathsf{in}(\beta_g) = \mathsf{mid}(\beta_g) = \alpha$ and thus the loop terminates. If case (2) occurs, then $p_I = \mathrm{Bth}(q_I) = \mathrm{Bth}(\beta_g)$ and all nodes on trails between p_I and β_g except β_g are moved to $\mathrm{Up}(h_I)$, and α is moved to $\mathrm{Up}(g_I)$ such that $\mathsf{in}(\beta_g) = \mathsf{mid}(\beta_g) = \alpha$. Thus, the loop terminates. In case (3) $f(\alpha)$ is set to $f(\hat{p}_I) - \varepsilon$ and an interchange of minima between α and $\hat{p}_I$ is performed. Thus, both trees are not empty. However, $q_{I+1}' = \mathsf{up}(q_I)$ is the next node upwards on the spine towards β_h and the next iteration is another iteration where $f(q_I') > f(u)$ for all $u \neq \beta_g \in \mathrm{Up}(g_i)$.

$q_I' = \beta_h$: This case is eventually reached, since the q_i' move upwards on the spine to β_h. We have defined $f(\beta_h) < f(\beta_g)$. There are again three cases to consider: (1) $\mathrm{Bth}(q_I') \neq \alpha$ and $p_I = \mathrm{Bth}(q_I')$; (2) $\mathrm{Bth}(q_I') = \alpha$ and $f(p_I) > f(\hat{p}_I)$; (3) $\mathrm{Bth}(q_I') = \alpha$ and $f(\hat{p}_I) > f(p_I)$. These are the same cases as above and in case (1) and (2) the loop terminates with $\mathsf{in}(\beta_g) = \mathsf{mid}(\beta_g) = \alpha$, as above. The third case, where $f(\hat{p}_I) > f(p_I)$, cannot occur since we defined $f(\hat{p}_I) = f(\mathsf{low}(\beta_h)) = f(\mathtt{nil}) = -\infty$. Thus, when $q_I' = \beta_h$, the algorithm terminates.

The choice that $f(\beta_h) < f(\beta_g)$ is arbitrary and we could have also chosen $f(\beta_g) < f(\beta_h)$. In this case we would get three symmetric cases for $q_I' = \beta_h$ with the same outcome.

Once the loop terminates after iteration I^* with $\mathsf{in}(\beta_g) = \mathsf{mid}(\beta_g) = \alpha$, the tree $\mathrm{Up}(h_{I^*})$ contains all items from $\mathrm{Up}(g)$ and $\mathrm{Up}(h)$ except for the dummy α. Thus, $\mathrm{Up}(h_{I^*}) = \mathrm{Up}(g \cdot h)$, which concludes the proof. □

5 BANANA TREES FOR THE PERSISTENCE IN TIME SERIES EXPERIMENTALLY

5.1 Introduction

Time series are pervasive across numerous disciplines, ranging from economics and finance to environmental science and healthcare. Often, time series data is dynamic, not static; for example, wearable devices continuously monitor health metrics such as heart rate or blood sugar levels, requiring robust techniques for instant analysis and decision-making. Effective synthesis of the underlying trends in such dynamic data is crucial for predictive modeling and strategic interventions.

Within the field of topological data analysis, persistent homology [5, 87] is recognized as a powerful framework for capturing multi-scale features in complex data sets. Researchers have long considered the challenge of dynamic persistence, and the vineyard algorithm [28] was the first developed to address updates in persistence diagrams (defined in Section 5.2.1) as the input data evolves; see [90] for an implementation. This algorithm, while capable of handling data more general than just time series, requires time linear in the size of the input complex per update, which is restricted to swapping the order of two values. This prompts us to question whether faster processing could be achieved for one-dimensional data. To address this challenge, we implement the banana tree data structure [93] (see also Chapter 3), recently introduced and tailored specifically for the persistent homology of dynamic time series. Their theoretical analysis promises the processing of each update in time logarithmic in the number of items plus linear in the number of changes in the persistent diagram; compare this with the linear time require to recompute the diagram [26]. This paper investigates what these results mean in practice. We highlight some of the specific contributions:

- *Efficiency at scale:* we demonstrate that for large data sets, such as those containing over 10^6 items, performing a local or topological update on an unbiased random walk using banana trees is at least 100 times faster than recomputing persistence with `Gudhi` [31], the state of the art static algorithm (see Figure 5.3.3).
- *Structure and performance:* we explore how the structure of banana trees, characterized by parameters such as the number of critical items, the nesting depth of bananas, and the lengths of trails, directly impacts the running time of our algorithms. This analysis identifies the types of data best suited for efficient processing (see Figure 5.3.1).
- *Worst-case scenarios:* we construct specific examples that challenge our algorithms, and show empirically that these scenarios are rare and highly unstable, reinforcing the robustness of our approach (see Table 5.4.1).
- *Real-world data:* through analysis of three specific data sets, we illustrate that real-world data can exhibit structural properties similar to those

of unbiased random walks. Preliminary evidence suggests potential for the broad applicability of banana trees in practical scenarios (see Table 5.4.2).

In software for topological data analysis, tools like the `Gudhi` library [31], Dionysus [94, 95], and Ripser [96] are established for static data sets but require complete recomputation for updates, making them less suited for dynamic contexts. We aim for our implementation to set a new standard in the topological processing of dynamic time series.

OUTLINE. Section 5.2 introduces persistent homology tailored to time series data and gives an overview of the banana trees data structure. Section 5.3 evaluates the performance of banana trees through experiments involving time series generated from random walks, with and without a bias. Section 5.4 explores three distinct types of input time series to assess the performance of banana trees: worst-case scenarios, quasi-periodic signals, and real-world data. Section 5.5 concludes the paper with a summary of our findings and their implications.

5.2 Banana Trees for Time Series

We start by introducing persistent homology, a method to discern features of data across multiple scales [24]. Forfeiting the generality of this theory, we focus on time series data. We also explain what banana trees are and how they relate to persistent homology; see Chapter 3 for details on this data structure and its algorithms.

5.2.1 Persistent Homology of Time Series

By a *time series* we mean a linear list of real numbers, $c_0, c_2, \ldots, c_{n-1}$, which we view as a piecewise linear map, $f : [0, n-1] \to \mathbb{R}$, with $f(i) = c_i$ for $0 \leq i \leq n-1$. We refer to i as an *item* and c_i its *value*. A *critical item* is a local minimum or a local maximum of this map, and all other items are *non-critical*, e.g. item i if $f(i-1) < f(i) < f(i+1)$. To simplify the discussion, we assume that the map is *generic*, by which we mean that its items have distinct values. In this case, the endpoints (items 0 and $n-1$) are necessarily critical.

The *sublevel set* of f at $t \in \mathbb{R}$, denoted $f_t = f^{-1}(-\infty, t]$, are the points $x \in [0, n-1]$ that satisfy $f(x) \leq t$. When t passes the value of a minimum from below, then the number of connected components of f_t increases by 1, and if t passes the value of a maximum from below, the number of connected components decreases by 1, unless the maximum is a endpoint, in which case the number does not change. Symmetrically, we call $f^t = f^{-1}[t, \infty)$ the *superlevel set* of f at t. Observe that f^t is the sublevel set of $-f$ at $-t$, and that the minima and maxima of $-f$ are the maxima and minima of f. Persistent homology tracks the evolution of the connected components while the sublevel set of f grows, and formally defines when a component is born and when it dies. Complementing this with the same information for the superlevel sets of f, we get what is formally referred to as *extended persistent homology*; see [22] for details. It is best constructed in two phases:

- In *Phase One*, we track the connected components of the sublevel set, f_t, as t increases from $-\infty$ to ∞. A component is *born* at the smallest value of t at which a point of the component belongs to f_t, which is necessarily a minimum. The component *dies* when it merges with another component that was born earlier, which is necessarily at a maximum in the interior of $[0, n-1]$. The *ordinary subdiagram* of f records the birth and death of every component with a point in the plane whose abscissa and ordinate are those values of t at which the component is born and dies, respectively; see Figure 5.2.1.

- In *Phase Two*, we track the connected components of the superlevel set, f^t, as t decreases from ∞ to $-\infty$. Birth and death are defined accordingly, and the components are recorded in the *relative subdiagram*.

By construction, the points in the ordinary subdiagram lie above and those of the relative subdiagram lie below the diagonal. The component born at the global minimum of f is special because it does not die during Phase One. Instead, it dies at the global minimum of $-f$, which is the global maximum of f. In topological terms, this happens because the one connected component still alive at the beginning of Phase Two dies in relative homology when its first point enters the superlevel set. This class is represented by the sole point in the *essential subdiagram*. The *extended persistence diagram* is the disjoint union of the three subdiagrams. Hence, the diagram is a multi-set of points in $\mathbb{R}^2$, and so are the three subdiagrams, unless the map is generic, in which case the diagram is a set. See the left panel in Figure 5.2.1 for an example but note that it only shows a small number of points in the ordinary subdiagram. An important property of persistence diagrams is their stability with respect to small perturbation of the input data, which was first proved in [23].

The points in the persistence diagram can be characterized using the concept of a *window* introduced in [25]: start with a rectangular frame spanned by a minimum and a maximum in the graph of f such that the minimum lies on the lower edge, and extend the frame horizontally as long as the graph does not intersect the upper edge in its interior. Call the initial frame the *mid-panel*, its extension the *in-panel*, and the symmetric extension for $-f$ the *out-panel*. As proved in [25], the min-max pair defines a point in the persistence diagram iff the graph of f reaches the lower edge in the out-panel. In this case, we call the twice extended frame a *triple-panel window*. The corresponding *double-panel window* consists of the mid-panel and the in-panel but not the out-panel. Any two double-panel windows of f are either disjoint or nested, a property not shared by the triple-panel windows. We use this property to *augment* the persistence diagram by drawing an arrow from a point to another, if the double-panel windows of the first point is nested inside the double-panel window of the other point, without any other window being nested between them. See Figure 5.2.1, which shows four double-panel windows and the corresponding points and arrows in the augmented persistence diagram. The displayed frames are indeed windows because each has an out-panel—next to the mid-panel and thus opposite to the in-panel—which extends as far as the horizontal projection of the minimum (a maximum of $-f$) to the graph of the function.

5.2.2 Banana Trees

We sketch the banana trees while referring to Chapter 3 for the details. Suffice to say that they are based on the Cartesian tree introduced by Vuillemin [30], which stores a list of items in order such that each path from a leaf to the root is ordered by value. This tree has a unique decomposition into paths that correspond to double-panel windows, and the banana tree splits each path into two trails representing the windows nested inside the in-panel and the mid-panel of the corresponding window.

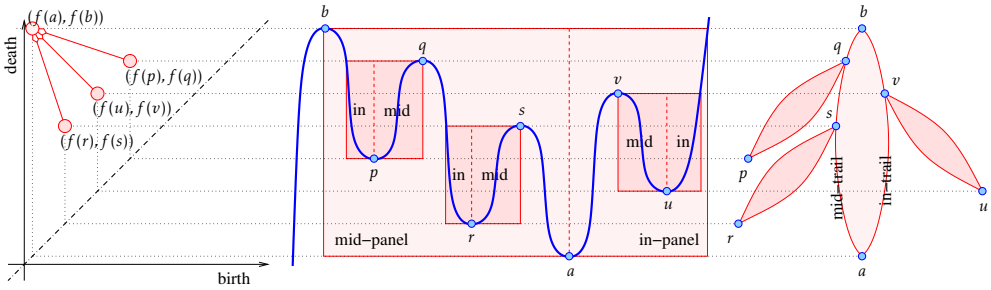

Figure 5.2.1: *Middle:* a piece of the graph of f with four double-panel windows of which three are nested inside the fourth. *Left:* the corresponding points in the persistence diagram and the arrows that reflect the nesting relation among the windows. *Right:* the four corresponding bananas in the up-tree of f.

A time series represented by a piecewise linear function, f, is stored in two banana trees together with a linked list and two standard dictionaries. We call the first banana tree the *up-tree* of f because it corresponds to Phase One of the persistence diagram construction. The second banana tree is the *down-tree* of f, which is the up-tree of $-f$. Because of this symmetry, it suffices to describe the up-tree of f. Its *leaves* are the minima of f, and its *internal nodes* are the maxima of f, with the exception of endpoint maxima, if they exist, since they are not critical in Phase One. Each point in the ordinary persistence diagram is a min-max pair, (a, b), and it is represented by a *banana* that connects the leaf a to the internal node b with two parallel trails. The *mid-trail* is a doubly-linked list connecting a to b with the maxima that span windows nested in the mid-panel of the window of a, b, and the *in-trail* is symmetrically defined; see the right panel in Figure 5.2.1. The tree structure arises because the maxima belong to the similarly defined bananas of the nested windows. Each trail is sorted in the order of value but also in the order of location along the time series. For example, the mid-trail of the big banana in Figure 5.2.1 stores a, s, q, b, which montonically increases in value and monotonically decreases in location. Compare this with the in-trail storing a, v, b, which monotonically increases in value and in location after we discard b. Indeed, we think of b as part of the min-trail while only connecting the in-trail to the rest of the tree without properly belonging to it. The global minimum of f (not shown in Figure 5.2.1) is special because it is not paired in Phase One. Indeed, it is the extra leaf whose path upward does not end at an internal node. We therefore add a *special root* as the parent of the root, connected to the global minimum by the *root banana*. While there is only one root banana, there are possibly many *leaf bananas*, which are the bananas with two empty trails (beside the minimum and maximum they connect). Indeed, we call the number of nodes between the minimum and maximum the *length* of the trail, which for trails of

leaf bananas is necessarily zero. Another important concept is the *nesting depth* of a banana, which is the number of windows that contain the corresponding window. The root banana has nesting depth zero, and all other bananas have positive nesting depth.

Each update to a time series stored in a banana tree reduces to a sequence of elementary operations, which we name by their impact on the function, f. An *interchange* happens when two maxima or two minima swap the order of their values. Unless their locations are near each other, there is a good chance that an interchange does not have any effect on the banana tree. Otherwise, it is akin a rotation in a binary search tree, if the interchange is between two maxima, and akin a local rearrangement of the path decomposition, if the interchange is between two minima. A *cancellation* removes a min-max pair or, equivalently, the corresponding leaf banana, an *anti-cancellation* is the inverse of a cancellation, and a *slide* swaps the criticality type of a critical item with that of a neighboring non-critical item. The latter three operations appear only a constant number of times in the sequence of operations whenever we adjust a value. Nevertheless, anti-cancellations present challenges to fast implementation as we have to find out where the new banana is to be inserted, and this search is not supported by any special purpose data structure. On the other hand, multiple interchanges may happen in sequence, so it is important to charge the time for processing each to a change in the persistence diagram. We refer to Chapter 3 for details.

5.3 Experimental Results for Random Walks

This section studies the structural properties of banana trees for random walks, both biased and unbiased. In addition, it compares the running times of local and topological operations with the static algorithm in `Gudhi` [31]. We do not compare it with vineyards [28], which lack optimized software for the one-dimensional case.

We implemented the banana trees data structure in C++20, using the binary search trees provided by `boost.intrusive` [97]. When performing topological operations we use splay trees, otherwise we use AVL trees. The experiments were run on a machine with two AMD EPYC 9534 64 core CPUs and 1.5 terabyte of main memory.

5.3.1 The Experimental Set-up

We write $\gamma \sim \mathcal{N}(\mu, \sigma)$ for a normally distributed random variable with mean μ and standard deviation σ. Given μ, σ, a *random walk* of length n is a real-valued function $r = r_{\mu,\sigma}$ defined by

$$r(0) = 0 \text{ and } r(i) = r(i-1) + \gamma_i \text{ for } 1 \leq i < n, \tag{10}$$

in which the $\gamma_i \sim \mathcal{N}(\mu, \sigma)$ are independent. The random walk is *unbiased* if $\mu = 0$ and *biased* if $\mu \neq 0$. We assume $\sigma = 1$ unless stated otherwise.

To evaluate the performance of local updates, we change the value of a single item, and since it depends on the amount, we fix parameters $\Delta \in \mathbb{R}$ and $k \in \mathbb{N}$ and change the value of the i-th item from $r(i)$ to $r(i) + \delta_j$, with $\delta_j = \frac{j}{k}\Delta$ for $-k \leq j \leq k$. For the experiments, we pick $\Delta = 5.0$ or 50.0 and $k = 10$, while

doing the update on a random walk of length between 10^2 and 10^6 generated for $\mu \in [-1.1]$ and $\sigma = 1$. Each experiment is repeated one hundred times and averages as well as deviations from the average are reported. For each experiment, we also measure the time to run `Gudhi` on the walk after the value change. This provides the appropriate reference time, since `Gudhi` needs to recompute the persistence diagram from scratch after each update.

To evaluate the performance of topological updates (split or glue), we pick a fraction, $c \in (0,1)$, generate a random walk of length n, and then cut it into a left walk of length $\lfloor cn \rfloor$ and a right walk of length $\lceil (1-c)n \rceil$. To *split*, we first construct the banana tree of the entire random walk, which we then cut into two banana trees, one for the left and the other for the right walk. To *glue*, we first construct the banana trees of the left and right walks, which we then combine to a single banana tree for the walk of length n. Doing this for fractions $c \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, we compare the time to split with the time used by `Gudhi` to construct the persistence diagrams of the left and right walks, and the time to glue with the time used by `Gudhi` to construct the persistence diagram of the entire random walk.

5.3.2 Structural Properties

We focus on three structural properties of banana trees, which have direct implications for the running time of our algorithms: the *number of critical items*, the *nesting depth of bananas*, and the *lengths of trails*. Since a banana tree does not store non-critical items, its number of nodes is the number of critical items. In an unbiased random walk, the expected number of critical items is 50% of all items. This fraction decreases when the bias increases, with about 27% for $\mu = \pm 1$; see the left panel in Figure 5.3.1 for more detailed information.

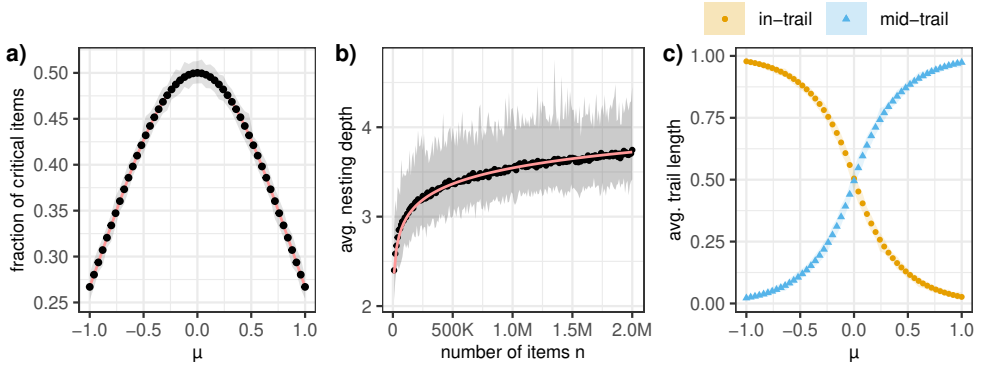

Figure 5.3.1: *Left:* the average fraction of items that are critical as a function of the bias. *Middle:* the average nesting depth of leaf bananas in banana trees of unbiased random walks with n items. The *ribbon* extends from the minimum to the maximum observed value for each n; the dots mark the mean. The *red line* is the graph of a constant times $\log n$ obtained by linear regression. *Right:* the average length of in-trails (*orange dots*) and mid-trails (*blue triangles*) in banana trees of random walks with bias μ, averaged over all input sizes.

The middle panel of Figure 5.3.1 shows the average nesting depth of a leaf banana for an unbiased random walk, which appears to scale like the logarithm of the number of items. In the unbiased case, even the maximum nesting depth seems to be small (about 13 on average for a random walk of length

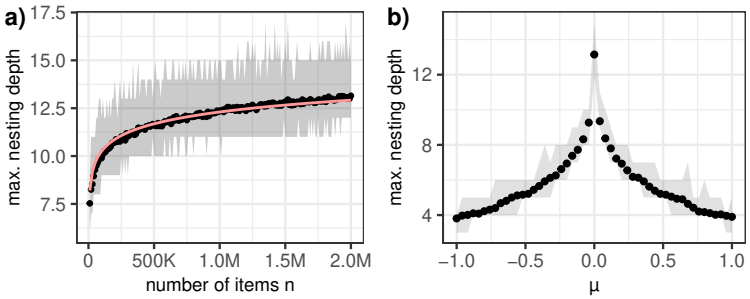

Figure 5.3.2: *Left:* the maximum nesting depth of the leaf bananas in the up-tree of an *unbiased* random walk ($\mu = 0$) of size n. The ribbon extends from the minimum to the maximum observed value for each n; the dots mark the mean. The red line is the graph of constant times $\log n$ obtained by linear regression. *Right:* Maximum nesting depth of 100 up-trees for *biased* random walks of size $2 \cdot 10^6$. The ribbon extends from the minimum to the maximum for each value of the bias parameter μ; the dots mark the mean.

$2 \cdot 10^6$), and scale logarithmically in the number of items; see Figure 5.3.2. With increasing bias, the maximum nesting depth decreases, to about 4 at $\mu = \pm 1$. This is because in the biased case, there are fewer critical items but also the bananas tend to arrange in parallel rather than inside each other. We observe that the nesting depth is about the same on average in up-trees and down-trees, both for biased and unbiased random walks, so our results hold for both trees.

Regarding trails, we observe that the length of the in-trails in the up-tree and the mid-trails in the down-tree are equal on average. By symmetry, the same holds for the mid-trails in the up-tree and the in-trails in the down-tree. Let us therefore focus on the up-tree. The right panel in Figure 5.3.1 shows how the average length of in- and mid-trails depends on the bias. In the unbiased case, both types of trails have the same length on average. For positive μ, the mid-trails tend to be longer than the in-trails, while the reverse happens for negative μ. The main cause for this trend is the longest trail, which starts to dominate the others in length as the bias increases. Assuming $\mu > 0$, the global minimum and maximum tend to lie to the left and right of the middle, respectively, with the majority of the items between them. By convention, the connecting in-trail and mid-trail contain the items to the left and right of the global minimum, respectively. This explains why the mid-trail dominates in this case, and why the in-trail dominates when $\mu < 0$. Indeed, for $\mu = \pm 1$, we observe about 95% of the internal nodes (the local maxima) on the longest trail.

5.3.3 Local Maintenance

We evaluate the performance of banana trees when the value of a single item changes by δ. Note that the corresponding update is a combination of some number of interchanges and at most one cancellation, at most one anti-cancellation, and at most two slides [93], see also Section 3.5.2. Not surprisingly, the time increases with the amount of change. For example, there is no effect at all on the persistence diagram for about 70% of the updates with $\delta = \pm 0.26$, for about 36% of the updates with $\delta = \pm 0.79$, but only about 1%

for updates with $\delta = \pm 3.4$. Taking this into account, it is not surprising that banana trees are faster than `Gudhi` when $\delta = \pm 0.26$ for all lengths of random walks we tried.

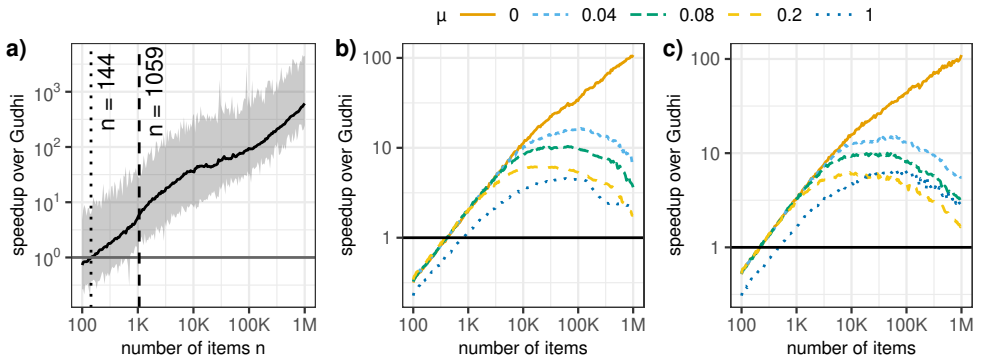

Figure 5.3.3: Comparing the maintenance of banana trees with reconstructing the persistence diagram with `Gudhi`, in which the baseline of no speedup ($10^0 = 1$) is marked with a *horizontal gray* line. *Left:* the speedup for updating a value with $\delta = \pm 5.0$ depending on the length of the random walk, n. The ribbon spans the minimum and maximum observed speedup, with the *black curve* tracing the median speedup. *Middle:* the speedup for using banana trees to cut a random walk with bias μ in half. The type and color of the curve encodes the amount of bias, and each curve shows the median speedup over a hundred repeats for each n. *Right:* the speedup for using banana trees to concatenate two equally long random walks with bias μ.

The left panel in Figure 5.3.3 focuses on the case $\delta = \pm 5.0$. As indicated by the vertical dashed and dotted lines, banana trees are faster than `Gudhi` for all random walks of length $n \geq 1059$, and faster in the median for all $n \geq 144$. Not shown in the panel is that the speedup decreases with increasing bias. In particular, for $n = 10^6$, the speedup of 612 at $\mu = 0$ decreases to 258 at $\mu = 1$, and for $n = 1059$, the speedup of 6.0 at $\mu = 0$ decreases to 2.9 at $\mu = 1$. This can be explained by recalling that for large bias the majority of the internal nodes belong to the longest trail, which connects the global minimum to the global maximum. Hence, items with similar values are likely to be near each other and thus require interchanges when an update is performed. However, even for a large change, such as for $\delta = \pm 50.0$, maintaining the banana tree is still faster than reconstructing the persistence diagram with `Gudhi`. For example, the median speedup for $n = 10^4$ exceeds 10 and for $n = 10^6$ it exceeds 100.

5.3.4 Topological Maintenance

We call the operations of *cutting* a list into two, and *concatenating* two lists to one topological because they change the number of lists in the overall organization of the data. We begin with evaluating the performance of cutting a random walk. The middle panel in Figure 5.3.3 shows the speedup over `Gudhi` when we cut a biased or unbiased random walk in half. In the unbiased case, the speedup increases with the length of the walk. Recalling the discussion of structural properties, this can be rationalized by noticing that the maximum nesting depth bounds the number of bananas that need to be split, and the trail length bounds the time to reorganize bananas. The maximum nesting depth scales like $\log n$ and the average trail length is only 0.5, so we anticipate logarithmic time for splitting, which we indeed observe in our experiments.

Altering the position of the cut increases the speedup, namely by about 2% for $c = 0.5 \pm 0.2$ and by about 7% for $c = 0.5 \pm 0.4$.

The advantage of the banana tree over `Gudhi` diminishes when the random walks get progressively more biased. The reason is again that the larger the bias, the larger the fraction of internals nodes in the longest trail of the banana tree. Splitting the corresponding banana requires the resetting of many pointers stored in nodes along this trail, which in some cases costs time proportional to the number of critical items, which we indeed observe in our experiments. Unfortunately, this more than compensates for the low nesting depth, which guarantees that only a small number of bananas need to be split. The upper half of Table 5.3.1 shows the running time and the speedup over `Gudhi` for cutting a random walk in half.

Table 5.3.1: *Upper half*: the average time for cutting a random walk of length $n = 10^6$ in half, and the speedup over `Gudhi`. *Lower half*; the average time for concatenating two random walks of length $n = 10^6/2$ each, and the speedup over `Gudhi`. All times are measured in micro-seconds.

		$\mu = 0$	$\mu = 0.04$	$\mu = 0.08$	$\mu = 0.2$	$\mu = 1$
Cut	time	93	1417	2807	5350	1683
	speedup	105.00	7.24	3.54	1.68	2.15
Concatenate	time	90	1809	3041	5138	1228
	speedup	105.00	5.31	3.12	1.74	3.22

We finally address the reverse operation, that of concatenating two random walks or, correspondingly, of gluing two banana trees. The curves in the right panel of Figure 5.3.3 are similar to those in the middle panel, which suggests that the performance is similar to that of cutting, which is confirmed by the numbers in Table 5.3.1. For $n \leq 10^4$, gluing appears to be slightly fast than splitting, but the difference is less clear already for $n = 10^6$.

5.4 Special Time Series

This section considers three types of time series: *worst-case examples* to expose when banana trees underperform, *quasi-periodic data* to illustrate how banana trees reflect periodicity, and *real-world data* to demonstrate that banana trees are not just theory. While banana trees struggle for data of the first type, they mirror the performance observed in unbiased random walks for the latter two types.

5.4.1 Worst-case Examples

The size of the augmented persistence diagram (number of points and arrows) is proportional to the number of critical items. There are configurations in which a single update changes almost the entire diagram and thus takes time at least linear in the number of such items. We construct such *worst-case inputs* and study the performance of banana trees when confronted with such data.

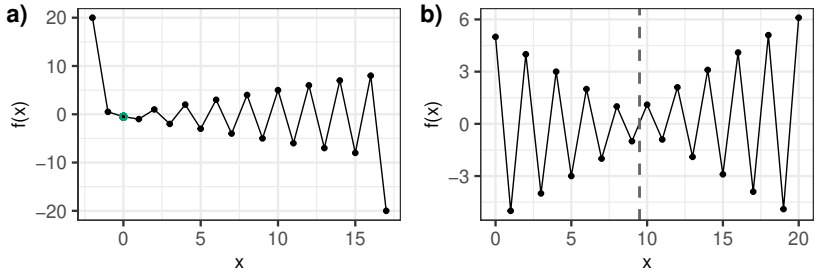

Figure 5.4.1: Worst-case examples for local and topological maintenance. *Left:* to increase the value of the marked item triggers a linear number of interchanges. *Right:* to cut the list at the *dashed line* affects every persistent pair. Both operations take time linear in the number of critical items, which for the two time series are all or almost all items.

The worst-case inputs are displayed in Figure 5.4.1. For local maintenance we define a function $w_\ell : \mathbb{Z} \to \mathbb{R}$ over n items for some even $n > 5$ as

$$\begin{aligned} w_\ell(-2) = n, \quad w_\ell(-1) = 0.5, \quad w_\ell(0) = -0.5, \quad w_\ell(n-3) &= -n \\ w_\ell(i) = (-1)^i \cdot \left\lfloor \frac{i+1}{2} \right\rfloor \quad \text{for } 0 < i < n-3. & \end{aligned} \tag{11}$$

The linear-time case for topological maintenance is a function $w_t : \mathbb{Z} \to \mathbb{R}$ defined over n items for some odd $n > 4$, which we give below. First, we define an auxiliary function

$$s(i) = (-1)^i \cdot \left(\left\lfloor \frac{i}{2} \right\rfloor + 1 \right). \tag{12}$$

The function w_t consists of two stages derived from s:

$$\begin{aligned} w_t(i) &= -s\left(\left\lfloor \frac{n}{2} \right\rfloor - 1 - i \right) && \text{for } 0 \le i < \left\lfloor \frac{n}{2} \right\rfloor, \\ w_t(i) &= s\left(i - \left\lfloor \frac{n}{2} \right\rfloor \right) + 0.1 && \text{for } \left\lfloor \frac{n}{2} \right\rfloor \le i < n. \end{aligned} \tag{13}$$

Consider first the operation that increases the value $w_\ell(0)$ of the marked item in the left panel of Figure 5.4.1. When its value passes that of its left neighbor, an anti-cancellation adds the banana they span to the banana tree. As we increase the value further, the marked item interchanges with the maxima to its right, each time moving up one level within the sequence of windows within which its window is nested. When the marked item finally becomes the global maximum, it will have interchanged with all other maxima, which takes $\Theta(n)$ time. For $n = 10^2$ and $n = 10^6$, we measured 3.4μs and 4.7ms on average, respectively. This corresponds to a median speedup of 0.05 and 0.02 if compared with `Gudhi`, which is really a slowdown by a factor of 20 and 50, respectively.

Consider second the operation that cuts the time series in the right panel of Figure 5.4.1 in the middle, which is marked by the vertical dashed line. We observe an average running time between 14μs at $n = 10^2$ and 98ms at $n = 10^6$. Compare this with an average running time of 10μs at $n = 10^2$ and 80ms at $n = 10^6$ for concatenating the two lists back to the original length. This comparison agrees with the earlier observation that gluing banana trees is slightly faster than splitting them. Table 5.4.1 lists the speedup if compared to `Gudhi`, which for a number less than 1 is really a slowdown. To create

Table 5.4.1: Performance of splitting and gluing banana trees for the example time series in the right panel of Figure 5.4.1, showing the median slowdown/speedup if compared to `Gudhi`. The parameter interpolates between these time series at $\lambda = 0$ and unbiased random walks at $\lambda = 1$.

	split				glue			
	$\lambda = 0$	10^{-4}	10^{-2}	10^{0}	$\lambda = 0$	10^{-4}	10^{-2}	10^{0}
$n = 10^2$	0.04	0.04	0.04	0.30	0.07	0.07	0.09	0.55
$n = 10^4$	0.02	0.02	4.60	10.00	0.05	0.06	7.80	18.00
$n = 10^6$	0.04	3.60	41.00	100.00	0.05	3.70	55.00	102.00

a more informative experiment, we interpolate between these time series and unbiased walks, with drastic improvements even for small $\lambda > 0$; see Table 5.4.1. Indeed, the noise quickly flattens the banana tree by decreasing the length of trails, which explains the improvement.

5.4.2 Quasi-periodic Data

Random walks are not periodic, but real-world time series often exhibit some amount of periodicity, at least approximately. To study banana trees under these conditions, we construct what we call *quasi-periodic* inputs. This term does not have a mathematical definition, and for our experiments just means a periodic signal that is randomly perturbed. Specifically, we generate such input by modifying the iterative construction in (10):

$$r(0) = 0 \quad \text{and} \quad r(i) = r(i-1) + \eta_i \quad \text{for } 1 \leq i < n, \tag{14}$$

in which the $\eta_i \sim \mathcal{N}(\mu_i, \sigma)$ are independent and normally distributed, with the mean changing periodically, $\eta_i = \sin(2\pi\omega i)$, as governed by the *frequency parameter* $0 \leq \omega < 1$, which we fix to $\omega = 5/n$. It will be useful to compare the influence of the standard deviation, so we no longer assume $\sigma = 1$.

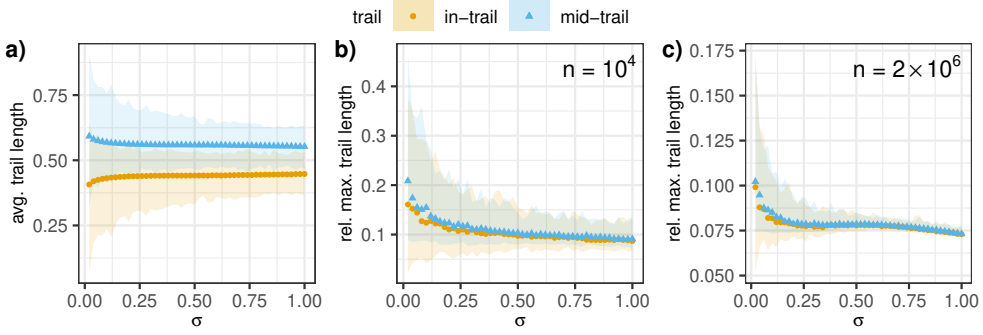

Figure 5.4.2: Measuring trail lengths. The in-trails and mid-trails are marked with *orange dots* and *blue triangles*, respectively, and the ribbons extend from the minimum to the maximum observed values. *Left:* the average length of in-trails and mid-trails depending on the standard deviation and averaged over all input sizes. *Middle and right:* the fraction of nodes on the longest in-trail and mid-trail, again depending on the standard deviation but now averaged over inputs of size 10^4 and $2 \cdot 10^6$, respectively.

For large σ, the banana trees of the time series show the features we have seen for unbiased random walks, while for small σ, they locally behave like

biased random walks. For example, for $\sigma = 1$, we observe about 37% of the items to be critical, while for $\sigma = 0.5$ and $\sigma = 0.02$ only about 21% and 1% of the items are critical, on average. Like the number of critical items, also the maximum nesting depth for quasi-periodic signals is similar to that of random walks: it scales like $\log n$ for large σ and decreases as σ goes to 0. In constrast, the average nesting depth is independent of n, ranging from 2 to 4.7 with mean 2.4. The behavior of the trail length is illustrated in Figure 5.4.2. As shown in the left panel, the average trail length is about 0.5, with consistently shorter in-trails than mid-trails, on average. This difference is more pronounced for small σ, when the quasi-periodic signal behaves more like a biased rather than unbiased random walk. The middle and right panels show how the fraction of internal nodes on the longest trail increases as σ goes to 0.

We conclude that banana trees on quasi-periodic signals with large standard deviation resemble unbiased random walks in terms of their structural parameters, and drift toward biased random walks as the standard deviation goes to 0.

5.4.3 Real-World Examples

We analyze structural properties of banana trees constructed on three types of real-world time series:

- electrocardiography (ECG) data from PhysioNet [98–100]; limiting ourselves to recordings of length at least 10^5, each separated into up to 12 leads, gives 32443 time series in total;
- audio data from EasyCom [101], of which we use 1336 audio files;
- physical activity data from PAMAP2 [102] (temperature, heart rate along with recordings from accelerometers, gyroscopes, and magnetometers), of which we use 560 time series.

We preprocess the data by first replacing multiple consecutive occurences of the same value by a single one, then adding uniformly distributed noise to ensure all values are distinct, making sure the noise is small enough so it breaks ties but does not otherwise alter the ordering.

Table 5.4.2: Structural properties of the banana trees constructed for time series from three databases. The maximum nesting depth is measured relative to the depth we observe for unbiased random walks with the same fraction of critical items.

	fraction of critical items	max nesting depth			average length	
		avg	min	max	in-trail	mid-trail
PhysioNet	0.24	1.2	0.8	6.3	0.48	0.52
EasyCom	0.22	2.7	1.6	4.9	0.48	0.52
PAMAP2	0.48	1.9	1.2	3.2	0.49	0.51

We observe that the average max nesting depth for the three data sets is only a small factor larger than for unbiased random walks; see the middle three columns in Table 5.4.2. The largest maximum nesting depth for PhysioNet data is a bit higher, by a factor 6.3, but 99% of the time series from this data have maximum nesting depth at most 1.68 times that for unbiased

random walks. The average trail lengths are again similar to those of unbiased random walks—see the last two columns, with standard deviation about 0.05 throughout—and so are the fractions of items in the longest trails, whose geometric means are 0.8%, 0.008%, 0.1% for the in-trails and 0.8%, 0.008%, 0.2% for the mid-trails in the three data sets, compared with about 0.3% and 0.4% in unbiased random walks.

5.5 Discussion

We provide an implementation of the banana tree data structure and supply experimental evidence demonstrating its efficiency. The work reported in this paper confirms the theoretical advantages of banana trees over static alternatives and opens up avenues for further theoretical inquiries and practical applications. One intriguing question that arises in the context of quasi-periodic data is whether we can determine the (possibly varying) period of the data without prior knowledge, or provide meaningful quantification of the extent to which a signal deviates from being periodic.

The potential applications of banana trees extend beyond academic research into practical, real-world scenarios. Comparisons with other methods and identifying impactful use cases represents a promising next step to provide valuable insights in fields such as healthcare, where real-time analysis of patient data is crucial, or finance, where the swift analysis of market data is essential.

6 DIFFERENTIALLY PRIVATE ALGORITHMS FOR GRAPHS UNDER CONTINUAL OBSERVATION

6.1 Introduction

Differential privacy aims to protect individuals whose data becomes part of an increasing number of data sets and is subject to analysis. A differentially private algorithm guarantees that its output depends only very little on an individual's contribution to the input data. Roughly speaking, an algorithm is *ϵ-differentially private* if the probability that it outputs O on data set D is at most an e^{ϵ}-factor of the probability that it outputs O on any adjacent data set D'. Two data sets are *adjacent* if they differ only in the data of a single user. Differential privacy was introduced in the setting of databases [42, 43], where users (entities) are typically represented by rows and data is recorded in columns. An important notion that allowed for the development of generic techniques and tools (like the Laplace and the exponential mechanism) is the *sensitivity* of a function f: the static sensitivity ρ of f is the maximum $|f(D) - f(D')|$ over all adjacent pairs D, D'. Differential privacy was later generalized to a more challenging setting, where data evolves over time [54, 55]: a differentially private algorithm under *continual observation* must provide the same privacy guarantees as before, but for a sequence (or stream) of data sets instead of just a single data set. Often, this sequence results from updates to the original data set that arrive over time. In this setting, the presence or absence of a single user in one update can affect the algorithm's output on all future data sets, i.e., two adjacent databases can differ on all future outputs and, thus, have infinite sensitivity.

In this paper, we study differentially private graph algorithms under continual observation, i.e., for *dynamic* graph problems. The input is a sequence of graphs that results from node or edge updates, i.e., insertions or deletions. *Partially dynamic* algorithms only allow either insertions or deletions, *fully dynamic* algorithms allow both. After each update, the algorithm has to output a solution for the current input, i.e., the algorithm outputs a sequence of answers that is equally long as the input sequence. For differentially private graph algorithms two notions of *adjacency of graph sequences* exist: node-adjacency and edge-adjacency. Two graph sequences are *edge-adjacent* if they only differ in a single insertion or deletion of an edge. Similarly, two graph sequences are *node-adjacent* if they only differ in an insertion or deletion of a node.[a]

We initiate the study of differentially private algorithms for *non-local* partially dynamic graph problems. We consider a problem *non-local* if its (optimum) value cannot be derived from the graph's frequency histogram of constant-size subgraphs and call it *local* otherwise. Non-local problems include the cost of the minimum spanning tree, the weight of the global and *s-t* minimum cut, and the density of the densest subgraph. We also give improved

a Of course, a graph can also be represented by a database, where, e.g., every row corresponds to an edge, but as we present algorithms that solve graph algorithmic problems we use the graph-based terminology through the paper.

algorithms for local graph problems and show various lower bounds on the additive error for differentially private dynamic graph algorithms.

LOCAL PROBLEMS. The only prior work on differentially private dynamic algorithms is an algorithm by Song et al. [53] for various *local* graph problems such as counting high-degree nodes, triangles and other constant-size subgraphs. We present an algorithm for these local problems that improves the additive error by a factor of $\sqrt{T}/\log^{3/2} T$, where T is the length of the update sequence. We also give the first differentially private partially dynamic algorithm for the value of the minimum spanning tree. Table 6.1.1 on Page 114 lists upper and lower bounds for these results, where n is the number of nodes in the graph, W is the maximum edge weight (if applicable), D is the maximum node degree, ϵ is an arbitrarily small positive constant, and δ is the failure probability of the algorithm. We state below our main contributions in more detail. The update time of all our algorithms is linear in $\log T$ plus the time needed to solve the corresponding non-differentially private dynamic graph problem.

Theorem 6.1.1 (see Section 6.3)**.** *Let $\varepsilon, \delta > 0$. There exist an ε-edge-differentially as well as an ε-node-differentially private algorithm for partially dynamic minimum spanning tree, edge count, the number of high-degree nodes, the degree histogram, triangle count and k-star count that with probability at least $1 - \delta$ give an answer with additive error as shown in Table 6.1.1.*

NON-LOCAL PROBLEMS. For non-local problems we present an algorithm that, by allowing a small multiplicative error, can obtain differentially private partially dynamic algorithms for a broad class of problems that includes the aforementioned problems. Table 6.1.2 lists our results for some common graph problems. The algorithm achieves the following performance.

Theorem 6.1.2 (see Theorem 6.4.1)**.** *Let $\varepsilon, \beta, \delta, r > 0$ and let f be a function with range $[1, r]$ that is monotone on all input sequences and has sensitivity ρ. There exists an ε-differentially private dynamic algorithm with multiplicative error $(1+\beta)$, additive error $O(\rho \log(r) \log(T)/\log(1+\delta))$ and failure probability δ that computes f.*

Note that for partially dynamic graph algorithms it holds that $T = O(n^2)$. Thus for local problems the bounds presented in Table 6.1.1 are superior to the bounds in Table 6.1.2.

LOWER BOUNDS. We complement these upper bounds by also giving some lower bounds on the additive error of any differentially private dynamic graph algorithm. For the problems in Table 6.1.1 we show lower bounds of $\Omega(W \log T)$, resp. $\Omega(\log T)$. Note that these lower bounds apply to the partially dynamic as well as to the fully dynamic setting.

The above notion of differential privacy is also known as *event-level* differential privacy, where two graph sequences differ in at most one "event", i.e., one update operation. A more challenging notion is *user-level* differential privacy. Two graph sequences are edge-adjacent *on user-level* if they differ in *any* number of updates for a single edge (as opposed to *one* update for a single edge in the case of the former *event-level* adjacency). Note that requiring user-level edge-differential privacy is a more stringent requirement on the

Table 6.1.1: Additive errors for partially dynamic ε-differentially private algorithms with failure probability δ. We use D for the maximum degree and W for the maximum edge weight, n for the maximum number of nodes of any graph in the input sequence, and $\Lambda = \log(1/\delta)/\varepsilon$. The upper bounds follow from Corollary 6.3.5 and Table 6.3.1. See Section 6.5 for results on event-level lower bounds and Section 6.6 for user-level lower bounds.

Graph function	partially dynamic		fully dynamic	
	edge-adj. event-level	node-adj. event-level	edge-adj. event-level	edge-adj. user-level
min. spanning tree	$\Omega(W \log T)$, $O(W \log^{3/2} T \cdot \Lambda)$	$\Omega(W \log T)$, $O(DW \log^{3/2} T \cdot \Lambda)$	$\Omega(W \log T)$	$\Omega(nW)$
min. cut, max. matching	$\Omega(W \log T)$	$\Omega(W \log T)$	$\Omega(W \log T)$	$\Omega(nW)$
edge count	$\Omega(\log T)$, $O(\log^{3/2} T \cdot \Lambda)$	$\Omega(D \log T)$, $O(D \log^{3/2} T \cdot \Lambda)$	$\Omega(\log T)$, $O(\log^{3/2} T \cdot \Lambda)$	$\Omega(n^2)$
high-degree nodes	$\Omega(\log T)$, $O(\log^{3/2} T \cdot \Lambda))$	$\Omega(D \log T)$, $O(D \log^{3/2} T \cdot \Lambda)$	$\Omega(\log T)$	$\Omega(n)$
degree histogram	$\Omega(\log T)$, $O(D \log^{3/2} T \cdot \Lambda))$	$\Omega(D \log T)$, $O(D^2 \log^{3/2} T \cdot \Lambda)$	$\Omega(\log T)$	$\Omega(n)$
triangle count	$\Omega(\log T)$, $O(D \log^{3/2} T \cdot \Lambda))$	$\Omega(D \log T)$, $O(D^2 \log^{3/2} T \cdot \Lambda)$	$\Omega(\log T)$	$\Omega(n^3)$
k-star count	$\Omega(\log T)$, $O(D^k \log^{3/2} T \cdot \Lambda)$	$\Omega(D \log T)$, $O(D^k \log^{3/2} T \cdot \Lambda)$	$\Omega(\log T)$	$\Omega(n^{k+1})$

Table 6.1.2: Private algorithms with failure probability δ with additional multiplicative error of $(1+\beta)$ for arbitrary $\beta > 0$. We use $\Lambda = 1/(\varepsilon\delta\log(1+\beta))$, D for the maximum degree, W for the maximum edge weight and n for the maximum number of nodes of any graph in the input sequence.

Graph function	partially dynamic, event-level	
	edge-adjacency	node-adjacency
minimum cut	$O(W\log(nW)\log(T)\cdot\Lambda)$	$O(DW\log(nW)\log(T)\cdot\Lambda)$
densest subgraph	$O(\log(n)\log(T)\cdot\Lambda)$	$O(\log(n)\log(T)\cdot\Lambda)$
minimum s,t-cut	$O(W\log(nW)\log(T)\cdot\Lambda)$	$O(DW\log(nW)\log(T)\cdot\Lambda)$
maximum matching	$O(W\log(nW)\log(T)\cdot\Lambda)$	$O(W\log(nW)\log(T)\cdot\Lambda)$

algorithm than event-level edge-differential privacy.[b] We show strong lower bounds for edge-differentially private algorithms on user-level for a broad class of dynamic graph problems.

Theorem 6.1.3 (informal, see Theorem 6.6.3)**.** *Let f be a function on graphs, and let G_1, G_2 be arbitrary graphs. There exists a $T \geq 1$ so that any ε-edge-differentially private dynamic algorithm on user-level that computes f must have additive error $\Omega(|f(G_1) - f(G_2)|)$ on input sequences of length at least T.*

This theorem leads to the lower bounds for fully dynamic algorithms stated in Table 6.1.1.

Technical contribution.

LOCAL PROBLEMS. Our algorithms for local problems (Theorem 6.1.1) incorporate a counting scheme by Chan et al. [55] and the *difference sequence technique* by Song et al. [53]. The difference sequence technique addresses the problem that two adjacent graphs might differ on all outputs starting from the point in the update sequence where their inputs differ. More formally, let $f_{\mathcal{G}}(t)$ be the output of the algorithm after operation t in the graph sequence $\mathcal{G}$. Then the *continuous global sensitivity* $\sum_{t=1}^{T} |f_{\mathcal{G}}(t) - f_{\mathcal{G}'}((t)|$ might be $\Theta(\rho T)$. Using the "standard" Laplacian mechanism for such a large sensitivity would, thus, lead to an additive error linear in T. The idea of [53] is to use instead the *difference sequence of f* defined as $\Delta f(t) = f(t) - f(t-1)$, as they observed that for various local graph properties the continuous global sensitivity of the difference sequence, i.e., $\sum_{t=1}^{T} |\Delta f_{\mathcal{G}}(t) - \Delta f_{\mathcal{G}'}((t)|$ can be bounded by a function independent of T. However, their resulting partially dynamic algorithms still have an additive error linear in $\sqrt{T}$. We show how to combine the continuous global sensitivity of the difference sequence with the binary counting scheme of Chan et al. [55] to achieve partially dynamic algorithms with additive error linear in $\log^{3/2} T$.

Furthermore we show that the approach based on the continuous global sensitivity of the difference sequence fails, if the presence or absence of a node or edge can significantly change the target function's value for all of the subsequent graphs. In particular, we show that for several graph problems like minimum cut and maximum matching changes in the function value between adjacent graph sequences can occur at every time step even for partially dynamic sequences, resulting in a continuous global sensitivity of

b Node-adjacency on user-level is defined accordingly but not studied in this paper.

the difference sequence that is linear in T. This implies that this technique cannot be used to achieve differentially private dynamic algorithms for these problems.

NON-LOCAL PROBLEMS. We leverage the fact that the sparse vector technique [56] provides negative answers to threshold queries with little effect on the additive error to approximate monotone functions f on graphs under continual observation (e.g., the minimum cut value in an incremental graph) with multiplicative error $(1+\beta)$: If r is the maximum value of f, we choose thresholds $(1+\beta),\dots,(1+\beta)^{\log_{1+\beta}(r)}$ for the queries. This results in at most $\log_{1+\beta}(r)$ positive answers, which affect the additive error linearly, while the at most T negative answers affect the additive error only logarithmically instead of linearly.

LOWER BOUNDS. Dwork et al. [54] had given a lower bound for counting in binary streams. We reduce this problem to partially dynamic graph problems on the event-level to achieve the event-level lower bounds.

For the user-level lower bounds we assume by contradiction that an ϵ-differentially private dynamic algorithm $\mathcal{A}$ with "small" additive error exists and construct an exponential number of graph sequences that are all user-level "edge-close" to a simple graph sequence $\mathcal{G}'$. Furthermore any two such graph sequences have at least one position with two very different graphs such that $\mathcal{A}$ (due to its small additive error) must return two different outputs at this position, which leads to two different output sequences if $\mathcal{A}$ answers within its error bound. Let O_i be the set of accurate output sequences of $\mathcal{A}$ on one of the graph sequences G_i. By the previous condition $O_i \cap O_j = \emptyset$ if $i \neq j$. As G_i is "edge-close" to $\mathcal{G}'$, there is a relatively large probability (depending on the degree of "closeness") that O_i is output when $\mathcal{A}$ runs on $\mathcal{G}'$. This holds for all i. However, since $O_i \cap O_j = \emptyset$ if $i \neq j$ and we have constructed exponentially many graph sequences G_i, the sum of these probabilities over all i adds up to a value larger than 1, which gives a contradiction. The proof is based on ideas of a lower bound proof for databases in [54].

Related Work.

Differential privacy, developed in [42, 43], is the de facto gold standard of privacy definitions and several lines of research have since been investigated [51, 103–107]. In particular, differentially private algorithms for the release of various graph statistics such as subgraph counts [108–112], degree distributions [113–115], minimum spanning tree [50], spectral properties [116, 117], cut problems [106, 117, 118], and parameter estimation for special classes of graphs [119] have been proposed. Dwork et al. [54] and Chan et al. [55] extended the analysis of differentially private algorithms to the regime of continual observation, i.e., to input that evolves over time. Since many data sets in applications are evolving data sets, this has lead to results for several problems motivated by practice [120–124]. Only one prior work analyzes evolving graphs: Song et al. [53] study problems in incremental bounded-degree graphs that are functions of local neighborhoods. Our results improve all bounds for undirected graphs initially established in [53] by a factor of $\sqrt{T}/\log^{3/2} T$ in the additive error.

6.2 Preliminaries

6.2.1 Graphs and Graph Sequences

We consider undirected graphs $G = (V, E)$, which change dynamically. Graphs may be edge-weighted, in which case $G = (V, E, w)$, where $w : E \to \mathbb{N}$. The evolution of a graph is described by a *graph sequence* $\mathcal{G} = (G_1, G_2, \dots)$, where $G_t = (V_t, E_t)$ is derived from G_{t-1} by applying updates, i.e., inserting or deleting nodes or edges. We denote by $|\mathcal{G}|$ the length of $\mathcal{G}$, i.e., the number of graphs in the sequence. At time t we delete a set of nodes ∂V_t^- along with the corresponding edges ∂E_t^- and insert a set of nodes ∂V_t^+ and edges ∂E_t^+. More formally, $V_t = (V_{t-1} \setminus \partial V_t^-) \cup \partial V_t^+$ and $E_t = (E_{t-1} \setminus \partial E_t^-) \cup \partial E_t^+$, with initial node and edge sets V_0, E_0, which may be non-empty. If a node v is deleted, then all incident edges are deleted at the same time, i.e., if $v \in \partial V_t^-$, then $(u, v) \in \partial E_t^-$ for all $(u, v) \in E_{t-1}$. Both endpoints of an edge inserted at time t need to be in the graph at time t, i.e., $\partial E_t^+ \subseteq ((V_{t-1} \setminus \partial V_t^-) \cup \partial V_t^+) \times ((V_{t-1} \setminus \partial V_t^-) \cup \partial V_t^+)$. The tuple $(\partial V_t^+, \partial V_t^-, \partial E_t^+, \partial E_t^-)$ is the *update* at time t. For any graph G and any update u, let $G \oplus u$ be the graph that results from applying u on G.

A graph sequence is *incremental* if $\partial E_t^- = \partial V_t^- = \emptyset$ at all time steps t. A graph sequence is *decremental* if $\partial E_t^+ = \partial V_t^+ = \emptyset$ at all time steps t. Incremental and decremental graph sequences are called *partially dynamic*. Graph sequences that are neither incremental nor decremental are *fully dynamic*.

Our goal is to continually release the value of a *graph function* f which takes a graph as input and outputs a real number. In other words, given a graph sequence $\mathcal{G} = (G_1, G_2, \dots)$ we want to compute the sequence $f(\mathcal{G}) = (f(G_1), f(G_2), \dots)$. We write $f(t)$ for $f(G_t)$. Our algorithms will compute an update to the value of f at each time step, i.e., we compute $\Delta f(t) = f(t) - f(t-1)$. We call the sequence Δf the *difference sequence of* f.

Given a graph function g the *continuous global sensitivity* $\mathrm{GS}(g)$ of g is defined as the maximum value of $\|g(S) - g(S')\|_1$ over all adjacent graph sequences S, S'. We will define adjacency of graph sequences below. In our case, we are often interested in the continuous global sensitivity of the difference sequence of a graph function f, which is given by the maximum value of $\sum_{t=1}^{T} |\Delta f_{\mathcal{G}}(t) - \Delta f_{\mathcal{G}'}(t)|$, where $\Delta f_{\mathcal{G}}$ and $\Delta f_{\mathcal{G}'}$ are the difference sequences of f corresponding to adjacent graph sequences $\mathcal{G}$ and $\mathcal{G}'$.

Two graphs are *edge-adjacent* if they differ in one edge. We also define global sensitivity of functions applied to a single graph. Let g be a graph function. Its *static global sensitivity* $\mathrm{GS}_{\mathrm{static}}(g)$ is the maximum value of $|g(G) - g(G')|$ over all edge-adjacent graphs G, G'.

6.2.2 Differential Privacy

The range of an algorithm $\mathcal{A}$, $\mathrm{Range}(\mathcal{A})$, is the set of all possible output values of $\mathcal{A}$. We denote the Laplace distribution with mean μ and scale b by $\mathrm{Lap}(\mu, b)$. If $\mu = 0$, we write $\mathrm{Lap}(b)$.

Definition 6.2.1 (ε-differential privacy)**.** A randomized algorithm $\mathcal{A}$ is *ε-differentially private* if for any two *adjacent* databases B, B' and any $S \subseteq \mathrm{Range}(\mathcal{A})$ we have $\Pr[\mathcal{A}(B) \in S] \leq e^{\varepsilon} \cdot \Pr[\mathcal{A}(B') \in S]$. The parameter ε is called the *privacy loss* of $\mathcal{A}$.

To apply Definition 6.2.1 to graph sequences we now define adjacency for graph sequences. First, we define edge-adjacency, which is useful if the data to be protected is associated with the edges in the graph sequence. Then, we define node-adjacency, which provides stronger privacy guarantees.

Definition 6.2.2 (Edge-adjacency)**.** Let $\mathcal{G}, \mathcal{G}'$ be graph sequences as defined above with associated sequences of updates (∂V_t^-), (∂V_t^+), (∂E_t^-), (∂E_t^+) and $(\partial V_t^{-\prime})$, $(\partial V_t^{+\prime})$, $(\partial E_t^{-\prime})$, $(\partial E_t^{+\prime})$. Let $\partial V_t^- = \partial V_t^{-\prime}$ and $\partial V_t^+ = \partial V_t^{+\prime}$ for all t. Let the initial node and edge sets for $\mathcal{G}$ and $\mathcal{G}'$ be $V_0 = V_0'$ and $E_0 = E_0'$. Assume w.l.o.g. that $\partial E_t^{-\prime} \subseteq \partial E_t^-$ and $\partial E_t^{+\prime} \subseteq \partial E_t^+$ for all t. The graph sequences $\mathcal{G}$ and $\mathcal{G}'$ are *adjacent on* e^* if $|\mathcal{G}| = |\mathcal{G}'|$, there exists an edge e^* and one of the following statements holds:

1. $\partial E_t^- = \partial E_t^{-\prime} \forall\, t$ and $\exists t^*$ such that $\partial E_{t^*}^+ \setminus \partial E_{t^*}^{+\prime} = \{e^*\}$ and $\partial E_t^+ = \partial E_t^{+\prime} \forall\, t \neq t^*$;
2. $\partial E_t^+ = \partial E_t^{+\prime} \forall\, t$ and $\exists t^*$ such that $\partial E_{t^*}^- \setminus \partial E_{t^*}^{-\prime} = \{e^*\}$ and $\partial E_t^- = \partial E_t^{-\prime} \forall\, t \neq t^*$;

Remark. If $\mathcal{G}$ and $\mathcal{G}'$ are edge-adjacent, then for any index i the graphs at index i in the two sequences are edge-adjacent.

Two edge-adjacent graph sequences differ in either the insertion or the deletion of a single edge e^*. There are several special cases that fit into this definition. For example, we may have $\partial V_t^- = \partial V_t^{-\prime} = \partial V_t^+ = \partial V_{t'}^+ = \emptyset$, so only edge updates would be allowed. Similarly, we can use the definition in the incremental setting by assuming $\partial V_t^- = \partial V_t^{-\prime} = \partial E_t^- = \partial E_t^{-\prime} = \emptyset$.

The definition of node-adjacency is similar to that of edge-adjacency, but poses additional constraints on the edge update sets.

Definition 6.2.3 (Node-adjacency)**.** Let $\mathcal{G}, \mathcal{G}'$ be graph sequences as defined above with associated sequences of updates (∂V_t^-), (∂V_t^+), (∂E_t^-), (∂E_t^+) and $(\partial V_t^{-\prime})$, $(\partial V_t^{+\prime})$, $(\partial E_t^{-\prime})$, $(\partial E_t^{+\prime})$. Assume w.l.o.g. that $\partial V_t^{-\prime} \subseteq \partial V_t^-$ and $\partial V_t^{+\prime} \subseteq \partial V_t^+$ for all t. The graph sequences $\mathcal{G}$ and $\mathcal{G}'$ are *adjacent on* v^* if $|\mathcal{G}| = |\mathcal{G}'|$, there exists a node v^* and one of the following statements holds:

1. $\partial V_t^- = \partial V_t^{-\prime} \forall\, t$ and $\exists t^*$ such that $\partial V_t^+ \setminus \partial V_t^{+\prime} = \{v^*\}$ and $\partial V_t^+ = \partial V_t^{+\prime} \forall\, t \neq t^*$;
2. $\partial V_t^+ = \partial V_t^{+\prime} \forall\, t$ and $\exists t^*$ such that $\partial V_t^- \setminus \partial V_t^{-\prime} = \{v^*\}$ and $\partial V_t^- = \partial V_t^{-\prime} \forall\, t \neq t^*$;

Additionally, all edges in ∂E_t^+ and ∂E_t^- are incident to at least one node in ∂V_t^+ and ∂V_t^-, respectively. Lastly, we require that $\partial E_t^{+\prime}$ $(\partial E_t^{-\prime})$ is the maximal subset of ∂E_t^+ (∂E_t^-) that does not contain edges incident to v^*.

We define the following notions of differential privacy based on these definitions of adjacency.

Definition 6.2.4. An algorithm is *ε-edge-differentially private* (on event-level) if it is ε-differentially private when considering edge-adjacency. An algorithm is *ε-node-differentially private* (on event-level) if it is ε-differentially private when considering node-adjacency.

When explicitly stated, we consider a stronger version of ε-differential privacy, which provides adjacency on user-level. While adjacency on event-level only allows two graph sequences to differ in a single update, user-level adjacency allows any number of updates to differ as long as they affect the same edge (for edge-adjacency) or node (for node-adjacency), respectively.

Definition 6.2.5. Let $\mathcal{G} = (G_1, \ldots), \mathcal{G}' = (G'_1, \ldots)$ be graph sequences. The two sequences are edge-adjacent *on user-level* if there exists an edge e^* and a sequence of graph sequences $\mathcal{S} = (\mathcal{G}_1, \ldots, \mathcal{G}_\ell)$ so that $\mathcal{G}_1 = \mathcal{G}, \mathcal{G}_\ell = \mathcal{G}'$ and, for any $i \in [\ell - 1]$, $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$ are edge-adjacent on e^*. An algorithm is ε-edge-differentially private on user-level if it is ε-differentially private when considering edge-adjacency on user-level.

6.2.3 Counting Mechanisms

Some of our algorithms for releasing differentially private estimates of functions on graph sequences rely on algorithms for counting in streams.

A *stream* $\sigma = \sigma(1)\sigma(2)\cdots$ is a string of *items* $\sigma(i) \in \{L_1, \ldots, L_2\} \subseteq \mathbb{Z}$, where the i-th item is associated with the i-th *time step*. A *binary stream* has $L_1 = 0$ and $L_2 = 1$. We denote the length of a stream, i.e., the number of time steps in the stream, by $|\sigma|$. Stream σ and σ' are adjacent if $|\sigma| = |\sigma'|$ and if there exists one and only one t^* such that $\sigma(t^*) \neq \sigma'(t^*)$ and $\sigma(t) = \sigma'(t)$ for all $t \neq t^*$.

A *counting mechanism* $\mathcal{A}(\sigma)$ takes a stream σ and outputs a real number for every time step. For all time steps t, $\mathcal{A}$'s output at time t is independent of all $\sigma(i)$ for $i > t$. At each time t a counting mechanism should estimate the count $c(t) = \sum_{i=1}^{t} \sigma(i)$.

Following Chan et al. [55] we describe our mechanisms in terms of *p-sums*, which are partial sums of the stream over a time interval. For a p-sum p we denote the beginning and end of the time interval by $\mathrm{start}(p)$ and $\mathrm{end}(p)$, respectively. With this notation the value of p is $\sum_{t=\mathrm{start}(p)}^{\mathrm{end}(p)} \sigma(t)$. To preserve privacy we add noise to p-sums and obtain *noisy p-sums*: given a p-sum p, a noisy p-sum is $\hat{p} = p + \gamma$, where γ is drawn from a Laplace-distribution.

6.2.4 Sparse Vector Technique

The *sparse vector technique* (SVT) was introduced by Dwork et al. [56] and was subsequently improved [125, 126]. SVT can be used to save privacy budget whenever a sequence of threshold queries $f_1, \ldots, f_T$ is evaluated on a database, but only $c \ll T$ queries are expected to exceed the threshold. Here, a threshold query asks whether a function f_i evaluates to a value above some threshold t_i on the input database. Using SVT, only queries that are answered positively reduce the privacy budget. We use the following variant of SVT, which is due to Lyu et al.

Lemma 6.2.6 ([57])**.** *Let D be a database, $\epsilon, \rho, c > 0$ and let $(f_1, t_1), \ldots$ be a sequence of mappings f_i from input databases to $\mathbb{R}$ and thresholds $t_i \in \mathbb{R}$, which may be generated adaptively one after another so that $\rho \geq \max_i \mathrm{GS}_{\mathrm{static}}(f_i)$. Algorithm 6.2.1 is ϵ-private.*

Algorithm 6.2.1: SVT algorithm [57]

Function `InitializeSvt`(D, ρ, ϵ, c)**:**
  $\epsilon_1 \leftarrow \epsilon/2, \zeta \leftarrow \mathrm{Lap}(\rho/\epsilon_1)$, $\epsilon_2 \leftarrow \epsilon - \epsilon_1, count \leftarrow 0$
Function `ProcessSvtQuery`(f_i, t_i)**:**
  $\nu_i \leftarrow \mathrm{Lap}(2c\rho/\epsilon_2)$
  if $count \geq c$ **then**
    return *abort*
  if $f_i(D) + \nu_i \geq t_i + \zeta$ **then**
    $count \leftarrow count + 1$, **return** $\top$
  else
    return $\bot$

6.3 Mechanisms Based on Continuous Global Sensitivity

Some of our mechanisms for privately estimating graph functions are based on mechanisms for counting in streams. In both settings, we compute the sum of a sequence of numbers and we will show that the mechanisms for counting can be transferred to the graph setting. However, there are differences in the analysis. In counting, the input streams differ at only one time step. This allows us to bound the difference in the true value between adjacent inputs and leads to low error. In the graph setting, the sequence of numbers can vary at many time steps. Here however, we use properties of the counting mechanisms to show that the total difference for this sequence can still be bounded, which results in the same error as in the counting setting.

We first generalize the counting mechanisms by Chan et al. [55] to streams of integers with bounded absolute value, and then transfer them to estimating graph functions.

6.3.1 Non-Binary Counting

We generalize the counting mechanisms of Chan et al. [55] to streams of numbers in $\{-L, \ldots, L\}$, for some constant L. We view these algorithms as releasing noisy p-sums from which the count can be estimated. The generic algorithm is outlined in Algorithm 6.3.1 on page 121.

The algorithm releases a vector of noisy p-sums over T time steps, such that at every time step the noisy p-sums needed to estimate the count up to this time are available. Each of the noisy p-sums is computed exactly once. See the proof of Corollary 6.3.5 for an example on how to use p-sums.

In order to achieve the desired privacy loss the mechanisms need to meet the following requirements. Let $\mathcal{A}$ be a counting mechanism. We define $\mathrm{Range}(\mathcal{A}) = \mathbb{R}^k$, where k is the total number of p-sums used by $\mathcal{A}$ and every item of the vector output by $\mathcal{A}$ is a p-sum. We assume that the time intervals represented by the p-sums in the output of $\mathcal{A}$ are deterministic and only depend on the length T of the input stream. For example, consider any two streams σ and σ' of length T. The ℓ-th element of $\mathcal{A}(\sigma)$ and $\mathcal{A}(\sigma')$ will be p-sums of the same time interval $[\mathrm{start}(\ell), \mathrm{end}(\ell)]$. We further assume that the p-sums are computed independently from each other in the following way: $\mathcal{A}$ computes the true p-sum and then adds noise from $\mathrm{Lap}(z \cdot \varepsilon^{-1})$, where z is

a sensitivity parameter. To analyze the error of the algorithms we need the

Algorithm 6.3.1: Generic counting mechanism

Input: privacy loss ε, stream σ of items $\{L_1, \dots, L_2\}$ with $|\sigma| = T$
Output: vector of noisy p-sums $a \in \mathbb{R}^k$, released over T time steps
Initialization: Determine which p-sums to compute based on T
At each time step $t \in \{1, \dots, T\}$:
    Compute new p-sums $p_i, \dots, p_j$ for t
    For $\ell = i, \dots, j$:
        $\hat{p}_\ell = p_\ell + \gamma_\ell, \quad \gamma_\ell \sim \mathrm{Lap}((L_2 - L_1)\varepsilon^{-1})$
    Release new noisy p-sums $\hat{p}_i, \dots, \hat{p}_j$

following Corollary from Chan et al. [55]:

Corollary 6.3.1 (Corollary 2.9 from [55])**.** *Suppose γ_i are independent random variables, where each γ_i has Laplace distribution $\mathrm{Lap}(b_i)$. Let $Y := \sum_i \gamma_i$ and $b_M = \max_i b_i$. Let $\nu \geq \sqrt{\sum_i b_i^2}$. Suppose $0 < \delta < 1$ and $\nu > \max\left\{\sqrt{\sum_i b_i^2}, b_M \sqrt{\ln \frac{2}{\delta}}\right\}$. Then, $\Pr\left[|Y| > \nu \sqrt{8 \ln \frac{2}{\delta}}\right] \leq \delta$.*

To simplify our presentation and improve readability, we choose $\nu := \sqrt{\sum_i b_i^2} \cdot \sqrt{\ln \frac{2}{\delta}}$ and use the following slightly weaker result: with probability at least $1 - \delta$, the quantity $|Y|$ is at most $O(\sqrt{\sum_i b_i^2} \log \frac{1}{\delta})$.

The next lemma is stated informally in [55].

Lemma 6.3.2 (Observation 1 from [55])**.** *Let $\mathcal{A}$ be a counting mechanism as described in Algorithm 6.3.1 with $L_1 = 0$ and $L_2 = 1$ that releases k noisy p-sums, such that the count at any time step can be computed as the sum of at most y p-sums and every item is part of at most x p-sums. Then, $\mathcal{A}$ is $(x \cdot \varepsilon)$-differentially private, and the error is $O(\varepsilon^{-1} \sqrt{y} \log \frac{1}{\delta})$ with probability at least $1 - \delta$ at each time step.*

Proof. The error follows from Corollary 6.3.1 since the noise at each time step is the sum of at most y independent random variables with distribution $\mathrm{Lap}(1/\varepsilon)$.

We show that the vector of all noisy p-sums output by $\mathcal{A}$ preserves $(x \cdot \varepsilon)$-differential privacy. Then, since the count is estimated by summing over $(x \cdot \varepsilon)$-differentially private noisy p-sums the count itself is $(x \cdot \varepsilon)$-differentially private. As described above, $\mathrm{Range}(\mathcal{A}) = \mathbb{R}^k$, where k is the number of p-sums that the algorithm computes in total.

Let σ, σ' be two adjacent streams. We consider the joint distribution of the random variables $Z_1, \dots, Z_k$ that correspond to the noisy p-sums output by $\mathcal{A}$. Let $p_i(z_i)$, $p_i'(z_i)$ for $z_i \in \mathrm{Range}(Z_i)$ be the probability density function of Z_i when the input to $\mathcal{A}$ is σ and σ', respectively. Note that each of these densities is the probability density function of a Laplace distribution. Let $p(z_1, \dots, z_k)$, $p'(z_1, \dots, z_k)$ be the joint probability density when the input to $\mathcal{A}$ is σ and σ', respectively. Note that the random variables Z_i are continuous, since the noise

added to the p-sums is drawn from a continuous distribution over $\mathbb{R}$. Since the Z_i are independent, we have

$$p(z_1,\dots,z_k) = \prod_{i=1}^{k} p_i(z_i)$$

and

$$p'(z_1,\dots,z_k) = \prod_{i=1}^{k} p'_i(z_i).$$

We now show that these joint densities differ by at most an $\exp(x\cdot\varepsilon)$-factor for all possible outputs of $\mathcal{A}$. Let $s = (s_1,\dots,s_k)^\top \in \mathrm{Range}(\mathcal{A})$. Note that $\mathcal{A}$ outputs a vector of noisy p-sums from which the count can be computed, so each item in s is a noisy p-sum. Let $c = (c_1,\dots,c_k)^\top$ be the noiseless p-sums of stream σ corresponding to the noisy p-sums computed by $\mathcal{A}$. Let I be the set of indices to (noisy) p-sums involving time t'. By assumption, $|I| \le x$, since any item participates in at most x (noisy) p-sums.

$$\begin{aligned}
\frac{p(s)}{p'(s)} &= \prod_{i=1}^{k} \frac{p_i(s_i)}{p'_i(s_i)} = \prod_{i\in I} \frac{p_i(s_i)}{p'_i(s_i)} \\
&= \prod_{i\in I} \frac{\exp(-\varepsilon|c_i - s_i|)}{\exp(-\varepsilon|c_i \pm 1 - s_i|)} \\
&= \prod_{i\in I} \exp(\varepsilon(|c_i \pm 1 - s_i| - |c_i - s_i|)) \\
&\le \prod_{i\in I} \exp(\varepsilon|c_i \pm 1 - s_i - c_i + s_i|) \\
&= \exp(\varepsilon)^{|I|} \le \exp(x\cdot\varepsilon). \qquad (15)
\end{aligned}$$

The first equality follows from the assumption that the noisy p-sums are independent, as discussed above. Since all noiseless p-sums that do not involve the item at t' are equal, the densities $p_j(s_j)$ and $p'_j(s_j)$ cancel out for $j \notin I$, so we can take the product over just the noisy p-sums corresponding to indices in I. Plugging in the density function of the Laplace distribution and applying the reverse triangle inequality yields the first inequality. The second inequality follows from the assumption $|I| \le x$.

Now, for any $S \subseteq \mathrm{Range}(\mathcal{A})$ we have

$$\Pr[\mathcal{A}(\sigma) \in S] = \int_S p(s)ds \le \int_S e^{x\cdot\varepsilon} p'(s)ds = e^{x\cdot\varepsilon} \Pr[A(\sigma') \in S],$$

where the inequality follows from (15). Thus, $\mathcal{A}$ is indeed $(x\cdot\varepsilon)$-differentially private. □

To extend the counting mechanisms by Chan et al. to non-binary streams of values in $\{-L,\dots,L\}$, we only need to account for the increased sensitivity in the scale of the Laplace distribution. By Lemma 6.3.3, we can use the mechanisms of Chan et al. [55] to compute the sum of a stream of numbers in $\{-L,\dots,L\}$, but gain a factor $2L$ in the error.

Lemma 6.3.3 (Extension of Lemma 6.3.2)**.** *Let $\mathcal{A}$ be a mechanism as in Algorithm 6.3.1 (see page 121) with $L_1 = -L$ and $L_2 = L$ that releases k noisy p-sums, such that the count at any time step can be computed as the sum of at most y p-sums, and every item is part of at most x p-sums. Furthermore, $\mathcal{A}$ adds noise $\mathrm{Lap}(2L/\varepsilon)$ to every p-sum. Then, $\mathcal{A}$ is $(x \cdot \varepsilon)$-differentially private, and the error is $O(L\varepsilon^{-1}\sqrt{y}\log\frac{1}{\delta})$ with probability at least $1-\delta$ at each time step.*

Proof. The error follows from Corollary 6.3.1.

We proceed as in the proof of Lemma 6.3.2. Let σ, σ' be two adjacent streams that differ only at time t', i.e., $\sigma(t) = \sigma'(t)$ for all $t \neq t'$ and $\sigma(t') \neq \sigma'(t')$. We denote by $\delta' = \sigma'(t') - \sigma(t')$ the difference in the items at time t'. Note that $|\delta'| \leq 2L$, since $\sigma(t), \sigma'(t) \in \{-L, \dots, L\}$ for all t. Let $c = (c_1, \dots, c_k)^\top$ be the noiseless p-sums of stream σ corresponding to the noisy p-sums computed by $\mathcal{A}$. Let I be the set of indices to (noisy) p-sums involving time t'. By assumption, $|I| \leq x$, since any item participates in at most x (noisy) p-sums.

As in the proof of Lemma 6.3.2 we consider the joint probability density of the output of $\mathcal{A}$ when the input is σ and σ'. Note that the noisy p-sums output by $\mathcal{A}$ are independent continuous random variables with range $\mathbb{R}$. We have

$$p(z) = \prod_{i=1}^{k} p_i(z_i)$$

and

$$p'(z) = \prod_{i=1}^{k} p'_i(z_i),$$

where the p_i and p'_i are the probability density function of the individual noisy p-sums with input stream σ and σ', respectively.

Let $s = (s_1, \dots, s_k)^\top \in \mathrm{Range}(\mathcal{A})$.

$$\begin{aligned}
\frac{p(s)}{p'(s)} &= \prod_{i=1}^{k} \frac{p_i(s_i)}{p'_i(s_i)} \\
&= \prod_{i \in I} \frac{p_i(s_i)}{p'_i(s_i)} \\
&= \prod_{i \in I} \frac{\exp\left(-\frac{\varepsilon|c_i - s_i|}{2L}\right)}{\exp\left(-\frac{\varepsilon|c_i + \delta' - s_i|}{2L}\right)} \\
&= \prod_{i \in I} \exp\left(\frac{\varepsilon(|c_i + \delta' - s_i| - |c_i - s_i|)}{2L}\right) \\
&\leq \prod_{i \in I} \exp\left(\frac{\varepsilon|c_i + \delta' - s_i - c_i + s_i|}{2L}\right) \\
&= \exp\left(\frac{\varepsilon|\delta'|}{2L}\right)^{|I|} \leq \exp(x \cdot \varepsilon).
\end{aligned}$$

Thus, $\mathcal{A}$ is $x \cdot \varepsilon$-differentially private, since

$$\Pr[\mathcal{A}(\sigma) \in S] = \int_S p(s)ds \leq \int_S e^{x \cdot \varepsilon} p'(s)ds = e^{x \cdot \varepsilon} \Pr[A(\sigma') \in S]$$

for any $S \subseteq \mathrm{Range}(\mathcal{A})$. □

6.3.2 Graph Functions via Counting Mechanisms

Algorithm 6.3.2: Generic graph sequence mechanism

Input: privacy loss ε, contin. global sensitivity Γ, graph sequence $\mathcal{G} = (G_1, \ldots, G_T)$
Output: vector of noisy p-sums $a \in \mathbb{R}^k$, released over T time steps

1 **Initialization:** Determine which p-sums to compute based on T
2 At each time step $t \in \{1, \ldots, T\}$:
3 Compute $f(t)$ and $\Delta f(t) = f(t) - f(t-1)$, $f(0) := 0$
4 Compute new p-sums $p_i, \ldots, p_j$ for the sequence Δf
5 **For** $\ell = i, \ldots, j$:
6 $\hat{p}_\ell = p_\ell + \gamma_\ell, \quad \gamma_\ell \sim \mathrm{Lap}(\Gamma\varepsilon^{-1})$
7 Release new noisy p-sums $\hat{p}_i, \ldots, \hat{p}_j$

We adapt the counting mechanisms to continually release graph functions by following the approach by Song et al. [53]. Algorithm 6.3.2 outlines the generic algorithm. It is similar to Algorithm 6.3.1, with the difference that the stream of numbers to be summed is computed from a graph sequence $\mathcal{G}$. The algorithm is independent of the notion of adjacency of graph sequences, however the additive error is linear in the continuous global sensitivity of the difference sequence Δf.

In counting binary streams we considered adjacent inputs that differ at exactly one time step. In the graph setting however, the stream of numbers that we sum, i.e., the difference sequence Δf, can differ in multiple time steps between two adjacent graph sequences. We illustrate this with a simple example. Let f be the function that counts the number of edges in a graph and consider two node-adjacent incremental graph sequences $\mathcal{G}, \mathcal{G}'$. $\mathcal{G}$ contains an additional node v^*, that is not present in $\mathcal{G}'$. Whenever a neighbor is added to v^* in $\mathcal{G}$, the number of edges in $\mathcal{G}$ increases by more than the number of edges in $\mathcal{G}'$. Thus, every time a neighbor to v^* is inserted, the difference sequence of f will differ between $\mathcal{G}$ and $\mathcal{G}'$.

To generalize this, consider two adjacent graph sequences $\mathcal{G}, \mathcal{G}'$ that differ in an update at time t'. Let $\Delta f_{\mathcal{G}}$ and $\Delta f_{\mathcal{G}'}$ be the difference sequences used to compute the graph function f on $\mathcal{G}$ and $\mathcal{G}'$. As discussed above we may have $\Delta f_{\mathcal{G}}(t) \neq \Delta f_{\mathcal{G}'}(t)$ for all $t \geq t'$. Thus, more than x p-sums can be different, which complicates the proof of the privacy loss. However, we observe that the set P of p-sums with differing values can be partitioned into x sets $P_1, \ldots, P_x$, where the p-sums in each P_i cover disjoint time intervals. By using a bound on the continuous global sensitivity of the difference sequence Δf, this will lead to a privacy loss of $x \cdot \varepsilon$. We now prove that the results on privacy loss and error transfer from the counting mechanisms to the graph mechanisms.

Lemma 6.3.4 (Lemma 6.3.3 for graph sequences)**.** *Let f be a graph function whose difference sequence has continuous global sensitivity Γ. Let $0 < \delta < 1$ and $\varepsilon > 0$. Let $\mathcal{A}$ be a mechanism to estimate f as in Algorithm 6.3.2 that releases k noisy p-sums and satisfies the following conditions:*

1. *at any time step the value of a graph function f can be estimated as the sum of at most y noisy p-sums,*

2. *$\mathcal{A}$ adds independent noise from* $\mathrm{Lap}(\Gamma/\varepsilon)$ *to every p-sum,*
3. *the set P of p-sums computed by the algorithm can be partitioned into at most x subsets $P_1, \ldots, P_x$, such that in each partition P_x all p-sums cover disjoint time intervals. That is, for all $P_i \in \{P_1, \ldots, P_x\}$ and all $j, k \in P_i$, $j \neq k$, it holds that (1) $\mathrm{start}(j) \neq \mathrm{start}(k)$ and (2) $\mathrm{start}(j) < \mathrm{start}(k) \implies \mathrm{end}(j) < \mathrm{start}(k)$.*

Then, $\mathcal{A}$ is $(x \cdot \varepsilon)$-differentially private, and the error is $O(\Gamma \varepsilon^{-1} \sqrt{y} \log \frac{1}{\delta})$ with probability at least $1 - \delta$ at each time step.

Proof. As before the error follows from Corollary 6.3.1 and condition 1.

Let $\mathcal{G} = (G_1, \ldots, G_T)$, $\mathcal{G}' = (G'_1, \ldots, G'_T)$ be two adjacent graph sequences and let $f(t) = f(G_t)$, $f'(t) = f(G'_t)$. By definition of adjacent graph sequences (see Definitions 6.2.2 and 6.2.3) we have $f(0) = f'(0) = f((V_0, E_0))$. At every time step the algorithm computes $\Delta f_{\mathcal{G}}(t) = f(t) - f(t-1)$ and $\Delta f_{\mathcal{G}'}(t) = f'(t) - f'(t-1)$.

Note that the noisy p-sums computed by $\mathcal{A}$ are independent continuous random variables with joint distribution

$$p(z) = \prod_{i=1}^{k} p_i(z_i)$$

and

$$p'(z) = \prod_{i=1}^{k} p'_i(z_i)$$

when the input stream is $\mathcal{G}$ and $\mathcal{G}'$, respectively. As in the proofs of Lemmas 6.3.2 and 6.3.3, the p_i and p'_i are the marginal probability density functions. The random variables are continuous, since the noise added to the p-sums is drawn from a continuous distribution over $\mathbb{R}$.

Furthermore, let $c = (c_1, \ldots, c_k)^\top$ and $c' = (c'_1, \ldots, c'_k)^\top$ be the noiseless p-sums corresponding to the noisy p-sums output by $\mathcal{A}$ on inputs $\mathcal{G}$ and $\mathcal{G}'$, respectively. For each time step $t \in \{1, \ldots, T\}$ we define $\delta(t) = \Delta f_{\mathcal{G}'}(t) - \Delta f_{\mathcal{G}}(t)$. Note that $\sum_{t=1}^{T} |\delta(t)| \leq \Gamma$, by definition of the continuous global sensitivity Γ of the difference sequence. In this proof we use $\mathrm{start}(i)$ and $\mathrm{end}(i)$ to denote the beginning and end of the time interval corresponding to the p-sums with index i. For each $i \in \{1, \ldots, k\}$ we define $\delta_i = c'_i - c_i = \sum_{t=\mathrm{start}(i)}^{\mathrm{end}(i)} \delta(t)$. Let $I = \{i \mid \delta_i \neq 0\}$ be the indices of p-sums where the values for the two graph sequences are different. Lastly, let $s = (s_1, \ldots, s_k)^\top \in \mathrm{Range}(\mathcal{A})$.

$$\begin{aligned}
\frac{p(s)}{p'(s)} &= \prod_{i=1}^{k} \frac{p_i(s_i)}{p'_i(s_i)} \\
&= \prod_{i \in I} \frac{p_i(s_i)}{p'_i(s_i)} \\
&= \prod_{i \in I} \frac{\exp\left(-\frac{\varepsilon |c_i - s_i|}{\Gamma}\right)}{\exp\left(-\frac{\varepsilon |c_i + \delta_i - s_i|}{\Gamma}\right)} \\
&= \prod_{i \in I} \exp\left(\frac{\varepsilon}{\Gamma} \cdot (|c_i + \delta_i - s_i| - |c_i - s_i|)\right) \\
&\leq \prod_{i \in I} \exp\left(\frac{\varepsilon}{\Gamma} \cdot |\delta_i|\right) \qquad (16)
\end{aligned}$$

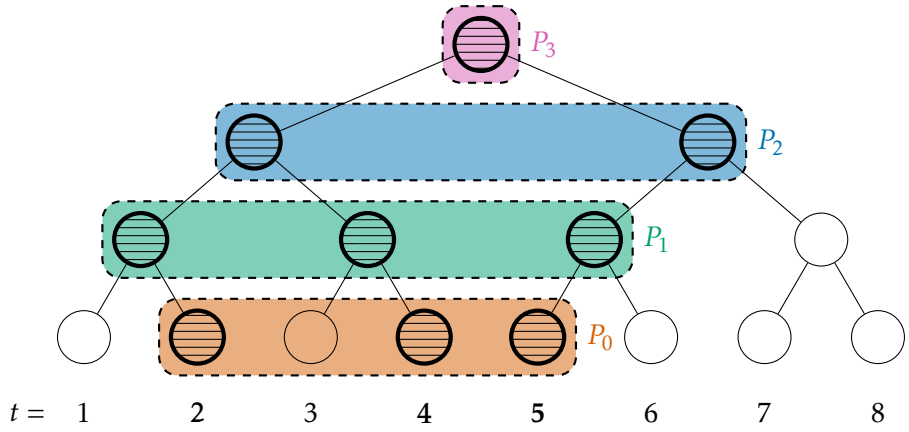

Figure 6.3.1: For a stream of length 8 the binary mechanism computes p-sums as shown. The leaves of the tree correspond to p-sums over the individual time steps; the root corresponds to a p-sum over the full stream. Now assume that in Algorithm 6.3.2 Δf differs at time steps 2, 4 and 5 (marked in bold) for adjacent graph sequences. The p-sums involving these time steps (shaded circles) can be partitioned into 4 sets P_0, P_1, P_2, P_3, indicated by dashed rectangles. P_i contains the p-sums of length 2^i.

We use condition 3 and partition I into sets of indices $I_1, \ldots, I_x$ such that for all $j \in \{1, \ldots, x\}$ the p-sums corresponding to indices in I_j cover disjoint time intervals. For each set I_j we then have

$$\sum_{i \in I_j} |\delta_i| \leq \sum_{i \in I_j} \sum_{t=\text{start}(i)}^{\text{end}(i)} |\delta(t)| \leq \sum_{t=1}^{T} |\delta(t)| \leq \Gamma. \tag{17}$$

Combining (16) and (17) yields

$$\begin{aligned}
\frac{p(s)}{p'(s)} &\leq \prod_{i \in I} \exp\left(\frac{\varepsilon}{\Gamma} \cdot |\delta_i|\right) \\
&= \prod_{j=1}^{x} \prod_{i \in I_j} \exp\left(\frac{\varepsilon}{\Gamma} \cdot |\delta_i|\right) \\
&= \prod_{j=1}^{x} \exp\left(\frac{\varepsilon}{\Gamma} \cdot \sum_{j \in I_j} |\delta_i|\right) \\
&\leq \prod_{j=1}^{x} \exp\left(\frac{\varepsilon}{\Gamma} \cdot \Gamma\right) = \exp(x \cdot \varepsilon).
\end{aligned}$$

Thus, $\mathcal{A}$ is $x \cdot \varepsilon$-differentially private, since

$$\Pr[\mathcal{A}(\sigma) \in S] = \int_S p(s)ds \leq \int_S e^{x \cdot \varepsilon} p'(s)ds = e^{x \cdot \varepsilon} \Pr[A(\sigma') \in S]$$

for any $S \subseteq \text{Range}(\mathcal{A})$. This concludes the proof of Lemma 6.3.4. □

We can compute the p-sums in Algorithm 6.3.2 as in the binary mechanism [55] to release ε-differentially private estimates of graph functions.

Corollary 6.3.5 (Binary mechanism). *Let f be a graph function whose difference sequence has continuous global sensitivity Γ. Let $0 < \delta < 1$ and $\varepsilon > 0$. For each $T \in \mathbb{N}$ there exists an ε-differentially private algorithm to estimate f on a graph sequence which has error $O(\Gamma\varepsilon^{-1} \cdot \log^{3/2} T \cdot \log \delta^{-1})$ with probability at least $1 - \delta$.*

Proof. We show that the binary mechanism by Chan et al. [55] can be employed to construct an algorithm as in Algorithm 6.3.2.

The binary mechanism divides the T time steps of the input sequence into $\lfloor T/2^i \rfloor$ intervals of length 2^i for each $i = 0, \ldots, \log T$. For each interval the binary mechanism computes a p-sum. The p-sum of length 2^i used to compute the output at time step t is indexed as

$$q_i(t) = \sum_{j=0}^{i-1} \left\lfloor \frac{T}{2^i} \right\rfloor + \left\lfloor \frac{t}{2^i} \right\rfloor.$$

Let $a = (a_1, \ldots, a_k)$ be the noisy p-sums of the difference sequence of a graph function f for any graph sequence of length T. The value $f(t)$ at time t is computed as the sum

$$\sum_{i=0}^{\log T} \mathrm{bin}_i(t) \cdot a_{q_i(t)},$$

where $\mathrm{bin}_i(t)$ is the i-th bit in the binary representation of t, with $i = 0$ for the least significant bit.

We have $y = \log T$, since the sum at any time t is computed from at most $\log T$ noisy p-sums corresponding to the at most $\log T$ bits of the binary representation of t. Every time step t participates in p-sums of length 2^i for all $i = 0, \ldots, \log T$. Thus, $x = \log T + 1$. The set P of all p-sums can be partitioned into x subsets of disjoint p-sums as illustrated in Figure 6.3.1: for each $i = 0, \ldots, \log T$ the set P_i contains the p-sums of length exactly 2^i. Thus, using the binary mechanism to sum the noisy p-sums we obtain an algorithm that satisfies the conditions of Lemma 6.3.4.

To obtain an ε-differentially private algorithm we set the privacy parameter in Lemma 6.3.4 to ε/x. The error is then $O(\Gamma\varepsilon^{-1} \cdot x \cdot \sqrt{y} \cdot \log \delta^{-1})$ with probability at least $1 - \delta$. The claim follows from $x, y = O(\log T)$. □

6.3.3 Bounds on Continuous Global Sensitivity

Song et al. [53] give bounds on the continuous global sensitivity of the difference sequence for several graph functions in the incremental setting in terms of the maximum degree D. Table 6.3.1 summarizes the results on the continuous global sensitivity of difference sequences for a variety of problems in the partially dynamic and fully dynamic setting, both for edge- and node-adjacency. Note that bounds on the continuous global sensitivity of the difference sequence for incremental graph sequences hold equally for decremental graph sequences as for every incremental graph sequence there exists an equivalent decremental graph sequence which deletes nodes and edges in the reverse order.

In the partially dynamic setting, the continuous global sensitivity based approach works well for graph functions that can be expressed as the sum of local functions on the neighborhood of nodes. For non-local problems the

Table 6.3.1: Global sensitivity of difference sequences

Graph Function f	Continuous Global Sensitivity of Δf			
	Partially Dynamic		Fully Dynamic	
	node-adjacency	edge-adjacency	node-adj.	edge-adj.
edge count[a]	D	1	$\geq T$	2
high-degree nodes[a]	$2D+1$	4	$\geq T$	$\geq T$
degree histogram[a]	$4D^2+2D+1$	$8D$	$\geq 2T$	$\geq 2T$
triangle count[a]	$\binom{D}{2}$	D	$\geq T$	$\geq T$
k-star count[a]	$D\binom{D-1}{k-1}+\binom{D}{k}$	$2\cdot\left(\binom{D}{k}-\binom{D-1}{k}\right)$	$\geq T$	$\geq T$
minimum spanning tree	$2DW$	$2W-2$	$\geq T$	$\geq T$
minimum cut	$\geq T$	$\geq T$	$\geq T$	$\geq T$
maximum matching	$\geq T$	$\geq T$	$\geq T$	$\geq T$

[a] Bounds for partially dynamic node-adjacency from Song et al. [53]

approach is less successful. For the weight of a minimum spanning tree the continuous global sensitivity of the difference sequence is independent of the length of the graph sequence. However, for minimum cut and maximum matching this is not the case. In the fully dynamic setting the approach seems not to be useful. Here, we can show that even for estimating the number of edges the continuous global sensitivity of the difference sequence scales linearly with T under node-adjacency. When considering edge-adjacency we only have low sensitivity for the edge count.

Using Corollary 6.3.5 we obtain ε-differentially private mechanisms with additive error that scales with $\log^{3/2} T$, compared to the factor $\sqrt{T}$ in [53]. Note that we recover their algorithm when using the Simple Mechanism II by Chan et al. [55] to sum the difference sequence.

For edge-adjacency we obtain the following bounds on the sensitivity of the difference sequence.

Lemma 6.3.6. *When considering edge-adjacency in the incremental setting, the continuous global sensitivity of the difference sequence of the*

1. *number of high-degree nodes is 4;*
2. *degree histogram is $8D$;*
3. *edge count is 1;*
4. *triangle count is D;*
5. *k-star count is $2\cdot\left(\binom{D}{k}-\binom{D-1}{k}\right)$.*

Proof. In the following, $\mathcal{G}$ and $\mathcal{G}'$ are edge-adjacent incremental graph sequences that differ in the insertion of an edge $e^* = \{u^*, v^*\}$.

1. In $\mathcal{G}$, the endpoints of e^* may cross the degree-threshold τ as soon as e^* is inserted, increasing the number of high-degree nodes by 2. In $\mathcal{G}'$, the same nodes may become high-degree nodes at a later update, increasing the number of high-degree nodes by 1 each. Thus, the difference sequences differ by at most 4.

2. The degrees of u^* and v^* are one less in $\mathcal{G}'$ compared to $\mathcal{G}$, once e^* is inserted. There are up to D insertions of edges incident to u^*, v^*. For each of these updates, the increase in the degree of u^* affects 2 histogram-bins in $\mathcal{G}$ and $\mathcal{G}'$, each. The same holds for the increase in the degree of v^*. Thus, the continuous global sensitivity of the difference sequence of the degree histogram is $8D$.

3. The edge-update sets ∂E_t^+ and $\partial E_t^{+\prime}$ have exactly the same size at all times, except when e^* is inserted. Thus, the global sensitivity of the difference sequence the edge count is the number of edges that are inserted into $\mathcal{G}$, but not into $\mathcal{G}'$, which is 1.

4. In $\mathcal{G}'$, the u^* and v^* are not part of shared triangles. In $\mathcal{G}$ however, they may become part of up to D triangles. Thus, the continuous global sensitivity of the difference sequence of the triangle count is D.

5. In $\mathcal{G}$, after inserting e^*, u^* and v^* may be the center of up to $\binom{D}{k}$-many k-stars. Then, they were the center of $\binom{D-1}{k}$-many k-stars before the insertion of e^*. Any k-star in $\mathcal{G}'$ is also present in $\mathcal{G}$. Thus, the continuous global sensitivity of the difference sequence of the k-star count is the number of k-stars that the insertion of e^* can contribute, which is $2\left(\binom{D}{k} - \binom{D-1}{k}\right)$. □

The following lemmata provide evidence that the difference sequence based approach is not useful for fully dynamic graphs.

Lemma 6.3.7. *Let $f_\tau((V,E)) = |\{v \in V \mid \deg(v) \geq \tau\}|$. The difference sequence of f_τ has continuous global sensitivity at least T when considering edge-adjacent or node-adjacent fully dynamic graph sequences.*

Proof. For the edge-adjacency case consider two graph sequences $\mathcal{G}$, $\mathcal{G}'$, where the initial graph is a τ-star with two edges e_1, e_2 removed. At time $t = 1$ insert e_1 into $\mathcal{G}$, but not into $\mathcal{G}'$. At odd times, insert e_2 into both sequences; at even times delete e_2 from both sequences. The sequence $\mathcal{G}'$ never has a node of degree at least τ, whereas $\mathcal{G}$ has one such node at odd, but not at even time steps. Thus, $\sum_{t=1}^{T} |\Delta f_{\tau,\mathcal{G}}(t) - \Delta f_{\tau,\mathcal{G}'}(t)| = T$.

For the node-adjacency case we again consider two graph sequences $\tilde{\mathcal{G}}$, $\tilde{\mathcal{G}}'$, where the initial graph is a τ-star with two nodes a, b and their incident edges removed. At time $t = 1$ insert a and its incident edge into $\tilde{\mathcal{G}}$, but not into $\tilde{\mathcal{G}}'$. At odd times, insert b and its incident edge into both sequence; at even times delete b from both sequences. As in the edge-adjacency case it follows that $\sum_{t=1}^{T} |\Delta f_{\tau,\tilde{\mathcal{G}}}(t) - \Delta f_{\tau,\tilde{\mathcal{G}}'}(t)| = T$. □

Lemma 6.3.8. *The difference sequence of the degree histogram has continuous global sensitivity at least $2T$ when considering edge-adjacent or node-adjacent fully dynamic graph sequences.*

Proof. We refer to the adjacent graph sequences introduced in the proof of Lemma 6.3.7. In one of the graph sequences, the bins in the degree histogram corresponding to degrees $\tau - 1$ and τ are alternatingly zero and one. In the other graph sequence the same is true for the bins corresponding to degrees $\tau - 2$ and τ. Let $h_{\mathcal{G}}(t)$, $h_{\mathcal{G}'}(t)$ be the degree histograms of the former and latter sequence at time t, respectively. The corresponding difference sequences

are $\Delta h_{\mathcal{G}}(t) = (d_{t,0}, \dots, d_{t,n-1})$ and $\Delta h_{\mathcal{G}'}(t) = (d'_{t,0}, \dots, d'_{t,n-1})$, where $d_{t,i}$ and $d'_{t,i}$ is the change in the number of nodes of degree i from time $t-1$ to t. For odd t, we have $d_{t,\tau} = 1$, $d_{t,\tau-1} = -1$, $d'_{t,\tau-1} = 1$, $d'_{t,\tau-2} = -1$; for even t, we have $d_{t,\tau} = -1$, $d_{t,\tau-1} = 1$, $d'_{t,\tau-1} = -1$, $d'_{t,\tau-2} = 1$. Note that $d_{t,1} = d'_{t,1}$ for $t \geq 2$ and $d_{1,1} = 2 = d_{1,1'} + 1$. All $d_{t,i}$ not explicitly given are 0. It follows that $\sum_{t=1}^{T} \|\Delta h_{\mathcal{G}}(t) - \Delta h_{\mathcal{G}'}(t)\|_1 \geq 2T$, which concludes the proof. □

Lemma 6.3.9. *Let H be a connected graph with at least two edges (nodes) and $f_H(G)$ be the function counting occurrences of H in G. When considering edge-adjacent (node-adjacent) fully dynamic graph sequences of length T, $\mathrm{GS}(\Delta f_H) \geq T$.*

Proof. To show the edge-adjacency case consider adjacent graph sequences $\mathcal{G}, \mathcal{G}'$ of length T, where the initial graph is a copy of H with two edges, say $\{a,b\}, \{c,d\}$, removed. In the first time step ($t = 1$), insert $\{a,b\}$ into $\mathcal{G}$, but not into $\mathcal{G}'$ and insert $\{c,d\}$ into both sequences. Thus, we have $f_H(G_1) = 1$ and $f_H(G'_1) = 0$. In time steps where t is even, delete $\{c,d\}$ from both sequences; if t is odd, insert $\{c,d\}$ into both sequences. At all time steps t we have $f_H(G'_t) = 0$ and $|\Delta f_{H,\mathcal{G}}(t)| = 1$, which implies $\sum_{t=1}^{T} |\Delta f_{H,\mathcal{G}}(t) - \Delta f_{H,\mathcal{G}'}(t)| = T$.

For the node-adjacency case consider again adjacent graph sequences $\tilde{\mathcal{G}}, \tilde{\mathcal{G}}'$ of length T, where the initial graph is a copy of H with two nodes, say a, b, removed. At time step $t = 1$ insert a along with the incident edges into $\tilde{\mathcal{G}}$, but not $\tilde{\mathcal{G}}'$. At every odd time step, insert b along with the incident edges into both sequences; at every even time step delete b from both sequences. Again, $\tilde{\mathcal{G}}'$ never contains a copy of H, whereas $\tilde{\mathcal{G}}$ contains a copy of H at every odd time step, but no on the even time steps. Thus, $\sum_{t=1}^{T} |\Delta f_{H,\tilde{\mathcal{G}}}(t) - \Delta f_{H,\tilde{\mathcal{G}}'}(t)| = T$. □

Corollary 6.3.10. *The difference sequence of triangle count and k-star count has continuous global sensitivity $\geq T$ when considering edge-adjacent or node-adjacent graph sequences. The difference sequence of the edge count has continuous global sensitivity $\geq T$ when considering node-adjacent graph sequences.*

Proof. The claim follows from Lemma 6.3.9, by observing that triangles and k-stars have at least two edges and nodes, and noting that for the edge count $H = (\{a,b\}, \{\{a,b\}\})$. □

Lemma 6.3.11. *The difference sequence of the weight of a minimum spanning tree has continuous global sensitivity at least T when considering edge-adjacent or node-adjacent fully dynamic graph sequences.*

Proof. Let $A = (V_A, E_A)$ and $B = (V_B, E_B)$ be connected graphs with $|V_A|, |V_B| \geq 3$ and $w(e) = 1$ for all $e \in E_A \cup E_B$. We denote nodes in V_A by a_i, nodes in V_B by b_i with index $i = 0, 1, \dots$. Let $G_0 = (V_A \cup V_B, E_A \cup E_B \cup \{\{a_0, b_0\}\})$ with $w(\{a_0, b_0\}) = W \geq 3$. Let $\mathcal{G}, \mathcal{G}'$ be graph sequences of length T with initial graph G_0.

At time $t = 1$ we insert the edge $e^* = \{a_1, b_1\}$ with weight $w(e^*) = 1$ into $\mathcal{G}$. Then, if t is even, we insert the edges $e' = \{a_2, b_2\}$ with weight $w(e') = W - 1$ into $\mathcal{G}$ and $\mathcal{G}'$. If t is odd, we delete e' from both sequences.

The sequences $\mathcal{G}$ and $\mathcal{G}'$ are edge-adjacent. The weight of the minimum spanning tree in G_0 is $w_0 = |V_A| - 1 + |V_B| - 1 + W = |V_A| + |V_B| + W - 2$. For all $t \geq 1$, the weight of the minimum spanning tree in $\mathcal{G}$ is $w_{\mathcal{G}} = 1 + |V_A| - 1 + |V_B| - 1 = |V_A| + |V_B| - 1$. If t is even and $t \geq 2$, then the weight of the minimum spanning tree in $\mathcal{G}'$ is $w_0 - 1$; if t is odd it is w_0. Thus, we have

$$|\Delta w_{\mathrm{MST}_{\mathcal{G}}}(t) - \Delta w_{\mathrm{MST}_{\mathcal{G}'}}(t)| \geq 1$$

for all $t = 1, \dots, T$, where $w_{\mathrm{MST}_{\mathcal{G}}}$ and $w_{\mathrm{MST}_{\mathcal{G}'}}$ denote the weight of the minimum spanning tree in $\mathcal{G}$ and $\mathcal{G}'$, respectively. Summing over all t proves the claim for edge-adjacent sequences.

To show that the bound holds for node-adjacent sequences, we replace the edge-insertions by node-insertions as follows: instead of e^* we insert a node v^* along with edges $e_1^* = \{a_1, v^*\}$ and $e_2^* = \{v^*, b_1\}$, where $w(e_1^*) = w(e_2^*) = 1$; instead of e' we insert a node v' along with edges $e_1' = \{a_2, v'\}$ and $e_2' = \{v', b_1\}$, where $w(e_1') = 1$ and $w(e_2') = W - 1$; where we delete e' in the edge-adjacency case we delete v' and the incident edges. We use the same initial graph G_0 and obtain node-adjacent graph sequences where the weights of the minimum spanning tree are increased by 1 compared to the weights in the corresponding node-adjacent graph sequences. The claim for node-adjacent graph sequences follows as for edge-adjacent graph sequences. □

For maximum cardinality matching and minimum cut the continuous global sensitivity of the difference sequence scales linearly with the length of the input even in the partially dynamic setting.

Lemma 6.3.12. *Let $M(G)$ be the size of a maximum cardinality matching of G. The difference sequence of M has continuous global sensitivity at least T when considering edge-adjacent or node-adjacent partially dynamic graph sequences.*

Proof. First, we construct edge-adjacent graph sequences $\mathcal{G}, \mathcal{G}'$, where $|\Delta M_{\mathcal{G}}(t) - \Delta M_{\mathcal{G}'}(t)| = 1$ for all t. Let $V = \{0, \dots, T\}$. At time $t = 1$ insert the edge $\{0, 1\}$ into $\mathcal{G}$ and insert no edge into $\mathcal{G}'$. $\Delta M_{\mathcal{G}}(1) = 1$, since G_1 has a maximum cardinality matching of size 1. $\Delta M_{\mathcal{G}'}(1) = 0$, since G_1' has no edges. At every time $t > 1$ insert the edge $\{t-1, t\}$ into both $\mathcal{G}$ and $\mathcal{G}'$. If t is even, then $\Delta M_{\mathcal{G}}(t) = 0$ and $\Delta M_{\mathcal{G}'}(t) = 1$. If t is odd, then $\Delta M_{\mathcal{G}}(t) = 1$ and $\Delta M_{\mathcal{G}'}(t) = 0$. Thus, we have $\sum_{t=1}^{T} |\Delta M_{\mathcal{G}}(t) - \Delta M_{\mathcal{G}'}(t)| = T$.

For node-adjacency, both graph sequences start with the node 1. In $\mathcal{G}$, we insert node 0 along with the edge $\{0, 1\}$ at time 1. Then, at every time step $t > 1$ we insert node t and the edge $\{t, t-1\}$ into both graph sequences. The difference sequence is the same as in the edge-adjacency case. Thus, the continuous global sensitivity of the difference sequence is also at least T. □

Lemma 6.3.13. *Let $w_{\mathrm{CUT}}(G)$ be the weight of a minimum cut of G. The difference sequence of w_{CUT} has continuous global sensitivity at least T when considering edge-adjacent or node-adjacent partially dynamic graph sequences.*

Proof. We show that for any $T \in \mathbb{N}$ there exist edge-adjacent graph sequences $\mathcal{G}, \mathcal{G}'$, such that $\sum_{t=1}^{T} |\Delta w_{\mathrm{CUT}_{\mathcal{G}}}(t) - \Delta w_{\mathrm{CUT}_{\mathcal{G}'}}(t)| \geq T$.

Let $A = (V_A, E_A)$, $B = (V_B, E_B)$, $C = (V_C, E_C)$ be graphs on at least $T + 2$ nodes with minimum cut of weight at least $W \cdot (T + 1)$. We denote nodes in V_A by a_i, nodes in V_B by b_i and nodes in V_C by c_i, with index $i = 0, 1, \dots$. Let $G_0 = (V_A \cup V_B \cup V_C, E_A \cup E_B \cup E_C \cup \{e_{ab}, e_{bc}\}$, where $e_{ab} = \{a_{T+1}, b_{T+1}\}$ and $e_{bc} = \{b_{T+1}, c_{T+1}\}$ with $w(e_{ab}) = 1$ and $w(e_{bc}) = 2$. Let $\mathcal{G}, \mathcal{G}'$ be graph sequences with initial graph G_0.

At time $t = 1$ we insert the edge $e^* = \{a_0, b_0\}$ with $w(e^*) = W$ into $\mathcal{G}$. Then, if t is even, we insert the edge $e_t = \{a_t, b_t\}$ with $w(e_t) = W$ into both $\mathcal{G}$ and $\mathcal{G}'$; If t is odd, we insert the edge $e_t = \{b_t, c_t\}$ with $w(e_t) = W$ into both $\mathcal{G}$ and $\mathcal{G}'$.

These graph sequences are edge-adjacent since they differ only in the insertion of e^*. At any time t the minimum cut in G_t is $(V_A \cup V_B, V_C)$. In $\mathcal{G}'$, the minimum cut is $(V_A, V_B \cup V_C)$ for odd t and $(V_A \cup V_B, V_C)$ for even t.

In $\mathcal{G}$, the value of the minimum cut increases by 1 at time $t = 1$ due to insertion of e^* and by W for odd $t \geq 3$, since an edge of weight W across the minimum cut is inserted. That is,

$$\Delta w_{\mathrm{CUT}_{\mathcal{G}}}(t) = \begin{cases} 1 & \text{if } t = 1, \\ 0 & \text{if } t \text{ is even}, \\ W & \text{if } t \text{ is odd and } t \geq 3. \end{cases} \tag{18}$$

In $\mathcal{G}'$, the value of the minimum cut does not change at time $t = 1$, since no edge is inserted. At even times, the cut switches to the edges between V_B and V_C and the weight of the minimum cut increases by 1. At odd times, the cut switches to the edges between V_A and V_B, and the weight of the minimum cut increases by $W - 1$. The difference sequence for $\mathcal{G}'$ is thus,

$$\Delta w_{\mathrm{CUT}_{\mathcal{G}'}}(t) = \begin{cases} 0 & \text{if } t = 1, \\ 1 & \text{if } t \text{ is even}, \\ (W-1) & \text{if } t \text{ is odd and } t \geq 3. \end{cases} \tag{19}$$

From (18) and (19) we get $|\Delta w_{\mathrm{CUT}_{\mathcal{G}}}(t) - \Delta w_{\mathrm{CUT}_{\mathcal{G}'}}(t)| = 1$ for all t. Summing over all t we get the proposed bound on the continuous global sensitivity.

We obtain the same bound for node-adjacent partially dynamic graph sequences by modifying the above construction as follows: we start with the same initial graph G_0 as above. We replace an insertion of the edge $\{a_i, b_i\}$ by an insertion of node a_i' along with edges $\{a_i', b_i\}$ and edges $\{a_i', a_j\}$ for all $a_j \in V_A$; all these edges have weight W. Similarly, we replace an insertion of the edge $\{b_i, c_i\}$ by an insertion of node c_i' along with edges $\{c_i', b_i\}$ and edges $\{c_i', c_j\}$ for all $c_j \in V_C$; all these edges have weight W. By counting the nodes a_i' and c_i' to V_A and V_C, respectively, we obtain the same sequence of minimum cuts as in the edge-adjacency case. Thus, we get the same lower bound on global sensitivity for the node-adjacency case. □

The difference sequence approach can be employed to privately estimate the weight of a minimum spanning tree in partially dynamic graph sequences. If the edge weight is bounded by W, then the continuous global sensitivity of the difference sequence is $O(W)$ and $O(DW)$ under edge-adjacency and node-adjacency, respectively. In Theorems 6.3.14 and 6.3.15 we prove these bounds and use Corollary 6.3.5 to derive edge- and node-differentially private algorithms.

Theorem 6.3.14. *There exists an ε-edge-differentially private algorithm that outputs the weight of a minimum spanning tree on an incremental graph sequence $\mathcal{G}$ with edge-weights from the set $\{1, \ldots, W\}$. At every time step, the algorithm has error $O(W\varepsilon^{-1} \cdot \log^{3/2} T \cdot \log \delta^{-1})$ with probability at least $1 - \delta$, where T is the length of the graph sequence.*

Proof. We will show that the difference sequence for the weight of a minimum spanning tree has global edge-sensitivity $2W - 2$ in the incremental setting. The claim then follows by Corollary 6.3.5.

Let $\mathcal{G} = (G_1, \ldots, G_T)$, $\mathcal{G}' = (G_1', \ldots, G_T')$ be two adjacent graph sequences with initial graphs G_0 and G_0'. We are considering edge-differential privacy and thus edge-adjacency (Definition 6.2.2). This means that $\mathcal{G}$ and $\mathcal{G}'$ differ in the insertion of an edge e^*. Without loss of generality, we assume that e^* is inserted into $\mathcal{G}$. For estimating $\mathrm{GS}(\Delta w_{\mathrm{MST}})$ we can further assume that e^* is

inserted into G_0 to obtain G_1, since $\Delta w_{\mathrm{MST}\mathcal{G}}(t) = \Delta w_{\mathrm{MST}\mathcal{G}'}(t)$ for all times t before the insertion of e^*. For all $t = 0,\dots,T$ the edge set of G_t is a superset of the edge set of G'_t. This implies $w_{\mathrm{MST}}(G_t) \leq w_{\mathrm{MST}}(G'_t)$ for all t.

Note that $w_{\mathrm{MST}}(G_0) = w_{\mathrm{MST}}(G'_0) = w_{\mathrm{MST}}(G'_1)$, which implies $\Delta w_{\mathrm{MST}}\mathcal{G}'(1) = 0$.

As e^* might either not replace any edge in a minimum spanning tree or might replace an edge of weight W, it holds that $0 \leq \Delta w_{\mathrm{MST}}\mathcal{G}'(1) \leq W - 1$. Thus, it follows that $0 \leq \Delta w_{\mathrm{MST}\mathcal{G}}(1) - \Delta w_{\mathrm{MST}\mathcal{G}'}(1) \leq W - 1$.

It remains to bound $\sum_{t=2}^{T} |\Delta w_{\mathrm{MST}\mathcal{G}}(t) - \Delta w_{\mathrm{MST}\mathcal{G}'}(t)|$. Any edge inserted into $\mathcal{G}$ is also inserted into $\mathcal{G}'$. If inserting an edge into $\mathcal{G}$ at time t reduces the weight of the minimum spanning tree of $\mathcal{G}$ at time t, then the weight of the minimum spanning tree in $\mathcal{G}'$ at time t is reduced by at least the same amount. Thus there are two cases: (1) the weight of the minimum spanning tree in $\mathcal{G}$ and $\mathcal{G}'$ change by the same amount; (2) the weight of the minimum spanning tree in $\mathcal{G}'$ is reduced more than the weight of a minimum spanning tree in $\mathcal{G}$. It follows that for all $t > 0$, $\Delta w_{\mathrm{MST}\mathcal{G}}(t) - \Delta w_{\mathrm{MST}\mathcal{G}'}(t) \geq 0$. Thus, $\sum_{t=2}^{T} |\Delta w_{\mathrm{MST}\mathcal{G}}(t) - \Delta w_{\mathrm{MST}\mathcal{G}'}(t)| = \sum_{t=2}^{T} \Delta w_{\mathrm{MST}\mathcal{G}}(t) - \Delta w_{\mathrm{MST}\mathcal{G}'}(t) = \sum_{t=2}^{T} \Delta w_{\mathrm{MST}\mathcal{G}}(t) - \sum_{t=2}^{T} \Delta w_{\mathrm{MST}\mathcal{G}'}(t)$. But then it follows that

$$
\begin{aligned}
&\sum_{t=2}^{T} \Delta w_{\mathrm{MST}\mathcal{G}}(t) - \sum_{t=2}^{T} \Delta w_{\mathrm{MST}\mathcal{G}'}(t) \\
=& w_{\mathrm{MST}}(G_T) - w_{\mathrm{MST}}(G_1) - w_{\mathrm{MST}}(G'_T) + w_{\mathrm{MST}}(G'_1) \\
\leq& W - 1,
\end{aligned}
$$

where the last inequality follows from $w_{\mathrm{MST}}(G'_1) - w_{\mathrm{MST}}(G_1) \leq W - 1$ and $w_{\mathrm{MST}}G_T \leq w_{\mathrm{MST}}G'_T$.

Combining the results for $t = 1$ and $t > 1$, we get

$$
\sum_{t=1}^{T} |\Delta w_{\mathrm{MST}\mathcal{G}}(t) - \Delta w_{\mathrm{MST}\mathcal{G}'}(t)| \leq 2W - 2,
$$

which implies $\mathrm{GS}(\Delta w_{\mathrm{MST}}) \leq 2W - 2$.

We now show that this bound on the sensitivity is tight. Let $A = (V_A = \{a_1,\dots\}, E_A)$, $B = (V_B = \{b_1,\dots\}, E_B)$ be arbitrary connected graphs with $|A|, |B| > W + 1$ and $w(e) = 1$ for all $e \in E_A \cup E_B$. Let $G = (V_A \cup V_B, E_A \cup E_B \cup \{a_1, b_1\})$ and $w(\{a_1, b_1\}) = W$. Note that the edge $\{a_1, b_1\}$ is guaranteed to be in any minimum spanning tree.

We construct adjacent graph sequences $\mathcal{G}_a = (G_{a1}, G_{a2}, G_{a3})$, $\mathcal{G}_b = (G_{b1}, G_{b2}, G_{b3})$ with $G_{a1} = G_{b1} = G$, such that $\sum_{t=1}^{|\mathcal{G}_a|} |\Delta w_{\mathrm{MST}\mathcal{G}_a}(t) - \Delta w_{\mathrm{MST}\mathcal{G}_b}(t)| = 2W - 2$.

Starting with $G_{a1} = G$ we insert the edge $\{a_2, b_2\}$ with $w(\{a_2, b_2\}) = 1$ into $\mathcal{G}_a$ to obtain G_{a2}. In $\mathcal{G}_b$ we insert no edges, i.e., $G_{b2} = G_{b1} = G$. Inserting the unit-weight edge into $\mathcal{G}_a$ reduces the weight of a minimum spanning tree by $W - 1$. In $\mathcal{G}_b$, the weight of a minimum spanning tree does not change. Thus, we have $|\Delta w_{\mathrm{MST}\mathcal{G}_a}(2) - \Delta w_{\mathrm{MST}\mathcal{G}_b}(2)| = W - 1$.

In the subsequent time step we insert the edge $\{a_3, b_3\}$ with $w(\{a_3, b_3\}) = 1$ into $\mathcal{G}_a$ and $\mathcal{G}_b$, which yields the graphs G_{a3} and G_{b3}, respectively. This does not change the weight of the minimum spanning tree in $\mathcal{G}_a$, but reduces the weight of the minimum spanning tree in $\mathcal{G}_b$ by $W - 1$, since $\{a_3, b_3\}$ replaces $\{a_1, b_1\}$ in the minimum spanning tree. Thus, we have $|\Delta w_{\mathrm{MST}\mathcal{G}_a}(3) - \Delta w_{\mathrm{MST}\mathcal{G}_b}(3)| = W - 1$.

This concludes the proof. □

Theorem 6.3.15. *There exists an ε-node-differentially private algorithm that outputs the weight of a minimum spanning tree on an incremental graph sequence $\mathcal{G}$ with edge-weights from the set $\{1,\dots,W\}$. At every time step, the algorithm has error $O(DW\varepsilon^{-1}\cdot\log^{3/2}T\cdot\log\delta^{-1})$ with probability at least $1-\delta$, where T is the length of the graph sequence and D is the maximum degree.*

Proof. We will show that the difference sequence for the weight of a minimum spanning tree has continuous global sensitivity $2DW$ under node-adjacency in the incremental setting. The claim then follows by Corollary 6.3.5.

Let $\mathcal{G}=(G_1,\dots,G_T)$, $\mathcal{G}'=(G'_1,\dots,G'_T)$ be two adjacent graph sequences with initial graphs G_0 and G'_0. We are considering node-differential privacy and thus node-adjacency (Definition 6.2.3). This means that $\mathcal{G}$ and $\mathcal{G}'$ differ in the insertion of a node v^*. Without loss of generality, we assume that v^* is inserted into $\mathcal{G}$. For estimating $\mathrm{GS}(\Delta w_{\mathrm{MST}})$ we can further assume that v^* is inserted into G_0 to obtain G_1, since $\Delta w_{\mathrm{MST}\mathcal{G}}(t)=\Delta w_{\mathrm{MST}\mathcal{G}'}(t)$ for all times t before the insertion of v^*.

The node v^* has degree at most D, so we have $|\Delta w_{\mathrm{MST}\mathcal{G}}(1)-\Delta w_{\mathrm{MST}\mathcal{G}'}(1)|\leq DW$.

We now bound $\sum_{t=2}^{T}|\Delta w_{\mathrm{MST}\mathcal{G}}(t)-\Delta w_{\mathrm{MST}\mathcal{G}'}(t)|=:\Gamma_2$. Let E^* be the set of edges incident to v^*. Note that $|E^*|\leq D$. For each edge $e\in E^*$ there are three cases: (1) e is not in the minimum spanning tree when it is inserted; (2) e replaces a heavier edge; (3) e increases the weight of the minimum spanning tree.

In case (1), there always exists a minimum spanning tree that does not contain e, since no edges are ever deleted. Thus, e does not contribute to the continuous global sensitivity of the difference sequence. To analyze case (2), let e_H be the edge replaced by e. This edge may be replaced in $\mathcal{G}'$ at times different from the time at which e was inserted. However, this contributes at most $w(e_H)\leq W$ to Γ_2. In case (3), the weight that e contributes to the minimum spanning tree in $\mathcal{G}$ can be reduced by at most $W-1$. However, the edge used to replace e may increase the weight of a minimum spanning tree in $\mathcal{G}'$ by 1. Thus, e contributes up to W to Γ_2.

With $|E^*|\leq D$, we thus have $\Gamma_2\leq DW$. This implies

$$\begin{aligned}\mathrm{GS}(\Delta w_{\mathrm{MST}}) &= \sum_{t=1}^{T}|\Delta w_{\mathrm{MST}\mathcal{G}}(t)-\Delta w_{\mathrm{MST}\mathcal{G}'}(t)|\\ &= |\Delta w_{\mathrm{MST}\mathcal{G}}(1)-\Delta w_{\mathrm{MST}\mathcal{G}'}(1)|+\Gamma_2\\ &\leq 2DW.\end{aligned}$$

□

6.4 Upper Bound for Monotone Functions

In Section 6.3.3, we show that privately releasing the difference sequence of a graph sequence does not lead to good error guarantees for partially dynamic problems like minimum cut. Intuitively, the reason is that even for neighboring graph sequences $\mathcal{G},\mathcal{G}'$, the differences of the difference sequence can be non-zero for all graphs G_i, G'_i. In other words, the difference of objective values for the graphs G_i and G'_i can constantly fluctuate during continual updates. However, the difference of objective values, regardless of fluctuations, is *always small*. We show that, by allowing an arbitrarily small *multiplicative* error, we can leverage this fact for a broad class of partially dynamic problems. In particular, we prove that there exist ε-differentially private algorithms for all dynamic problems that are non-decreasing (or non-increasing) on all

valid input sequences. This includes, e.g., minimum cut, maximum matching and densest subgraph on partially dynamic inputs. See Algorithm 6.4.1 for the details and Table 6.1.2 on page 115 for explicit upper bounds for applications. We state the result for monotonically increasing functions, but it is straightforward to adapt the algorithm to monotonically decreasing functions.

Algorithm 6.4.1: Multiplicative error algorithm for monotone functions

Function `Initialize`$(D, \rho, \epsilon, r, \beta)$**:**
  $k_0 \leftarrow 0$
  `InitializeSvt` $(D, \rho, \epsilon, \log_{1+\beta}(r))$ `// see Algorithm 6.2.1`
Function `Process`(f_i)**:**
  $k_i \leftarrow k_{i-1}$
  while `ProcessSvtQuery`$(f_i, (1+\beta)^{k_i}) = \top$ **do** `// see Algorithm 6.2.1`
    $k_i \leftarrow k_i + 1$
  return $(1+\beta)^{k_i}$

Theorem 6.4.1. *Let $r > 0$ and let f be any monotonically increasing function on dynamic inputs (e.g., graphs) with range $[1, r]$ and static global sensitivity $\rho := \mathrm{GS}_{\mathrm{static}}(f)$. Let $\beta \in (0,1), \delta > 0$ and let $\alpha = 16 \log_{1+\beta}(r)\rho \cdot \ln(2T/\delta)/\varepsilon$. There exists an ε-differentially private algorithm for computing f with multiplicative error $(1+\beta)$, additive error α and failure probability δ.*

Proof. We reduce the dynamic problem to a problem that can be solved using the sparse vector technique: Assume that we are given the whole sequence of dynamic updates as a database D in advance. We will remove this assumption later.

Formally, let $D := \mathcal{G} = (G_1, \ldots, G_T)$ be the input to the dynamic problem and let f_i be the function that evaluates f on the i-th entry G_i (e.g., graph) of such a database. To initialize our algorithm, we run `Initialize`$(D, \rho, \epsilon, r, \beta)$. Then we call `Process`$(f_i)$ for all f_i, $i \in [T]$. We claim that the output of `Process`(f_i) is an approximation to $f_i(D) = f(G_i)$ with multiplicative error $(1+\beta)$, additive error α and failure probability δ over the coins of the whole algorithm, and that the algorithm is ϵ-differentially private.

ERROR. We condition on the event that all noises in Algorithm 6.2.1 are small, i.e., $|\zeta| < \alpha/4$ and, for all i, $|\nu_i| < \alpha/4$. It follows from the definition of the Laplace distribution and a union bound over all queries that this event occurs with probability at least $1 - e^{-\alpha\varepsilon_1/(4\rho)} - T \cdot e^{-\alpha\varepsilon_2/(8\log_{1+\beta}(r)\rho)} \geq 1 - \delta$. Consider query $f_i(D)$ and let $f_i' := f_i(D) + \nu_i - \zeta$. After the while-loop of `Process`(f_i), we know that $f_i(D) - \alpha/2 \leq (1+\beta)^{k_i}$ and $f_i(D) + \alpha/2 \geq f_{i-1}(D) + \alpha/2 \geq (1+\beta)^{k_i - 1}$ due to the monotonicity of $f_1(D), \ldots, f_T(D)$. We analyze the return value of `Process`(f_i) and show that it is a $(1+\beta)$ approximation with additive error α:

$$f_i(D) - \alpha \leq \left((1+\beta)^{k_i} + \frac{\alpha}{2}\right) - \alpha \leq (1+\beta)^{k_i}$$
$$(1+\beta) f_i(D) + \alpha \geq (1+\beta)\left((1+\beta)^{k_i - 1} - \frac{\alpha}{2}\right) + \alpha \geq (1+\beta)^{k_i}.$$

Since $f(G_T) \leq r = (1+\beta)^{\log_{1+\beta}(r)}$, it follows that the algorithm has a multiplicative error $(1+\beta)$ and an additive error of at most α with probability at least $1-\delta$.

PRIVACY. The privacy follows directly from Lemma 6.2.6 because the return value $(1+\beta)^{k_i}$ is derived from the answers of Algorithm 6.2.1, which is ϵ-private, and public knowledge (i.e., the fact that $k_0 = 0$).

REMOVING THE ASSUMPTION. Observe that to answer query f_i on D, Algorithm 6.2.1 only needs access to G_i or, equivalently, access to the first i updates of the dynamic update sequence. In other words, G_i is read for the first time after answering query f_{i-1}. Therefore, we may ask all previous queries f_j, $j < i$, even if G_i is not available yet, e.g., when processing update j. For dynamic graph sequences and edge-adjacency or node-adjacency, any pair of same-length prefixes of two graph sequences $\mathcal{G}_1, \mathcal{G}_2$ is neighboring if $\mathcal{G}_1$ and $\mathcal{G}_2$ are neighboring (see Definitions 6.2.2 and 6.2.3). Therefore, privacy is not affected by making G_i only available to Algorithm 6.4.1 just before the i-th query, as required by the dynamic setting. □

6.5 Lower Bounds for Event-Level Privacy

In this section we show lower bounds for the error of edge-differentially private algorithms in the partially dynamic setting. We consider the problem of releasing the weight of a minimum spanning tree, the minimum cut and maximum weight matching. To derive the bounds we reduce differentially private counting in binary streams to these problems and apply a lower bound of Dwork et al. [54], which we restate here.

Theorem 6.5.1 (Lower bound for counting in binary streams [54])**.** *Any differentially private event-level algorithm for counting over T rounds must have error $\Omega(\log T)$ (even with $\varepsilon = 1$).*

Note that any lower bound for the incremental setting can be transferred to the decremental setting, using the same reductions but proceeding in reverse.

6.5.1 Minimum Spanning Tree

We show a lower bound for the error in ε-edge-differentially private estimation of the weight of a minimum spanning tree when the weights are bounded by W.

Lemma 6.5.2. *For any pair of adjacent binary streams σ, σ' of length T there exists a pair of incremental edge-weighted graph sequences $\mathcal{G}$, $\mathcal{G}'$ that are edge-adjacent, such that the weights of minimum spanning trees in $\mathcal{G}$ and $\mathcal{G}'$ are*

$$w_{\mathrm{MST}}(G_t) = W \sum_{i=1}^{t} \sigma(i) + T$$

and

$$w_{\mathrm{MST}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i) + T,$$

respectively, at all time steps $t \in \{1, \ldots, T\}$, where edge weights are in $\{1, \ldots, W\}$.

Proof. We construct graph sequences on the node set

$$V = \{0,\dots,T\} \cup \{T+1,\dots,2T+1\}. \tag{20}$$

The nodes $1,\dots,T$ will form a path in these graph sequences. To achieve this we use the edge set $E_0 = \{\{i-1,i\} \mid i = 1,\dots,T\}$ with weight 1 for each edge and initial the graph sequences with the graph $G_0 = (V, E_0)$. The edges in E form a minimum spanning tree of G_0 of weight T.

For any two adjacent binary streams σ, σ' we construct corresponding graph sequences $\mathcal{G}, \mathcal{G}'$ with initial graph G_0. At all times the node set of the graphs in $\mathcal{G}$ and $\mathcal{G}'$ is V, as defined in (20). We now define for each time t the sets of edge-insertions for $\mathcal{G}$ and $\mathcal{G}'$. Let $e_0^t = \{t, (t+2) \bmod T\}$ with $w(e_0^t) = W$ and $e_1^t = \{T+t, T+t+1\}$ with $w(e_1^t) = W$. At time t we insert

$$\partial E_t^+ = \begin{cases} \{e_0^t\} & \text{if } \sigma(t) = 0, \\ \{e_0^t, e_1^t\} & \text{if } \sigma(t) = 1, \end{cases}$$

$$\partial E_t^{+\prime} = \begin{cases} \{e_0^t\} & \text{if } \sigma'(t) = 0, \\ \{e_0^t, e_1^t\} & \text{if } \sigma'(t) = 1, \end{cases}$$

into $\mathcal{G}$ and $\mathcal{G}'$, respectively. Note that $\partial E_t^+ \neq \partial E_t^{+\prime}$ only if $\sigma(t) \neq \sigma'(t)$. Thus, $\mathcal{G}$ and $\mathcal{G}'$ only differ in the insertion of a single edge and are adjacent.

Inserting an edge e_0^t never changes the weight of a minimum spanning tree, since the edges e^i defined above make up a spanning tree of the nodes $\{1,\dots,T\}$ and have lower weight. Inserting an edge e_1^t increases the weight of the minimum spanning tree by W. Thus, at any time t, the weights of the minimum spanning trees in $\mathcal{G}$ and $\mathcal{G}'$ are

$$w_{\mathrm{MST}}(G_t) = W \sum_{i=1}^{t} \sigma(i) + T$$

and

$$w_{\mathrm{MST}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i) + T,$$

respectively, which concludes the proof. □

Lemma 6.5.3. *For any pair of adjacent binary streams σ, σ' of length T there exists a pair of incremental edge-weighted graph sequences $\mathcal{G}, \mathcal{G}'$ that are node-adjacent, such that the weights of minimum spanning trees in $\mathcal{G}$ and $\mathcal{G}'$ are*

$$w_{\mathrm{MST}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{MST}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i),$$

respectively, at all time steps $t \in \{1,\dots,T\}$, where edge weights are in $\{1,\dots,W\}$.

Proof. Let $G_0 = (v_0, \emptyset)$. Let $\mathcal{G}, \mathcal{G}'$ be graph sequences with initial graph G_0.
At time t we insert the node sets

$$\begin{aligned} \partial V_t^+ &= \begin{cases} \{u_t\} & \text{if } \sigma(t) = 0, \\ \{u_t, v_t\} & \text{if } \sigma(t) = 1, \end{cases} \\ \partial V_t^{+\prime} &= \begin{cases} \{u_t\} & \text{if } \sigma'(t) = 0, \\ \{u_t, v_t\} & \text{if } \sigma'(t) = 1, \end{cases} \end{aligned} \tag{21}$$

into $\mathcal{G}$ and $\mathcal{G}'$, respectively. When inserting v_t we also insert the edge $\{v_0, v_t\}$ with weight W. The nodes u_t stay isolated throughout the graph sequences. The weight of the minimum spanning tree is the number of nodes v_t that were inserted into the graph sequence, multiplied by the maximum edge weight W. That is, at any time t, the weights of the minimum spanning trees in $\mathcal{G}$ and $\mathcal{G}'$ are

$$w_{\text{MST}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\text{MST}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i)$$

respectively, which concludes the proof. □

Theorem 6.5.4. *Any ε-edge-differentially private algorithm to compute the weight of a minimum spanning tree in an incremental graph sequence of length T must have additive error $\Omega(W \log T)$ when the edge weights are in $\{1, \dots, W\}$, even with $\varepsilon = 1$.*

Proof. Suppose there exists an ε-edge-differentially private algorithm to compute the weight of a minimum spanning tree in an incremental graph sequence with error $o(W \log T)$ at all time steps. Using this algorithm and the reduction from Lemma 6.5.2, we can compute the sum of any binary stream with error $o(\log T)$ while preserving ε-differential privacy. This contradicts Theorem 6.5.1. □

Theorem 6.5.5. *Any ε-node-differentially private algorithm to compute the weight of a minimum spanning tree in an incremental graph sequence of length T must have additive error $\Omega(W \log T)$ when the edge weights are in $\{1, \dots, W\}$, even with $\varepsilon = 1$.*

Proof. Suppose there exists an ε-node-differentially private algorithm to compute the weight of a minimum spanning tree in an incremental graph sequence with error $o(W \log T)$ at all time steps. Using this algorithm and the reduction from Lemma 6.5.3, we can compute the sum of any binary stream with error $o(\log T)$ while preserving ε-differential privacy. This contradicts Theorem 6.5.1. □

6.5.2 Weighted Minimum Cut

We can prove a similar lower bound for edge-differentially private weighted minimum cut in incremental graph sequences. We again reduce differentially private counting in binary streams to differentially private minimum cut in incremental graph sequences.

Lemma 6.5.6. *For any pair of adjacent binary streams σ, σ' of length T there exists a pair of incremental edge-weighted graph sequences $\mathcal{G}$, $\mathcal{G}'$ that are edge-adjacent, such that the weights of the minimum cut in $\mathcal{G}$ and $\mathcal{G}'$ are*

$$w_{\mathrm{CUT}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{CUT}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i)$$

respectively, at all time steps $t \in \{1,\dots,T\}$, where edge weights are in $\{1,\dots,W\}$.

Proof. At the core of our reduction is a $(T+1)$-edge-connected graph with sufficiently many missing edges. We use the $(T+1)$-dimensional hypercube graph $Q = (V_Q, E_Q)$. Each node $v \in V_Q$ is associated with a vector $\mathbf{v} \in \{0,1\}^{T+1}$. We associate every time step $t \in \{1,\dots,T\}$ with two nodes in Q, which correspond to the following vectors:

$$b_t = (b_1,\dots,b_T,0)$$
$$\text{and} \quad \widehat{b_t} = (1-b_1,\dots,1-b_T,1),$$

where $b_1,\dots,b_T \in \{0,1\}$ such that $t = \sum_{i=1}^{T} b_i \cdot 2^{i-1}$. Since b_t and $\widehat{b_t}$ differ in at least 2 elements, they are not connected by an edge in Q.

In the following, every edge has weight W, including the edges in Q. Let $G_0 = (\{v^*\}, \emptyset) \cup Q$. G_0 has a minimum cut of weight 0, since v^* is isolated. Let $\mathcal{G}$, $\mathcal{G}'$ be incremental graph sequences with initial graph G_0. For every time step we define the edges $e_0^t = \{b_t, \widehat{b_t}\}$ and $e_1^t = \{v^*, b_t\}$. At time t we insert

$$\partial E_t^+ = \begin{cases} \{e_0^t\} & \text{if } \sigma(t) = 0, \\ \{e_0^t, e_1^t\} & \text{if } \sigma(t) = 1, \end{cases}$$
$$\partial E_t^{+\prime} = \begin{cases} \{e_0^t\} & \text{if } \sigma'(t) = 0, \\ \{e_0^t, e_1^t\} & \text{if } \sigma'(t) = 1, \end{cases}$$

into $\mathcal{G}$ and $\mathcal{G}'$, respectively. Note that these graph sequences differ in exactly one edge exactly when $\sigma(t) \neq \sigma'(t)$.

Denote by $\deg_{\mathcal{G}}(v^*, t)$ the degree of v^* in $\mathcal{G}$ at time t. We observe that the degree of v^* increases by 1 in the graph sequence if and only if there is a 1 in the corresponding binary stream. Otherwise, its degree does not change. The minimum cut is always the trivial cut around v^*, since the degree of v^* never exceeds T and Q is $(T+1)$-edge-connected. Thus, the weight of the minimum cut at time t is

$$w_{\mathrm{CUT}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{CUT}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i),$$

which concludes the proof. □

Lemma 6.5.7. *For any pair of adjacent binary streams σ, σ' of length T there exists a pair of incremental edge-weighted graph sequences $\mathcal{G}, \mathcal{G}'$ that are node-adjacent, such that the weights of the minimum cut in $\mathcal{G}$ and $\mathcal{G}'$ are*

$$w_{\mathrm{CUT}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{CUT}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i)$$

respectively, at all time steps $t \in \{1, \dots, T\}$, where edge weights are in $\{1, \dots, W\}$.

Proof. Let $G_0 = (\{u_0, v_0\}, \emptyset)$. Let $\mathcal{G}, \mathcal{G}'$ be graph sequences with initial graph G_0. In the following all edges have weight W.

At time t we insert the node sets

$$\partial V_t^+ = \begin{cases} \{u_t\} & \text{if } \sigma(t) = 0, \\ \{u_t, v_t\} & \text{if } \sigma(t) = 1, \end{cases}$$

$$\partial V_t^{+\prime} = \begin{cases} \{u_t\} & \text{if } \sigma'(t) = 0, \\ \{u_t, v_t\} & \text{if } \sigma'(t) = 1, \end{cases}$$

into $\mathcal{G}$ and $\mathcal{G}'$, respectively. Along with every node u_t, we insert the edges $\{u_t, u_j\}$ for all $0 \leq j < t$. If we insert v_t, we also insert the edges $\{v_t, v_j\}$ for all $0 \leq j < t$ and the edge $\{v_t, u_0\}$.

Let $U_t = \{u_0, \dots, u_t\}$ and $V_t = \{v_0, \dots, v_t\}$. The subgraphs induced by U_t and V_t are always complete. The cut (U_t, V_t) always has minimum weight. The weight of this cut in $\mathcal{G}$ increases by W only if $\sigma(t) = 1$; otherwise, it does not change. The same holds for $\mathcal{G}'$ and σ'. Thus, the weight of the minimum cut at time t is

$$w_{\mathrm{CUT}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{CUT}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i),$$

which concludes the proof. □

Theorem 6.5.8. *Any ε-edge-differentially private algorithm to compute the weight of a minimum cut in an incremental graph sequence of length T must have additive error $\Omega(W \log T)$ when the edge weights are in $\{1, \dots, W\}$, even with $\varepsilon = 1$.*

Proof. Suppose there exists an ε-edge-differentially private algorithm to compute the weight of a minimum cut in an incremental graph sequence with error $o(W \log T)$ at all time steps. Using this algorithm and the reduction from Lemma 6.5.6, we can compute the sum of any binary stream with error $o(\log T)$ while preserving ε-differential privacy. This contradicts Theorem 6.5.1. □

Theorem 6.5.9. *Any ε-node-differentially private algorithm to compute the weight of a minimum cut in an incremental graph sequence of length T must have additive error $\Omega(W \log T)$ when the edge weights are in $\{1,\dots,W\}$, even with $\varepsilon = 1$.*

Proof. Suppose there exists an ε-node-differentially private algorithm to compute the weight of a minimum cut in an incremental graph sequence with error $o(W \log T)$ at all time steps. Using this algorithm and the reduction from Lemma 6.5.7, we can compute the sum of any binary stream with error $o(\log T)$ while preserving ε-differential privacy. This contradicts Theorem 6.5.1. □

6.5.3 Maximum Weighted Matching

Lemma 6.5.10. *For any pair of adjacent binary streams σ, σ' of length T there exists a pair of incremental edge-weighted graph sequences $\mathcal{G}$, $\mathcal{G}'$ that are edge-adjacent, such that the weights of the maximum weighted matching in $\mathcal{G}$ and $\mathcal{G}'$ are*

$$w_{\mathrm{M}}(G_t) = W \sum_{i=1}^{t} \sigma(i) + W \cdot T$$

and

$$w_{\mathrm{M}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i) + W \cdot T,$$

respectively, at all time steps $t \in \{1,\dots,T\}$, where edge weights are in $\{1,\dots,W\}$.

Proof. Let $G_0 = (U \cup V, E)$, where $U = \{u_1,\dots,u_{2\cdot T}\}$, $V = \{v_1,\dots,v_{2\cdot T}\}$ and $E = \bigcup_{i=1}^{T} \{u_i, u_{(i+1) \bmod T}\}$. All edges have weight W. Let $\mathcal{G}$ and $\mathcal{G}'$ be graph sequences with initial graph G_0.

For all times $t \in \{1,\dots,T\}$ we define the edges $e_0^t = \{u_t, v_t\}$ with $w(e_0^t) = 1$ and $e_1^t = \{u_{T+t}, v_{t+T}\}$ with $w(e_1^t) = W$. At time t we insert

$$\partial E_t^+ = \begin{cases} \{e_0^t\} & \text{if } \sigma(t) = 0, \\ \{e_0^t, e_1^t\} & \text{if } \sigma(t) = 1, \end{cases}$$

$$\partial E_t^{+\prime} = \begin{cases} \{e_0^t\} & \text{if } \sigma'(t) = 0, \\ \{e_0^t, e_1^t\} & \text{if } \sigma'(t) = 1, \end{cases}$$

into $\mathcal{G}$ and $\mathcal{G}'$, respectively. Note that these graph sequences differ in exactly one edge exactly when $\sigma(t) \neq \sigma'(t)$.

G_0 has a matching of weight $W \cdot T$, which consists of the edges in E. Inserting an edge e_0^t for any t does not change the weight of the maximum matching. Inserting an edge e_1^t however increases the weight of the matching by W, since two previously isolated nodes are now connected by this new edge. Thus, the weight of the maximum matching at time t is

$$w_{\mathrm{M}}(G_t) = W \sum_{i=1}^{t} \sigma(i) + W \cdot T$$

and

$$w_{\mathrm{M}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i) + W \cdot T,$$

which concludes the proof. □

We can also reduce differentially private binary counting to node-differentially private maximum weight matching in incremental graph sequences.

Lemma 6.5.11. *For any pair of adjacent binary streams σ, σ' of length T there exists a pair of incremental edge-weighted graph sequences $\mathcal{G}$, $\mathcal{G}'$ that are node-adjacent, such that the weights of the maximum weighted matching in $\mathcal{G}$ and $\mathcal{G}'$ are*

$$w_{\mathrm{M}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{M}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i)$$

respectively, at all time steps $t \in \{1,\dots,T\}$, where edge weights are in $\{1,\dots,W\}$.

Proof. Let $G_0 = (\{v_1,\dots,v_T\},\emptyset)$. Let $\mathcal{G}$, $\mathcal{G}'$ be graph sequences with initial graph G_0.

At time t, we insert a node u_t into $\mathcal{G}$ and $\mathcal{G}'$. If $\sigma(t) = 1$ we additionally insert node v'_t along with the edge $\{v'_t, v_t\}$ with weight W into $\mathcal{G}$. Similarly, if $\sigma'(t) = 1$ we insert node v'_t along with the edge $\{v'_t, v_t\}$ with weight W into $\mathcal{G}'$.

Then, the weight of the maximum matching in $\mathcal{G}$ at time t increases by W if $\sigma(t) = 1$; otherwise, it does not change. The same holds for σ' and $\mathcal{G}'$. Thus, the weight of the maximum matching at time t is

$$w_{\mathrm{M}}(G_t) = W \sum_{i=1}^{t} \sigma(i)$$

and

$$w_{\mathrm{M}}(G'_t) = W \sum_{i=1}^{t} \sigma'(i),$$

which concludes the proof. □

Theorem 6.5.12. *Any ε-edge-differentially private algorithm to compute the weight of a maximum weight matching in an incremental graph sequence of length T must have additive error $\Omega(W \log T)$ when the edge weights are in $\{1,\dots,W\}$, even with $\varepsilon = 1$.*

Proof. Suppose there exists an ε-edge-differentially private algorithm to compute the weight of a maximum weight matching in an incremental graph sequence with error $o(W \log T)$ at all time steps. Using this algorithm and the reduction from Lemma 6.5.10, we can compute the sum of any binary stream with error $o(\log T)$ while preserving ε-differential privacy. This contradicts Theorem 6.5.1. □

Theorem 6.5.13. *Any ε-node-differentially private algorithm to compute the weight of a maximum weight matching in an incremental graph sequence of length T must have additive error $\Omega(W \log T)$ when the edge weights are in $\{1,\dots,W\}$, even with $\varepsilon = 1$.*

Proof. Suppose there exists an ε-node-differentially private algorithm to compute the weight of a maximum weight matching in an incremental graph sequence with error $o(W \log T)$ at all time steps. Using this algorithm and the reduction from Lemma 6.5.11, we can compute the sum of any binary stream with error $o(\log T)$ while preserving ε-differential privacy. This contradicts Theorem 6.5.1. □

6.5.4 Subgraph Counting, High-Degree Nodes and Degree Histogram

We show that, for the number of high-degree nodes, the degree histogram and the subgraph counting problems adjacent binary streams can be reduced to adjacent incremental graph sequences. This implies a lower bound of $\Omega(\log T)$ on the error for these problems.

Lemma 6.5.14. *There exist functions $\mathcal{G}_\tau$, $\mathcal{G}_h$, $\mathcal{G}_\Delta$, $\mathcal{G}_k$, that map a binary stream to an incremental graph sequence of the same length, such that the following holds: At time t*

1. *the number of nodes of degree at least τ in $\mathcal{G}_\tau(\sigma)$,*
2. *the number of nodes of degree 2 in $\mathcal{G}_h(\sigma)$,*
3. *the number of triangles in $\mathcal{G}_\Delta(\sigma)$ and*
4. *the number of k-stars in $\mathcal{G}_k(\sigma)$*

is equal to $\sum_{i=1}^{t} \sigma(i)$. Furthermore, all functions map adjacent binary streams to edge-adjacent graph sequences.

Proof. Let σ be a binary stream.

1. $\mathcal{G}_\tau$ outputs a graph sequence with initial graph $G_\tau = \cup_{i=1}^{T} S_i$. Each S_i is a τ-star with one missing edge. At time t, if $\sigma(t) = 1$, then the missing edge is inserted into S_t.
2. $\mathcal{G}_h$ outputs a graph sequence on the nodes $V_h = \cup_{i=1}^{T} \{a_i, b_i, c_i\}$. Initially, there are edges $\{b_i, c_i\}$ for each $i \in \{1, \dots, T\}$. At time t, if $\sigma(t) = 1$, then the edge $\{a_t, b_t\}$ is inserted. Note that $\deg(b_t) = \sigma(t) + 1$ and $\deg(a_i), \deg(c_i) \leq 1$ for all t.
3. $\mathcal{G}_\Delta$ outputs a graph sequence on the nodes $V_\Delta = \cup_{i=1}^{T} \{a_i, b_i, c_i\}$. Initially, there are edges $\{a_i, b_i\}$ and $\{b_i, c_i\}$ for each $i \in \{1, \dots, T\}$. At time t, if $\sigma(t) = 1$, then the edge $\{a_t, c_t\}$ is inserted.
4. $\mathcal{G}_k$ outputs a graph sequence with initial graph $G_k = \cup_{i=1}^{T} S_i$. Each S_i is a k-star with one missing edge. At time t, if $\sigma(t) = 1$, then the missing edge is inserted into S_t.

In all graph sequences no edge is inserted if $\sigma(t) = 0$; if $\sigma(t) = 1$, then a single edge is inserted. Thus, the functions map adjacent binary streams to edge-adjacent graph sequences. □

Lemma 6.5.15. *There exist functions $\mathcal{G}'_\tau$, $\mathcal{G}'_h$, $\mathcal{G}'_e$, $\mathcal{G}'_\Delta$, $\mathcal{G}'_k$, that map a binary stream to an incremental graph sequence of the same length with maximum degree $D > 3$, $\tau, k \leq D$, such that the following holds: Let σ be a binary stream. At time t*

1. *the number of nodes of degree at least τ in $\mathcal{G}'_\tau(\sigma)$,*

2. *the number of nodes of degree 2 in* $\mathscr{G}'_h(\sigma)$,
3. *the number of edges in* $\mathscr{G}'_e(\sigma)$,
4. *the number of triangles in* $\mathscr{G}'_\Delta(\sigma)$,
5. *the number of k-stars in* $\mathscr{G}'_k(\sigma)$

is equal to $D\sum_{i=1}^{t}\sigma(i)$*. Furthermore, all functions map adjacent binary streams to node-adjacent graph sequences.*

Proof. Let σ be a binary stream.

1. $\mathscr{G}'_\tau$ outputs a graph sequence with initial graph $G_\tau = \cup_{i=1}^{T} S_i$. Each S_i is the union of $(D-1)$-many $(\tau-1)$-stars. At time t, if $\sigma(t)=1$, then a node v_t is inserted into S_t, along with edges from v_t to the central node of each $(\tau-1)$-star in S_t. Since $D \geq \tau$, the number of τ-stars in S_t is D if $\sigma(t)=1$ and zero otherwise. In S_t the centers of the τ-stars have degree $\geq \tau$, all other nodes have degree $< \tau$.
2. $\mathscr{G}'_h$ outputs a graph sequence with initial graph $G_h = \cup_{i=1}^{T} M_i$, where M_i consists of D pairwise connected nodes, i.e., $M_i = (\bigcup_{j=1}^{D}\{a_j, b_j\}, \bigcup_{j=1}^{D}\{\{a_j, b_j\}\})$. At time t, if $\sigma(t)=1$, then a node v_t is inserted into M_t along with edges $\{v_t, a_j\}$ for all $j = 1,\dots,D$. The number of nodes of degree 2 is D in M_t if $\sigma(t)=1$ and zero otherwise.
3. $\mathscr{G}'_e$ outputs a graph sequence with initial graph $G_e = \cup_{i=1}^{T} V_i$, where $V_i = (\{1,\dots,D\},\emptyset)$. At time t, if $\sigma(t)=1$, then a node v_t is inserted into V_t along with edges from v_t to all nodes in V_t. The number of edges is D in V_t if $\sigma(t)=1$ and zero otherwise.
4. $\mathscr{G}'_\Delta$ outputs a graph sequence with initial $G_\Delta = \cup_{i=1}^{T} C_i$, where C_i is a cycle of D nodes. At time t, if $\sigma(t)=1$, then a node v_t is inserted into C_t along with edges from v_t to every node in C_t. Thus, if $\sigma(t)=1$ the number of triangles in C_t is D; otherwise, it is zero.
5. $\mathscr{G}'_k$ outputs a graph sequence with initial graph $G_\tau = \cup_{i=1}^{T} S_i$. Each S_i is the union of $(D-1)$-many $(\tau-1)$-stars. At time t, if $\sigma(t)=1$, then a node v_t is inserted into S_t, along with edges from v_t to the central node of each $(\tau-1)$-star in S_t. Since $D \geq \tau$, the number of τ-stars in S_t is D if $\sigma(t)=1$ and zero otherwise.

In all graph sequences no node is inserted if $\sigma(t)=0$; if $\sigma(t)=1$, then a single node is inserted. Thus, the functions map adjacent binary streams to node-adjacent graph sequences. □

Theorem 6.5.16. *Any ε-edge-differentially private algorithm to compute*

1. *the number of high-degree nodes,*
2. *the degree histogram,*
3. *the number of edges,*
4. *the number of triangles,*
5. *the number of k-stars*

in an incremental graph sequence of length T must have additive error $\Omega(\log T)$, even with $\varepsilon = 1$.

Proof. By Lemma 6.5.14, adjacent binary streams can be encoded in edge-adjacent incremental graph sequences of the same length. Counting the number of edges is equivalent to binary counting. The claim follows by Theorem 6.5.1. □

Theorem 6.5.17. *Any ε-node-differentially private algorithm to compute*

1. *the number of high-degree nodes,*
2. *the degree histogram,*
3. *the number of edges,*
4. *the number of triangles,*
5. *the number of k-stars*

in an incremental graph sequence of length T must have additive error $\Omega(D \log T)$, where D is the maximum degree, even with $\varepsilon = 1$.

Proof. By Lemma 6.5.15, adjacent binary streams can be encoded in node-adjacent incremental graph sequences of the same length. The claim follows by Theorem 6.5.1. □

6.6 Lower Bound for User-Level Privacy

We show that for several fundamental problems on dynamic graphs like minimum spanning tree and minimum cut, a differentially private algorithm with edge-adjacency on user-level must have an additive error that is linear in the maximum function value. Technically, we define the *spread* of a graph function as the maximum difference of the function's value on any two graphs. Then, we show that any algorithm must have an error that is linear in the graph function's spread. We write this section in terms of edge-adjacency but the corresponding result for node-adjacency carries over. See Table 6.1.1 on page 114 for the resulting lower bounds.

Definition 6.6.1 (adjacency transformation)**.** Let G_1 and G_2 be a pair of graphs. We define $\tau(G_1, G_2)$ to be an update sequence $u_1, \ldots, u_\ell$ of minimum length that transforms G_1 into G_2. We denote the graph sequence that results from applying $\tau(G_1, G_2)$ to G_1 by $\mathrm{T}(G_1, G_2)$, i.e., $\mathrm{T}(G_1, G_2) = (G_1, G_1 \oplus u_1, (G_1 \oplus u_1) \oplus u_2, \ldots, G_2)$.

Definition 6.6.2 (spread)**.** Let $s, \ell : \mathbb{N} \to \{2i \mid i \in \mathbb{N}\}$ be functions. A graph function f has spread $(s(n), \ell(n))$ on inputs of size n if, for every n, there exist two graphs G_1, G_2 of size n so that $|f(G_1) - f(G_2)| \geq s(n)$ and $|\tau(G_1, G_2)| = \ell(n)$. Furthermore, f *spares an edge e* if $e \in E(G_1) \cap E(G_2)$.

Theorem 6.6.3. *Let $\epsilon, \delta > 0$ and let f be a graph function with spread (s, ℓ) that spares an edge e. For streams of length T on graphs of size n, where $T > 2\ell \log(e^{4\epsilon\ell}/(1-\delta)) \in O(\epsilon\ell^2 + \ell \log(1/(1-\delta)))$, every ϵ-differentially private dynamic algorithm with user-level edge-adjacency that computes f with probability at least $1 - \delta$ must have error $\Omega(s(n))$.*

Proof. For any update u, let u^{-1} be the update that reverses u. For the sake of contradiction, assume that there exists an ϵ-differentially private graph algorithm $\mathcal{A}$ with edge-adjacency on user-level that with probability at least $1-\delta$ computes f with error less than $s(n)/2$. Let $n > 0$, $s := s(n)$, $\ell = \ell(n)$ and let G_1, G_2 be graphs of size n with spread (s,ℓ) on f that spares an edge e. We assume below wlog that ℓ is even, if it is odd the proof can be easily adapted. Let u be the update operation that inserts e into the current graph (see below). Let $(o_0,\dots,o_{\ell-1}) := \tau(G_1,G_2)$ and let $(o'_0,\dots,o'_{\ell-1}) := \tau(G_2,G_1)$. Without loss of generality, assume that T is a multiple of 2ℓ. Let $\mathcal{B}$ be the set of bit strings of length $T/(2\ell)$. For every bit string $b = (b_0,\dots,b_{T/(2\ell)-1}) \in \mathcal{B}$, we construct a unique graph sequence $\mathcal{G}_b = (H_0,\dots,H_T)$ as follows, resulting in $2^{T/(2\ell)}$ different graph sequences.

The sequence is partitioned into phases of length 2ℓ. Each phase has a *type*: it is either a *forward* phase or a *backward* phase. The sequence starts with a forward phase and alternates between forward and backward phases. Intuitively, forward phases transform G_1 into G_2, while backward phases transform G_2 into G_1. This takes ℓ updates. To generate an exponential number of different but "close" graph sequences each phase also contains ℓ *placeholder updates*. The purpose of a placeholder update is to basically not modify the graph in any of the graph sequences. Thus the first placeholder update updates e and the next update is the inverse operation. This is repeated $\ell/2$ times. For every phase, a corresponding bit in b decides whether the transformation happens *before* the placeholder updates or *after* placeholder updates. Thus, $2^{T/(2\ell)}$ different graph sequences are generated, where the graphs at position $2i\ell$ for any integer $i \geq 0$ in all these sequences are identical.

Formally, for integer $i = 0,1,...,T/(2\ell)-1$, the bit b_i corresponds to the phase $H_{2i\ell},\dots,H_{2(i+1)\ell-1}$ in the sequence $\mathcal{G}_b$. Bit b_i corresponds to graphs in a forward phase if and only if i is even. We define $\mathcal{G}_b$ as follows:

FORWARD PHASE, $b_i = 0$: For any j, $0 \leq j < \ell$ (i.e., the first half of the phase), we define $H_{2i\ell+j+1} := H_{2i\ell+j} \oplus o_j$. The second half of the phase, i.e., $H_{(2i+1)\ell},\dots,H_{2(i+1)\ell-1}$, is defined by alternating between $H_{(2i+1)\ell}$ and $H_{(2i+1)\ell} \oplus u$, which are placeholder updates.

FORWARD PHASE, $b_i = 1$: This phase results from swapping the first and the second half of the forward phase with $b_i = 0$. In particular, the first half of the phase, i.e., $H_{2i\ell},\dots,H_{(2i+1)\ell-1}$, is defined by alternating between $H_{2i\ell}$ and $H_{2i\ell} \oplus u$, which are placeholder updates. For any j, $\ell \leq j < 2\ell$, $H_{2i\ell+j+1} := H_{2i\ell+j} \oplus o_{j-\ell}$.

BACKWARD PHASE, $b_i \in \{0,1\}$: The graphs $H_{2i\ell},\dots,H_{2(i+1)\ell-1}$ are defined analogously to a forward phase with $b_i = 0$ or $b_i = 1$, respectively. The only difference is that instead of $o_0,\dots,o_{\ell-1}$, the updates $o'_0,\dots,o'_{\ell-1}$ are used.

It follows from the construction that for any bit string b and its corresponding graph sequence $(H_0,\dots,H_T)$, it holds that $H_i = G_1$ if $i \mod 4\ell = 0$ and $H_i = G_2$ if $i \mod 4\ell = 2\ell$. However, for every pair of bit strings b, b', $b \neq b'$, and their corresponding graph sequences $(H_0,\dots,H_{T-1})$ and $(H'_0,\dots,H_{T-1})$, there exists an i with $i \mod 2\ell = \ell$, so that $|f(H_i) - f(H'_i)| \geq s$. This stems from the fact that f has spread (s,ℓ) and that b and b' must differ in at least one bit, say $b_i = 0$ and $b'_i = 1$. Thus, by construction, $H_{(2i+1)\ell} = G_2$ (as $\tau(G_1,G_2)$ has already been executed in this phase) and $H'_{(2i+1)\ell} = G_1$ (as $\tau(G_1,G_2)$ has not yet been executed in this phase).

Next we show that any graph sequence $\mathcal{G}_b$ is on user-level "edge-close" to a very generic graph sequence. More specifically, let $\mathcal{G}'$ be the sequence of graphs that alternates between G_1 and $G_1 \oplus u$, i.e., $(G_1, G_1 \oplus u, G_1, G_1 \oplus u, \ldots)$. We argue that there is a (short) sequence $\Gamma_1, \Gamma_2, \ldots \Gamma_{2\ell+1}$ of graph sequences such that Γ_i and Γ_{i+1} are *user-level edge-adjacent* graph sequences with $\Gamma_1 = \mathcal{G}_b$ and $\Gamma_{2\ell+1} = \mathcal{G}'$. In other words, the sequence of graph sequences transforms $\mathcal{G}_b$ into $\mathcal{G}'$. Instead of talking about a transforming graph sequence we use below the notation of *user-level edge-adjacency operation*: It takes one operation to transform a graph sequence Γ_i into a user-level edge-adjacent graph sequence Γ_{i+1}.

We now give the details: For any b, each phase of $\mathcal{G}_b$ can be divided into two halves $(H_{2i\ell}, \ldots, H_{(2i+1)\ell-1})$ and $(H_{(2i+1)\ell}, \ldots H_{(2(i+1)\ell-1)})$: in one half (the first if $b_i = 0$, the latter if $b_i = 1$), the updates are already alternating between u, i.e., inserting e, and u^{-1}, i.e., deleting e. Therefore, we only need to transform the updates on the other half into a sequence that alternates between u and u^{-1}. The other half is either $\mathrm{T}(G_1, G_2)$ (in forward phases) or $\mathrm{T}(G_2, G_1)$ (in backward phases). By Lemma 6.6.4 (see below), there is a sequence of $2\ell + 2$ graph sequences $\mathcal{G}'_0, \ldots, \mathcal{G}'_{2\ell+1}$ that starts with $\mathcal{G}'_0 = \mathrm{T}(G_1, G_2)$ and ends with $\mathcal{G}'_{2\ell+1} = \mathcal{G}'$. Furthermore, for each $0 \leq i \leq 2\ell$ there is an edge e_i such that $\mathcal{G}'_i$ and $\mathcal{G}'_{i+1}$ are *event-level* edge-adjacent on e_i. Similarly, such sequence and edges $(e'_0, \ldots, e'_{2\ell})$ exist for $\mathrm{T}(G_2, G_1)$. Recall that *user-level edge-adjacency* allows to modify a graph sequence by updating an arbitrary number of graphs in the sequence *using the same edge e, inserting e into some graphs that do not contain e and removing it from some of the graphs that contain e.* Also recall that $\mathcal{G}_b$ consists of many repetitions of $\mathrm{T}(G_1, G_2)$. Thus one user-level edge-adjacency operation using the edge e_1 allows to modify *all* occurrences of $\mathrm{T}(G_1, G_2)$ to reflect the update of edge e_1. Afterwards a second user-level edge-adjacency operation using e_2 modifies the resulting sequence etc. After $2\ell + 1$ user-level edge-adjacency operations all occurrences of $\mathrm{T}(G_1, G_2)$ have been transformed into sequences that alternate G_1 and $G_1 \oplus u$. The same can be done for the repetitions of $\mathrm{T}(G_2, G_1)$. This shows that at most $2\ell + 1$ user-level edge-adjacency operations suffice to transform $\mathcal{G}_b$ into $\mathcal{G}'$.

Recall that we assumed the existence of an ϵ-differentially private graph algorithm $\mathcal{A}$ on user-level that, with probability at least $1 - \delta$, computes f with error less than $s/2$. After each update $\mathcal{A}$ has to output f, i.e., the sequence of outputs has the same length as the sequence of updates. Let O_b be the set of output sequences of $\mathcal{A}$ with additive error at most $s/2$ when processing $\mathcal{G}_b$ with G_1 as initial graph. Now we consider the family of output sets O_b for all $b \in \mathcal{B}$. For any two $b, b' \in \mathcal{B}$ with $b \neq b'$ consider the value of f applied to the two graph sequences $\mathcal{G}_b$ and $\mathcal{G}_{b'}$ and recall that there is at least one index in these two graph sequences where the f-value applied to the two graphs at this index differs by at least s. Thus, the event that $\mathcal{A}$ gives an answer with additive error less than $s/2$ on $\mathcal{G}_b$ (i.e., returns a sequence from O_b) and the event hat $\mathcal{A}$ gives an answer with additive error less than $s/2$ on $\mathcal{G}_{b'}$ (i.e., returns a sequence from $O_{b'}$) are pairwise disjoint. However, for any $b \in \mathcal{B}$ by ϵ-differential privacy, the probability that $\mathcal{A}$ outputs an sequence from O_b when run on $\mathcal{G}'$, i.e. $\Pr[\mathcal{A}(\mathcal{G}') \in O_b]$, is at least $e^{-(2\ell+1)\epsilon} \cdot \Pr[\mathcal{A}(\mathcal{G}_b) \in O_b] \geq e^{-4\epsilon\ell}(1 - \delta)$. Since $T > 2\ell \log(e^{4\epsilon\ell}/(1 - \delta))$, we have that $|\mathcal{B}| = 2^{T/2\ell} > e^{4\epsilon\ell}/(1 - \delta)$. It follows that $\Pr[\mathcal{A}(\mathcal{G}') \in \cup_{b\in\mathcal{B}} O_b] = \sum_{b\in\mathcal{B}} e^{-4\epsilon\ell} \Pr[\mathcal{A}(\mathcal{G}_b) \in O_b] \geq |\mathcal{B}| e^{-4\epsilon\ell}(1 - \delta) > 1$, which is a contradiction. □

Lemma 6.6.4. *Let $n > 0$ and let f be a graph function with spread (s, ℓ) that spares an edge e on G_1, G_2 according to Definition 6.6.1. Let u be the update that inserts*

e, and let $\mathcal{G}' := (G_1, G_1 \oplus u, G_1, G_1 \oplus u, \dots,)$ *of length* $\ell + 1 := |\tau(G_1, G_2)| + 1$. *Then, there exists a sequence of graph sequences* $(\mathcal{G}_0 := \mathrm{T}(G_1, G_2), \dots, \mathcal{G}_{2\ell+1} := \mathcal{G}')$ *and a sequence of edges* $(e_0, e_1, \dots, e_{2\ell+1})$ *so that, for any* $0 \leq i \leq 2\ell$, $\mathcal{G}_i$ *and* $\mathcal{G}_{i+1}$ *are event-level edge-adjacent on* e_i.

Proof. For $i \in \{0\} \cup [2\ell+1]$, we construct sequences $\mathcal{G}_i = (G_{i,1}, \dots, G_{i,\ell+1})$. Define $\mathcal{G}_0 := \mathrm{T}(G_1, G_2)$. We transform $\mathcal{G}_0$ by alternatingly applying u (i.e. we insert e) and u^{-1} (i.e. we delete e) to its graphs (in addition to the existing updates). For $i \in \{1, \dots, \ell\}$, we define

$$\mathcal{G}_i := (G_{i-1,1}, \dots, G_{i-1,i}, G_{i-1,i+1} \oplus u_i, G_{i-1,i+2}, \dots, G_{i-1,\ell+1}),$$

where

$$u_i = \begin{cases} u & \text{if } i \text{ odd} \\ u^{-1} & \text{otherwise.} \end{cases}$$

For example, for $i = 1$, $\mathcal{G}_1$ is the graph sequence where u is applied to the graph $G_{0,2}$. The remaining sequences $\mathcal{G}_{\ell+1}, \dots, \mathcal{G}_{2\ell+1}$ remove the updates made by applying $\tau(G_1, G_2)$. Let $(u'_1, \dots) := \tau(G_1, G_2)$. For $i \in \{\ell + 1, \dots, 2\ell + 1\}$, we define

$$\mathcal{G}_i := (G_{i-1,1}, \dots, G_{i-1,i}, G_{i-1,i+1} \oplus u_i'^{-1}, G_{i-1,i+2} \oplus u_i'^{-1}, \dots, G_{i-1,\ell} \oplus u_i'^{-1}).$$

In other words, in $\mathcal{G}_i$, the first i updates of $\mathrm{T}(G_1, G_2)$ are not present compared to $\mathcal{G}_\ell$. Since $\mathrm{T}(G_1, G_2)$ is a shortest sequence that transforms G_1 into G_2, an edge is either inserted or deleted at most once. Thus, reverting u'_i by executing $\oplus u_i'^{-1}$ for all graphs $G_{i-1,j}$ with $j \geq i + 1$ cannot invalidate the graph sequence. The claim follows because u'_i only affects a single edge. □

Fact 6.6.5. *Minimum spanning tree has spread* $(\Theta(nW), \Theta(n))$. *Minimum cut has spread* $(\Theta(nW), \Theta(n^2))$. *Maximal matching has spread* $(\Theta(n), \Theta(n))$. *Maximum cardinality matching has spread* $(\Theta(n), \Theta(n))$. *Maximum weight matching has spread* $(\Theta(nW), \Theta(n))$.

7 DYNAMIC DEMAND-AWARE LINK SCHEDULING FOR RECONFIGURABLE DATACENTERS

7.1 Introduction

The performance of many cloud-based applications critically depends on the underlying network, requiring high-throughput datacenter networks which provide extremely large bandwidth [127–129]. For example, in distributed machine learning and data mining applications that periodically require large data transfers, the network is increasingly becoming a bottleneck. High network throughput requirements are also introduced by today's trend of resource disaggregation in datacenters, where fast access to remote resources (e.g., GPUs or memory) is critical, and the trend to hardware-driven workloads such as distributed training [127].

The stringent throughput requirements of modern datacenter applications have led researchers to propose innovative datacenter network designs which rely on *reconfigurable topologies* [61–64, 70–73, 130, 131]: emerging optical technologies allow to enhance existing datacenter networks with reconfigurable optical matchings. In particular, optical circuit switches can provide direct connectivity between datacenter racks, in the form of one *matching* per optical circuit switch. This technology enables *demand-aware networks* [132]: the optical matchings, and hence the datacenter topology interconnecting the racks, are adjusted depending on the current communication traffic.

Such demand-aware topology optimizations are attractive as datacenter traffic typically features much spatial and temporal structure, and most transmitted bytes belong to a small number of so-called *elephant flows* [59, 60, 133]. Thus, throughput may be significantly increased by optimizing the topology towards the current traffic matrix and elephant flows. This can be achieved, for example, by providing direct connectivity between frequently communicating racks rather than serving traffic along multiple hops, which introduces a "bandwidth tax" [64, 134].

However, demand-aware topology optimizations are computationally expensive. In fact, state-of-the-art algorithms [65, 69, 70, 135–138] to compute optimized demand-aware switch matchings for a given demand matrix can

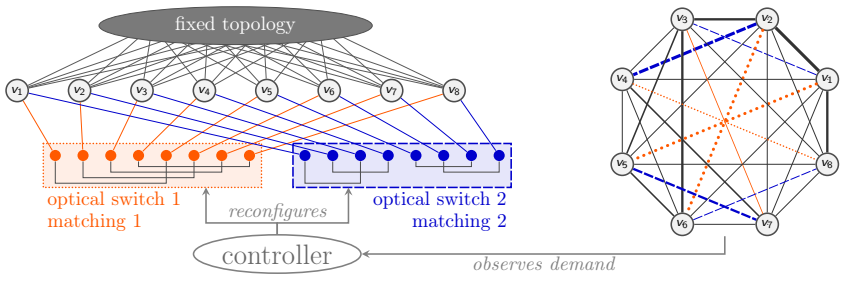

Figure 7.1.1: Two optical switches establishing direct connections between nodes $v_1, \ldots, v_8$, as instructed by the network controller, and the demand graph with the corresponding two matchings.

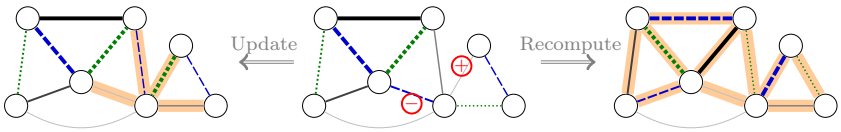

Figure 7.1.2: Updating vs. recomputing a solution from scratch for $k = 2$: The two matchings are visualized in dotted green and dashed blue, the line width of an edge corresponds to its weight. Shaded edges have been changed.
Center: Initial situation. Two updates arrive, one increases an edge weight ⊕, another decreases an edge weight ⊖. *Left:* The result after processing the updates. Only four edges are affected. *Right:* The potential result after a full recomputation from scratch. The whole network is reconsidered and also unaffected edges may have change, which can be inefficient.

have a running time that is *significantly higher* than the actual reconfiguration time provided by the state-of-the-art optical technologies (which is in the order of microseconds or even less [74, 139]). Thus, *the computation of the demand-aware topology can be the bottleneck for demand-aware datacenter networks.*

Formally, the problem of optimizing the topology of a demand-aware network is a novel variant of a matching problem consisting of "heavy" disjoint matchings [74] (see Figure 7.1.1): given k optical switches and a traffic demand (a.k.a. demand matrix) represented as a weighted graph where each node corresponds to a datacenter rack and weighted edges represent demands, compute k *(edge-)disjoint matchings of high weight.* The weight of these matchings hence corresponds to the amount of traffic which can be offloaded to the reconfigurable network, and hence to the throughput achieved by the datacenter network. In most existing reconfigurable datacenter architectures, including Helios [65], c-Through [69], or recently Google's Gemini [140], among many others [135–138], this topology optimization is performed by a centralized software controller. Computing such a set of matchings of maximum weight is NP-hard and cannot be approximated with arbitrary precision for all $k \geq 2$ [74, 75].

The goal of this work is to improve the running time for computing k disjoint matchings to allow datacenter networks to react to changes in the demand more quickly and improve throughput further. Our main idea is as follows: Since traffic exhibits much temporal locality, it can be inefficient to *recompute* the datacenter topology from scratch for each traffic matrix. Rather, we study *dynamic algorithms*, i.e., algorithms which *update* the topology incrementally, as a reaction to shifts in the demand matrix. Besides speeding up the computation, such dynamic algorithms may also require fewer changes in configurations to achieve a given throughput. The latter benefits ongoing flows as fewer of them will be interrupted.

We illustrate our motivation with a simple example network, see also Figure 7.1.2: Assume the communication demand, i.e. edge weight, of a node to two of its neighbors changes. A *dynamic* algorithm, which processes the updates itself, has the possibility to adjust the configuration *locally* and to leave most of the old solution untouched. By contrast, a *static* one, which does a full recomputation from scratch, needs to process the *entire* network and, as it is unaware of the old configuration, may additionally introduce unnecessary

changes. This can negatively affect both the running time of the algorithm and the number of changes to implement on the optical switches.

Thus, we study the following *dynamic weighted k-disjoint matching problem*: Given an undirected graph G with edge weights representing the current communication demands, process a sequence of *batch updates*, each consisting of a set of (edge) updates, where each (edge) update either inserts or deletes an edge or changes the weight of one edge. The main goal is to process each batch as quickly as possible and deliver an up-to-date configuration, i. e., k edge-disjoint matchings, after each batch such that the total weight is maximized. A secondary goal is to keep the *recourse* small, i. e., to minimize the number of changes to the matchings.

CONTRIBUTIONS. We develop and evaluate a diverse set of algorithmic techniques for the weighted k-disjoint matching problem arising in the context of reconfigurable datacenter networks. Based on the best static algorithms in [74], we design two *dynamic* algorithms, which process each update individually, three *batch-dynamic* algorithms, which process a batch of updates collectively, and two *hybrid* algorithms, which combine subroutines of static and dynamic approaches. Furthermore, we introduce a *universal speedup technique* that filters out insignificant updates, as well as a *universal post-processing routine* that ensures an at least $\frac{1}{2}$-approximation of the maximum weight and can also be run standalone. We compare these algorithms and their speedup and postprocessing versions in detail with respect to solution weight, running time, and recourse in theory and practice to the best static algorithms from [74].

MAIN EXPERIMENTAL RESULTS. Our extensive study on 39 real-world and 176 synthetic instances shows that our batch-dynamic and dynamic algorithms can beat the best static algorithm w.r.t. running time and recourse, while essentially retaining the solution weight. The combination of the post-processing and speedup technique proved to be particularly important for the solution weight of the dynamic algorithms. On instances where batches are small, their advantage over static algorithms is even more pronounced. Our two hybrid algorithms successfully combine the advantages of static and dynamic algorithms and are a good general-purpose choice.

We note that our approach is compatible with most existing reconfigurable datacenter architectures, including [65, 69, 135–138, 141, 142], which can hence directly benefit from these performance improvements.

7.2 Preliminaries

BASICS. We model the communication demand between peers as an undirected, weighted graph $G = (V, E, w)$ with node set V, edge set E, and use $n := |V|$, $m := |E|$. The function $w : E \to \mathbb{N}_0$ maps each edge to a weight, which corresponds to the respective communication demand. An edge e with $w(e) = 0$ is considered *absent*. The maximum edge weight is $W := \max_{e \in E} w(e)$. The weight of a set of edges S is $w[S] = \sum_{e \in S} w(e)$. Edges sharing an endpoint are *adjacent*. For two sets of vertices $X, Y \subseteq V$, $E(X, Y) = E(Y, X) := \{\{x, y\} \in E \mid x \in X, y \in Y\}$ is the set of all edges between X and Y. The *neighborhood*

$N(e)$ of an edge e is $N(e) := E(\{u,v\},V) \setminus \{e\}$. Adding a set $S \subseteq E$ as subscript restricts the neighborhood to $N_S(e) := N(e) \cap S$. The degree $\deg(v)$ of a vertex v is the number of edges incident to v, and $\Delta := \max_{v \in V} \deg(v)$ is the maximum degree.

A *matching* in G is a subset of E such that no two edges are adjacent. A *k-disjoint matching* $\mathcal{M}$ is a set of k matchings $(M_1, \ldots, M_k)$ such that $M_i \cap M_j = \emptyset$ for all $i, j \in [k]$, $i \neq j$. The *weight* of a k-disjoint matching $\mathcal{M}$ is $w(\mathcal{M}) := \sum_{i=1}^{k} w[M_i]$. We can alternatively view a k-disjoint matching as a *partial (edge) k-coloring*. Such a coloring $\mathcal{C} : E \to [k] \cup \{\perp\}$ assigns to each edge one of k colors or the symbol $\perp$, where for each pair of adjacent edges e, e' either $\mathcal{C}(e) \neq \mathcal{C}(e')$ or $\perp \in \{\mathcal{C}(e), \mathcal{C}(e')\}$. An edge e is *uncolored* if $\mathcal{C}(e) = \perp$ and *colored* otherwise. A color $c \in [k]$ is *free* at vertex v (resp. edge e) if there is no edge e' incident to v (resp. adjacent to e) such that $\mathcal{C}(e') = c$. The set of all neighboring edges of an edge e that have color c is denoted by $N_c(e)$. The *weight* of a coloring $\mathcal{C}$ is $w(\mathcal{C}) := \sum_{e \in E : \mathcal{C}(e) \neq \perp} w(e)$, i.e., the sum of the weights of all colored edges. We mainly use the coloring perspective and k always refers to the number of colors, i. e. disjoint matchings.

DYNAMIC SETTING. Our algorithms maintain an partial k-coloring under *edge updates*, where an (edge) update $\mathcal{U}(e, \delta)$ is a change in the weight of an edge e by some amount $\delta \neq 0$, i. e., $w(e) \leftarrow w(e) + \delta$. The weight may either increase or decrease. We consider a change from 0 to a positive weight to be an *insertion* and a change from a positive weight to 0 to be a *deletion*. The updates are presented to the algorithm in a fixed order, and no information about the updates themselves or the length of the update sequence is known to the algorithms beforehand. Additionally, the sequence of updates is partitioned into consecutive subsequences, each forming a *batch*. For each edge e if there are multiple updates of e in a batch, we add up the weight of the corresponding updates creating at most one update per edge per batch. The set of edges updated within a batch $\mathcal{B}$ is referred to as $E_{\mathcal{B}}$. We assume that after each batch the algorithms are required to (i) output the value of an up-to-date solution, and to (ii) be able to output the update to the solution itself (i. e., the differences in the coloring) in time linear in the *recourse*. The *recourse* corresponds to the total number of *changes* in the coloring over a batch, i. e., if $\mathcal{C}'$ and $\mathcal{C}$ are the edge colorings before and after a batch of updates $\mathcal{B}$, respectively, the recourse is $|\{e \in E \mid \mathcal{C}'(e) \neq \mathcal{C}(e)\}|$.

We consider four types of algorithms: *Dynamic* algorithms update their solution after every individual update; *batch-dynamic* algorithms process the entire batch at once and in self-chosen order; *static* algorithms recompute from scratch after each batch. *Hybrid* algorithms decide on the basis of the observed updates whether they update the solution like a dynamic algorithm or recompute from scratch like a static one.

RELATED WORK. The study of the k-disjoint matching problem is motivated by emerging datacenter networks which augment a static (fixed) topology, with a dynamic topology implemented with k optical (circuit) switches. Each optical switch provides a set of exclusive direct connections between top-of-rack switches that form a matching. For a recent survey of the field see Hall et al. [58].

Our work builds upon a recently suggested approach by Hanauer et al. [74], who showed that the static version of the problem is NP-hard and no FPTAS can exist. They evaluated a variety of algorithms in theory and practice.

The *matching problem* has been intensively studied in the *dynamic* setting, see [143] for a survey. For the *dynamic edge coloring* problem, only algorithms for the unweighted setting exist, which are not applicable to our setting.

Bienkowski et al. [144] studied a problem in the online setting that is similar to k-disjoint matchings, but can be solved to optimality in polynomial time.

7.3 Algorithms

Our algorithms are in part based on static algorithms described in [74], specifically `GreedyIt`, `NodeCentered`, and `kEC`. We will first review these algorithms briefly. We then present our dynamic, batch-dynamic, and hybrid algorithms, as well as a speedup technique and a postprocessing routine. Table 7.3.1 gives a compact overview over their asymptotic running time and recourse.

Below we use the two functions **`SwapIn`**(e, c) and **`SwapOut`**(e), where e is an edge and $c \in [k]$ is a color. `SwapIn`(e, c) colors e with c and uncolors e's neighbors $N_c(e)$ if $w(e) > w[N_c(e)]$, otherwise it does nothing. `SwapOut`(e) looks for up to two non-adjacent, uncolored edges e' and e'' in $N(e)$ such that (i) $\mathcal{C}(e)$ is free for both e' and e'' and (ii) their total weight w' is maximum over all such edge pairs. If and only if $w' > w(e)$, `SwapOut`(e) colors these edges with $\mathcal{C}(e)$ and uncolors e.

7.3.1 Static Algorithms

We use the best static algorithms from Hanauer et al. [74], which we refer to for more details. They are rerun from scratch after each batch of updates. As they are ignorant of previous colorings, they have a worst-case recourse of $O(m)$.

Table 7.3.1: Running times and recourse for a batch of b updates. m, n and Δ refer to the cardinalities *afterwards*.

Algorithm	Batch Update Time	Recourse
`GreedyIt`	$O(m \log m + km)$	$O(m)$
`NodeCentered`	$O(n \log n + m \log \Delta + km)$	$O(m)$
`kEC`	$O(m \log m + kn^2)$	$O(m)$
`dyn-greedy`(α, β)*	$O(b(k \cdot 2^\alpha + \beta^2))$	$O(b \cdot 2^\alpha)$
`dyn-kEC`*	$O(b \cdot n)$	$O(b \cdot n)$
`batch-greedy-l`	$O(\Delta b \log(\Delta b) + k\Delta^2 b)$	$O(b \cdot \Delta)$
`batch-greedy`	$O(\Delta b \log(\Delta b) + k\Delta b)$	$O(b \cdot \Delta)$
`batch-NC`	$O(b(\log b + \Delta \log \Delta + k\Delta))$	$O(b \cdot \Delta)$
`batch-2apx`	$O(m(k + \log m))$	$O(m)$
`hybrid-greedy`(α, β)	*see Lemma 7.3.5*	$O(m)$
`hybrid-kEC`	$O(m \log m + kn^2)$	$O(m)$

*for b calls to `AttemptColor` or `DecreaseWeight`

The **GreedyIt** algorithm is a "greedy" matching algorithm: For each color c, all uncolored edges are processed in non-increasing order of their weight and colored with c if admissible. At the end of the iteration for c, it optionally performs a **LocalSwaps** procedure, where SwapOut(e) is run on each edge of color c.

The **NodeCentered** algorithm is also greedy, but proceeds node by node. The nodes are processed in non-increasing order by their rating, which corresponds to the sum of the weights of the k heaviest incident edges. For each node, up to k incident edges are colored with any available color in order of non-increasing weight. To avoid an overly greedy coloring, the algorithm defers the coloring of edges with weight below $\theta \cdot W$, where θ is a threshold parameter, until after all nodes have been processed. We use $\theta = 0.2$, as suggested in [74].

The **kEC** algorithm is based on the edge coloring algorithm of Misra and Gries (MG) [81]. Hanauer et al. [74] adapted this algorithm to consider edge weights and to limit the number of colors to k, and considered different speedup techniques. The key observations that allow a modification of the algorithm without compromising its correctness are that MG (i) colors the edges in arbitrary order and (ii) never uncolors an already colored edge. We use kEC as suggested [74] with the flags CC and RL enabled. kEC tries to color the edges in non-increasing order according to their weight, using a procedure **kColorEdge**, which largely follows the routine in MG, as follows: To color an edge $e = \{u, v\}$, kEC first checks whether u and v both have at least one free color each, and leaves e uncolored otherwise. If a *common* free color exists, it is used for e (CC flag). Otherwise, following MG, kEC constructs a fan $\mathcal{F}_u$ around u. A *fan* is a maximal sequence $\mathcal{F}_u = (f_0 = v, f_1, \ldots, f_\ell)$ of distinct neighbors of u, where for all $1 \leq i \leq \ell$, if $\{u, f_i\}$ has color c_i, then color c_i is free on f_{i-1}. Note that $\mathcal{F}_u$ is not necessarily unique. Now we pick a color c that is free at u and a color d that is free at f_ℓ. In MG, d always exists, whereas in kEC, the number of colors is limited to k. Thus, if kEC does not find a free color at f_ℓ, kColorEdge has failed for u and kEC repeats kColorEdge symmetrically at v. If kColorEdge failed at both u and v, $\{u, v\}$ remains uncolored.

Otherwise, assume that f_ℓ in fan $\mathcal{F}_u$ has a free color d. We consider two cases. (1) If d is free at u, kEC *rotates* the full fan $\mathcal{F}_u$ (RL flag), i. e., for all $1 \leq i \leq \ell$, the edge $\{u, f_{i-1}\}$ is recolored with the color of $\{u, f_i\}$, and $\{u, f_\ell\}$ receives the color d. (2) Otherwise, as d was free at f_ℓ and the fan is maximal, there must be a neighbor f_j in $\mathcal{F}_u$ such that $\{u, f_j\}$ has color d. Like MG, kEC constructs a path in the graph of maximal length that starts with the edge $\{u, f_j\}$ and whose colors alternate between d and c. The colors d and c are then swapped along this path, which guarantees that d is free both at u and at at least one vertex in $\mathcal{F}_u$ [81]. kEC (and MG) now rotate $\mathcal{F}_u$ as in (1), but only up to the first vertex f_x where d is free, and then recolor $\{u, f_x\}$ with d.

7.3.2 Dynamic Algorithms

The algorithm **dyn-greedy** is a dynamic enhancement of GreedyIt and takes two parameters α and β. Each update is treated separately, where the subroutines AttemptColor and DecreaseWeight handle weight *increases* of *uncolored* edges and weight *decreases* of *colored* edges, respectively. Nothing is done on

weight increases of already colored edges or weight decreases of uncolored edges.

`AttemptColor`: If there is a free color on e, we color e with this color. Otherwise, we try to find a color c where `SwapIn`(e, c) is successful. If yes, we recursively call `AttemptColor` on the newly uncolored edges up to a recursion depth of α. To determine c, if $\beta \geq k$, c is the color of the edges adjacent to e that have minimum total weight. If $\beta < k$, we pick a subset of β colors uniformly at random from $[k]$ and just among these, we use the color where the edges adjacent to e have the minimum total weight.

`DecreaseWeight`: We first call a modified `SwapOut`(e) which only considers a random subset of β incident edges per end node. If this changed the coloring, the algorithm tries to recolor e using `AttemptColor` with $\alpha = 0$ and β as given. The procedure is deterministic for $\beta = \Delta$.

Lemma 7.3.1. `dyn-greedy` *processes an edge weight increase in time* $O(k \cdot 2^\alpha)$ *and a decrease in* $O(\beta^2 + k)$*. The recourse is* $O(2^\alpha)$*, respectively* $O(1)$*.*

Proof. In `AttemptColor`, finding a free color for edge $e = \{u, v\}$ takes time $O(k)$. If this fails, determining c takes $O(\beta)$, as we can obtain the mates of u and v in each matching in constant time. The total number of recursive calls is $2^{\alpha+1} - 1$, which yields a worst-case update time of $O(k \cdot 2^\alpha)$.

In `DecreaseWeight` the call to `SwapOut`(e) takes $O(\beta^2)$ time to find the pair with maximum weight among the sampled edges. The call to `AttemptColor` takes time $O(k)$, since the parameter α is set to zero here. In total, the worst-case update time for `DecreaseWeight` is $O(\beta^2 + k)$.

For the recourse observe that in `AttemptColor` without recursion at most two edges are uncolored and one edge is colored. Thus, the number of changes to the coloring is $O(2^\alpha)$, i.e., the number of recursive calls. `DecreaseWeight` colors at most two edges and uncolors at most one edge in addition to the changes by a call to `AttemptColor` with $\alpha = 0$, so the recourse is constant. □

Picking β edges uniformly at random does not affect the asymptotic running time of `AttemptColor`. However, in practice we observe a speedup when setting $\beta \ll k$.

The dynamic k-edge coloring algorithm **`dyn-kEC`** uses the `kColorEdge` procedure of the static `kEC` algorithm to color edges. Similar to `dyn-greedy`, it tries to color a previously uncolored edge if its weight increases, and the heaviest uncolored edges adjacent to an updated, colored edge in case of a weight decrease. Weight increases of colored edges and weight decreases of uncolored edges are again ignored.

Before calling `kColorEdge` for an edge $\{u, v\}$, the algorithm ensures that both u and v have at least one free color each (cf. Section 7.3.1): Let $E_u = \emptyset$ if u has a free color and let E_u otherwise be the singleton containing the colored edge incident to u with the smallest weight. E_v is defined likewise for v. If $E_u \cup E_v = \emptyset$, `kColorEdge` is called immediately. Otherwise, if $w[E_u \cup E_v] < w(\{u, v\})$, the edges in $E_u \cup E_v$ are uncolored before `kColorEdge` is called. The uncoloring is undone if `kColorEdge` failed to color $\{u, v\}$, otherwise, the algorithm just tries to recolor an edge in $E_u \cup E_v$ with a free color. No action is taken if $w[E_u \cup E_v] \geq w(\{u, v\})$.

Lemma 7.3.2. *Algorithm* `dyn-kEC` *processes each update in time* $O(n)$ *and has a recourse of* $O(n)$*.*

Proof. Finding E_u and E_v can be done in time $O(k)$ by iterating over the incident colored edges. The call to `kColorEdge` takes time $O(n)$: constructing a fan and the alternating path as well as rotating the fan and inverting the path can be done in time $O(n)$. The running time follows from $k \leq n$. A call to `kColorEdge` on edge $\{u, v\}$ changes the coloring of up to Δ edges when rotating the fan and $O(n)$ edges when swapping colors along the path. Additional changes to the coloring only concern two edges incident to $\{u, v\}$. Thus, the total recourse is $O(n)$. □

7.3.3 Batch-Dynamic Algorithms

The algorithm **`batch-greedy`** is a batch-dynamic adaptation of the static `GreedyIt` algorithm. For each batch, it processes only those edges that are present and either updated or adjacent to at least one updated or deleted edge. The latter ensures that an edge with large weight increase can be colored at the expense of an unchanged neighboring edge. The edges to be processed are first uncolored and then treated like the set of all edges in `GreedyIt`, including the optional `LocalSwaps` procedure. We refer to the algorithm by `batch-greedy-l` with `LocalSwaps` and by `batch-greedy` without.

Lemma 7.3.3. *For a batch of size b,* `batch-greedy` *and* `batch-greedy-l` *run in time $O(b\Delta \log(b\Delta) + kb\Delta)$ and $O(b\Delta \log(b\Delta) + kb\Delta^2)$, respectively. The recourse is $O(b\Delta)$.*

Proof. There are b updated edges and up to 2Δ edges adjacent to each updated edge. This yields a total of $O(b\Delta)$ edges to be processed. The running time and recourse follow from the running time and recourse for `GreedyIt` (see Table 7.3.1). □

The batch node centered algorithm **`batch-NC`** is an adaptation of the static `NodeCentered` algorithm and, similar to `batch-greedy`, processes only updated edges and their adjacent edges. For every node that is incident to an updated edge, its rating is computed as in `NodeCentered` and *all* its incident edges are uncolored. These nodes are then ranked and processed as in `NodeCentered`, using the same threshold parameter $\theta = 0.2$.

Lemma 7.3.4. *For a batch of size b,* `batch-NC` *runs in time $O(b(\log b + \Delta \log \Delta + k\Delta))$. The recourse is $O(b\Delta)$.*

Proof. There are at most $2b$ nodes incident to updated edges and each of these nodes has up to $O(\Delta)$ incident edges. Thus, the algorithm processes $O(b)$ nodes and $O(b\Delta)$ edges. The running time and recourse then follows from the running time and recourse for `NodeCentered` (see Table 7.3.1). □

7.3.4 Hybrid Algorithms

If a large fraction of the edges in the graph is updated, it can be better to recompute a coloring from scratch instead of processing each update in the batch individually. To this end, we combine the static algorithm with the best time-for-weight tradeoff [74], `kEC`, with the update procedures of our two dynamic algorithms, `dyn-greedy` and `dyn-kEC`, and thus obtain a **`hybrid-greedy`** and a **`hybrid-kEC`** algorithm.

The hybrid algorithms choose between the static and dynamic algorithm with the goal to minimize the running time. For the decision, they use a simple heuristic that can be computed quickly and relates the size of the batch b to n: if $b < n$, the dynamic algorithm should be faster; otherwise, we expect the dynamic algorithm to do much more work than the static one, in part due to bookkeeping, so the static algorithm should be faster. To run the procedures of the dynamic algorithms at the time the update is observed, the decision needs to be made *before* the next batch arrives. We hence use the batch size b' of the *previous* batch as an estimate for the current batch.

The following lemma states the running time and recourse, which follow from those for `kEC`, `dyn-greedy`, and `dyn-kEC` (see Table 7.3.1).

Lemma 7.3.5. `hybrid-greedy`, *parameterized by* α *and* β, *and* `hybrid-kEC` *process a batch of updates in* $O(n \cdot (k \cdot 2^{\alpha} + \beta^2) + m \log m + kn^2)$ *and* $O(m \log m + kn^2)$ *time, respectively. The recourse is* $O(n)$.

7.3.5 Reducing Work by Filtering Updates

Minor changes in the weight of an edge do not have a large effect on the overall solution weight. This holds particularly in large graphs, where the contribution of an individual edge to the solution weight is small by comparison. We filter such updates with the goal of improving the running time without degrading the weight of the solutions.

Our strategy is parameterized by a threshold $t \geq 1$. Let $\mathcal{U}(e, \delta)$ be an update that changes the weight of e from w to $w' = w + \delta$. With the filtering enabled, updates with $w, w' \neq 0$ and $\frac{w'}{w} \in [1/t, t]$ are not processed by the dynamic algorithms. Insertions and deletions are never discarded. We suffix an algorithm with `-f` if filtering is used.

7.3.6 Post-Processing and Approximation Guarantees

To improve the weight of a given coloring, we consider a post-processing routine that performs local optimization in the neighborhood of uncolored edges. It ensures that for every color i, an uncolored edge has either one or two edges with color i in its neighborhood that are (in sum) at least as heavy as the uncolored edge. It establishes the following invariant:

Invariant 7.3.6. *For every uncolored edge* $e \in E$ *and each color* $c \in [k]$, $w[N_c(e)] \geq w(e)$.

The post-processing algorithm places all uncolored edges in a priority queue, ordered by decreasing edge weight. For each edge e that is removed from the queue, it (1) colors it with a free color c if one exists, or (2) finds a color c' for which the invariant is violated and performs a `SwapIn`(e, c'), which must succeed due to the violation. The newly uncolored edges are then pushed onto the queue for an invariant check. Due to the violated invariant, they must both be strictly lighter than e. Thus, each edge can be enqueued and dequeued at most once and the procedure terminates after at most m iterations. (3) If the invariant is satisfied for all colors, e remains uncolored.

The post-processing can be combined with any of the aforementioned algorithms by applying it to the coloring after a batch is complete. We suffix an algorithm with `-p` if post-processing is used.

Lemma 7.3.7. *The post-processing algorithm outputs at least a $\frac{1}{2}$-approximation for the weighted k-disjoint matching problem.*

Proof. Let $\mathcal{M}^* = (M_1^*, M_2^*, \ldots, M_k^*)$ be an optimal solution, let $\mathcal{M} = (M_1, M_2, \ldots, M_k)$ be a solution computed by the algorithm and consider an edge $e \in \mathcal{M}^* \setminus \mathcal{M}$. Let $i \in [k]$ such that $e \in M_i^*$. As $e \notin \mathcal{M}$, there is either adjacent edge $e' \in N(e) \cap M_i$ with $w(e') \geq w(e)$ or two adjacent edges $e', e'' \in N(e) \cap M_i$ with $w(e') + w(e'') \geq w(e)$ by Invariant 7.3.6. We charge e' a cost of $w(e)$ in the first case and e' and e'' together a cost of $w(e)$ in the second case, such that the respective cost shares do not exceed $w(e')$ and $w(e'')$.

Conversely, consider an edge $e \in \mathcal{M} \setminus \mathcal{M}^*$ and let $j \in [k]$ such that $e \in M_j$. Then, there can be at most two edges $e', e'' \in N(e)$ that are contained in M_j^*, but not in $\mathcal{M}$, so e can be charged at most a cost of $2w(e)$.

Hence, $w(\mathcal{M}^*) = w(\mathcal{M}^* \cap \mathcal{M}) + w(\mathcal{M}^* \setminus \mathcal{M}) \leq w(\mathcal{M}^* \cap \mathcal{M}) + 2w(\mathcal{M} \setminus \mathcal{M}^*) \leq 2w(\mathcal{M})$. □

The post-processing can also be run on its own as a batch-dynamic algorithm, which we refer to as **batch-2apx**. This algorithm collects those edges for which the invariant may have been violated by an update. At the end of a batch, it processes the collected edges as described above.

Lemma 7.3.8. *The post-processing algorithm and* batch-2apx *run in time $O(m(k + t_+ + t_-))$, where t_+ and t_- are the times for enqueuing and dequeuing an edge. The recourse is $O(m)$.*

Proof. The algorithm first enqueues $O(m)$ uncolored edges in a priority queue Q. For each edge e that is dequeued it takes $O(k)$ time to find a free color or a color for which Invariant 7.3.6 is violated. If the invariant is restored, then up to two previously colored edges are enqueued in Q, which are strictly lighter than e. It follows that each edge is enqueued and dequeued at most once. Thus, the total time is $O(m(k + t_+ + t_-))$. Only edges that were in Q at some point during the algorithm change their color and thus the recourse is $O(m)$.

□

7.4 Experiments

We evaluate the running time, solution weight, and recourse of the algorithms from Section 7.3 in practice on a large and diverse set of instances for $k \in \{2, 4, 8, 16, 32\}$.

7.4.1 Instances

To perform a thorough analysis we include both *real-world* as well as large *synthetic* instances. All real-world instances were generated from measured or simulated network traffic on real-world networks. Synthetic instances are random dynamic weighted graphs generated according to different models.

REAL-WORLD INSTANCES. We obtained dynamic instances from three real-world data collections that have been used to analyze network communication before [74, 144].

Table 7.4.1: Real-world and split instances: max. #edges M at any point in time, max. batch size B, max. #batches b, max. #nodes n, percentage of insertions I, deletions D and weight changes C among updates, and the number of instances in each data set.

data set	M	B	b	n	I	D	C	#
FB	66423	66421	1399	367	6 %	4 %	90 %	9
hpc	9330	16755	5000	1024	36 %	36 %	28 %	12
pfab	1390	2588	4999	144	33 %	33 %	33 %	9
FB/s	62799	64777	27980	367	48 %	47 %	5 %	26

The facebook (FB) [133] data sets consist of IP packet information from Database, WebService and Hadoop clusters. Each packet is associated with a Unix timestamp, the packet size, and the source and destination server. We construct our instances as follows: (1) We group x timestamps into one batch of updates. (2) For each source-destination pair (a, b), we sum the size of all packets occurring in this batch, which defines the new weight of the edge (a, b) in the graph. This may result in an edge insertion or deletion if the old or new weight, respectively, is zero. From each of the three clusters we obtain three instances by choosing $x \in \{60, 1800, 3600\}$.

The three pfab [60, 145] and four hpc [60] data sets consist of source-destination pairs, each of which is associated with a unique sequence number. We proceed similarly to the FB data set, with the sequence number as timestamp and the number of packets per batch as the packet size and edge weight. We obtain three instances per data set by choosing the group size $x \in \{10, 100, 1000\}$.

SPLIT INSTANCES. The FB instances are based on traces with a temporal resolution of one second. To simulate instances with higher resolution and more frequent reconfigurations of the optical network as well as to study the influence of edge weights, we generate a set of *split instances* as follows: We distribute the weight of an edge in a batch over multiple "sub-batches" such that its new weight never exceeds a threshold $z \in \{10^5, 10^6\}$ and set the number of sub-batches that are created from each original batch to $y \in \{5, 10, 15, 20\}$. For an original batch $\mathcal{B}$, we create exactly y sub-batches $\mathcal{B}_1, \ldots, \mathcal{B}_y$ and, for each edge that is updated to weight w in $\mathcal{B}$, we randomly select $\lceil \frac{w}{z} \rceil$ consecutive sub-batches from $\mathcal{B}_1, \ldots, \mathcal{B}_y$ and create the corresponding update operations. For example, for $y = 5$, and $z = 10^5$, if there is an edge e that is updated to $w = 215342$ in $\mathcal{B}$, e will have weight z during two sub-batches and the remaining weight $r = 15342$ in a third. If the three randomly chosen consecutive sub-batches are $\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$, we create update operations that set e's weight to z in $\mathcal{B}_2$, reduce it to r in $\mathcal{B}_4$, and to weight 0 in $\mathcal{B}_5$ (*zeroing*), which means to delete e. Note that e keeps its weight of z during $\mathcal{B}_3$, so no update is required. The zeroing is omitted if the last sub-batch is among the randomly chosen ones and e's weight is updated in the first sub-batch $\mathcal{B}'_1$ of the following (original) batch $\mathcal{B}'$. Thus, we create at most three updates for each update in the original instance. As a split instance has y times as many batches as the one it was created from, the batch sizes in the split instances are decreased. Smaller values for z increase the number of sub-batches $\lceil \frac{w}{z} \rceil$ that need to be selected, which reduces the number of zeroings and further decreases the batch sizes. For each combination of s and l and each instance in FB, we create a split

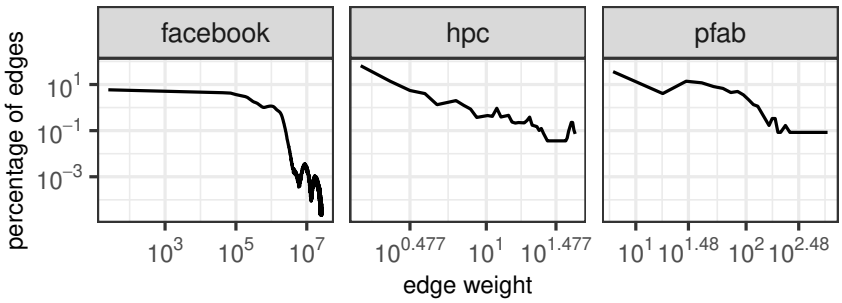

Figure 7.4.1: Weight distributions of the four data sets on a log-log scale.

instance, but only if its maximum edge weight is at most $y \cdot z$. The resulting data set `FB/s` contains 26 instances.

Table 7.4.1 summarizes the properties of the real-world and split instances. On average over the batches, 47 % of the updates in `FB`, `hpc`, and `pfab` are weight changes, 28 % are insertions, 25 % are deletions and 130 %[a] of the edges are updated. Splitting the batches and distributing the updates over sub-batches in `FB/s` greatly increases the relative number of insertions and deletions. Figure 7.4.1 shows the weight distributions.

SYNTHETIC INSTANCES. To diversify our test set and to further study the algorithms' behavior on large instances, we generated synthetic dynamic graphs from RMAT [146] instances, which have also been used to evaluate the static algorithms [74]. Following [74], the initiator matrices for the RMAT generator are $(0.55, 0.15, 0.15, 0.15)$ (`rmat_b`), $(0.45, 0.15, 0.15, 0.25)$ (`rmat_g`), and $(0.25, 0.25, 0.25, 0.25)$ (`rmat_er`), with a number of nodes $n = 2^x$ and $14 \leq x \leq 18$. Whereas each edge is equally likely for the `rmat_er` (Erdős-Rényi) graphs, `rmat_b` and `rmat_g` instances have skewed normal degree distributions, small-world properties, and larger clustering coefficients [147]. The weights follow an exponential distribution with values between 1 and 500000. We create the dynamic instances as follows: In the first batch, all edges of the static graph are inserted. In every subsequent batch, a fraction $f \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ of edges is updated. If the edge currently has positive weight, its weight is set to zero with probability $p \in \{0.1, 0.3\}$, i.e., the edge is deleted. Otherwise we assign a new weight uniformly at random from the weights in the original instance. We obtain 150 instances altogether, which we group by graph size n. Every group contains 30 instances ($3 \times 5 \times 2$), each of which has 30 batches of updates.

On average over all synthetic instances, a batch consists of 17 % insertions, 16 % deletions and 67 % weight changes.

7.4.2 Setup and Methodology

We implemented the algorithms in C++17 on top of the Algora library[b]. We record the running time and weight of the solution for each batch of updates. When applying a batch, the Algora framework notifies the algorithms of every update individually. The dynamic algorithms `dyn-greedy` and `dyn-kEC`

a W.r.t. the number of edges *after* the batch. Deletions can cause values above 100 %.
b `https://libalgora.gitlab.io`

immediately update the disjoint matching; the batch-dynamic algorithms `batch-greedy`, `batch-NC`, and `batch-2apx` use these notifications to prepare the batch, i.e., to build up the sets of (nodes incident to) updated edges.

We obtain the number of changes to the coloring over a batch by comparing the coloring before and after the batch. This is done by explicitly storing the state of the coloring before the batch and counting the changes in a separate run, where no running times are measured.

We run our experiments on a machine (`A`) with two Intel Xeon Gold 6130 CPUs (16 cores each) and 256 GB of main memory, and a machine (`B`) with two Intel Xeon E5-2643 CPUs and 2×750 GB of main memory. Machine `A` was used for all experiments on real-world and synthetic instances, whereas experiments for the split instances were run on machine `B`. For each experiment, the process was pinned to a NUMA node and its local memory. We focus on the time to obtain a new configuration, i. e., the running time of our algorithms, which are designed to be run on a central server.

To obtain reliable running times, we repeated each experiment three times. For the analysis, we take the median time over each batch for the deterministic algorithms and the arithmetic mean for the randomized algorithms, using different seeds. For the *average per-update time* $\bar{\tau}_I(\mathcal{A})$ of an algorithm $\mathcal{A}$ on an instance I, we divide the time per batch by the batch size and take the arithmetic mean over all batches. The *average solution weight* $\bar{\sigma}_I(\mathcal{A})$ is the arithmetic mean over all batches. In case of deterministic algorithms, the *average recourse* $\bar{o}_I(\mathcal{A})$ is defined analogously, whereas for randomized algorithms, we first take the mean recourse per batch over all repetitions and then the mean over these means.

We compare the algorithms relative to each other w.r.t. speedup, solution weight, and recourse. When comparing algorithm $\mathcal{A}$ to a reference algorithm $\mathcal{R}$ for an instance I, we obtain (1) the *speedup* as $\frac{\bar{\tau}_I(\mathcal{R})}{\bar{\tau}_I(\mathcal{A})}$, (2) the *relative solution weight* as $\frac{\bar{\sigma}_I(\mathcal{A})}{\bar{\sigma}_I(\mathcal{R})}$, (3) the *relative recourse* as $\frac{\bar{o}_I(\mathcal{A})}{\bar{o}_I(\mathcal{R})}$. To average over a data set, we always use the geometric mean.

7.4.3 Results

We analyze our algorithms first in smaller groups on the set of real-world instances, then consider their performance in terms of running time, solution weight, and recourse on the split and synthetic instances.

FILTERING SMALL UPDATES. The update filtering strategy described in Section 7.3.5 aims at ignoring small updates that are not expected to lead to large changes in the coloring. A preliminary parameter study demonstrated that the decrease in running time due to filtering roughly matched the decrease in solution weight. Combined with post-processing, the speedup was retained, but the solution weight was not impaired. Hence, we present only results for filtering together with post-processing.

PARAMETERS FOR `dyn-greedy`. Preliminary results showed that increasing the recursion-depth parameter α beyond $\alpha = 1$ yields no significant improvement in the solution weight, but increases the running time. Similarly, when comparing the randomized version of `dyn-greedy` with $\beta < \max\{k, \Delta\}$ to

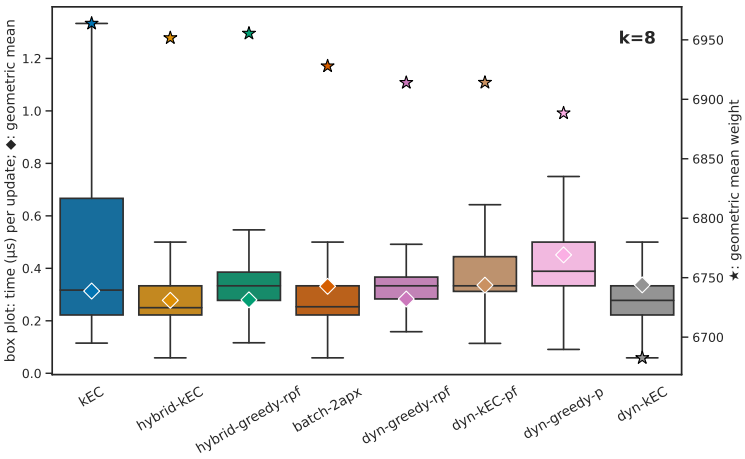

Figure 7.4.2: Average per-update times (left axis, boxes and ⧫) and weight (right axis, ★) for $k = 8$ on all real-world instances.

the deterministic version with $\beta = \max\{k, \Delta\}$, we observed that setting $\beta = 1$ yields the best trade-off between running time and solution weight for the randomized algorithm. In the following we thus consider only two versions: a randomized one with $\beta = 1$, which is identified by the suffix `-r`, and a deterministic one without this suffix.

Real-World Instances

Figure 7.4.2 provides an overview over per-update times and solution weights for selected algorithms.

BATCH-DYNAMIC AND STATIC ALGORITHMS. We evaluate the performance of these algorithms relative to each other. Our results *confirm* the earlier study for the static setting [74] and show that `kEC` is on average the *fastest static* algorithm. The same applies to `batch-greedy`, `batch-greedy-1`, and `batch-NC`. For all instances and all k, the speedup of `NodeCentered` and `GreedyIt-1` is between 0.2 and 1.5. However, for each value of k, the average speedup across all instances does not exceed 0.9. `GreedyIt` is faster than `kEC` only on the `pfab` data set with speedups up to 1.3, but takes more than twice as long as `kEC` on the other data sets. Only `batch-2apx` achieves significant improvements over `kEC`, with the average speedup over all `pfab` instances ranging from 1.3 to 1.7. On average across all instances, however, `kEC` *remains faster than all batch-dynamic algorithms*.

In terms of solution weight, the algorithms differ by no more than 2.5 % from the solution weight of `kEC`. We conclude that among the static and batch-dynamic algorithms, `kEC` *performs best* and that `batch-2apx` is the only batch-dynamic algorithm that is *competitive* on some data sets. *In the following, we hence use* `kEC` *as reference for comparisons.*

DYNAMIC ALGORITHMS. We compare the algorithm `dyn-greedy` and the randomized algorithm `dyn-greedy-r` to the respective variants with both filtering and post-processing, `dyn-greedy-pf` and `dyn-greedy-rpf`. The latter two achieve equal solution weight, which is in all cases at least 98 % of the weight achieved by `kEC`. `dyn-greedy-pf` is always slower than `dyn-greedy-rpf`,

which achieves speedups between 0.6 and 3.3 relative to kEC across all k and instances. The close performance in solution weight of dyn-greedy-pf and dyn-greedy-rpf suggests that the *post-processing contributes significantly to the solution weight* and can compensate for low-weight solutions produced by the base algorithm.

The randomized dyn-greedy-r cannot match kEC in terms of solution weight. For each k, the average solution weight across all instances in FB is below 89 % relative to kEC. Also dyn-greedy does not provide an advantage over kEC: it is either significantly slower or its solutions are more than 10 % worse. Among the dyn-greedy variants, dyn-greedy-rpf *thus offers the best trade-off* between solution weight and running time.

The dynamic version of kEC, dyn-kEC, only becomes competitive with kEC when combined with filtering and post-processing. Without either its solutions are about 10 % worse than that of kEC on average for each k and it takes up to three times as long as kEC, except for $k \leq 4$ on FB, where the speedup is up to 1.7. Enabling post-processing (dyn-kEC-p) improves the solution weight slightly, but makes it even slower. The combination with filtering (dyn-kEC-pf) again compensates the loss in running time: dyn-kEC-pf is faster than dyn-kEC overall, but still slower than kEC, however by only 4 % on average. Its solution weight is comparable to dyn-greedy-rpf (cf. Figure 7.4.2).

HYBRID ALGORITHMS. kEC and hybrid-kEC perform similar in terms of solution weight, differing in no more than 2 % on any instance. hybrid-kEC also tends to be at least as fast as kEC: On average over all k and all instances, it is equally fast on FB, 40 % faster on pfab and 8 % faster on hpc. Both algorithms do not benefit from post-processing, which increases the running time but not the solution weight.

Hybridizing kEC and dyn-greedy-rpf yields hybrid-greedy-rpf, which produces solutions that are for each k on average no more than 1 % worse. It is also faster than kEC, with average speedups up to 40 %.

SUMMARY. kEC is clearly the best static algorithm on the real-world instances, and also superior to all batch-dynamic algorithms except batch-2apx. Among the dynamic algorithms, dyn-greedy-rpf provides the best performance overall, with large speedups and solutions close in weight to those of kEC. hybrid-greedy-rpf and hybrid-kEC are also faster than kEC and have comparable solution weight.

Split Instances

On the FB/s instances, we see a similar, though not identical picture in comparison to the real-world instances (cf. Figure 7.4.3). One notable difference is that due to the limited maximum edge weight, *filtering remains essentially without effect*. As the number of updates per sub-batch decreases as the number of sub-batches increases or for a smaller weight limit, the speedup achieved by the dynamic algorithms relative to kEC improves in these cases.

The speedups are more pronounced on average for small k. The *fastest algorithms* across all values of k are dyn-greedy-rp and batch-2apx, both with a mean speedup across all k of 2.5 and with a maximum speedup over kEC on

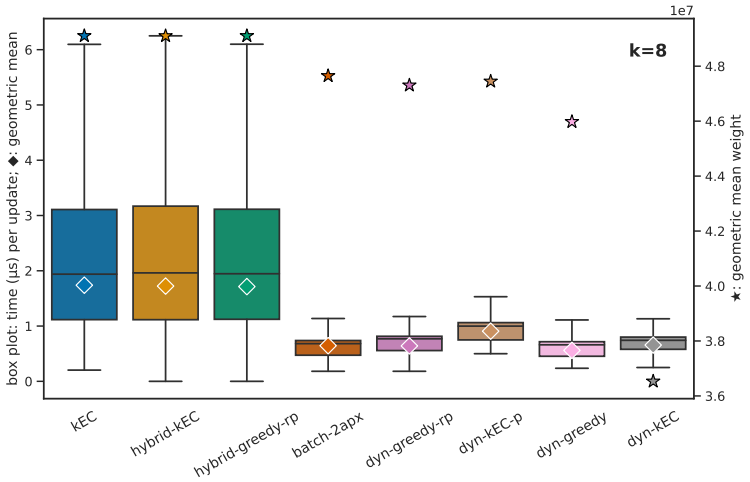

Figure 7.4.3: Average per-update times (left axis, boxes and ◆) and weight (right axis, ★) for $k = 8$ on all split instances.

an instance of 5.1 and 6.0, respectively. The hybrid algorithms show a similar performance as kEC with a speedup between 0.85 and 1.2.

All algorithms perform similar w.r.t. solution weight, only dyn-greedy-r and dyn-kEC (both without post-processing) fall significantly behind the others with a weight relative to kEC in the ranges 0.7–0.9 and 0.5–0.9, respectively.

Synthetic Instances

kEC is the fastest static algorithm on the synthetic instances. It is also faster than the greedy and node-centered batch-dynamic algorithms batch-greedy, batch-greedy-1, and batch-NC. batch-2apx is faster than kEC if at most 10 % of the edges are updated, and for $k = 2$ even if up to half of the edges are updated. The average speedups of batch-2apx over kEC for each k are between 4 % and 2.9.

Figure 7.4.4 shows the speedup of (batch-)dynamic and hybrid algorithms over kEC. The dynamic algorithms are on average faster than kEC if 10 %

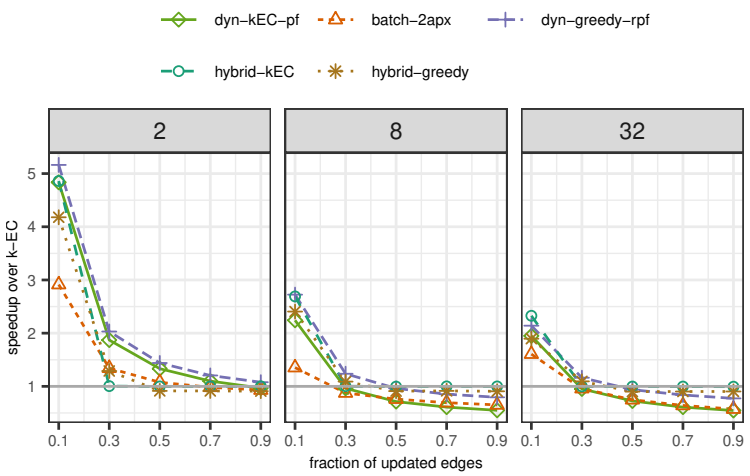

Figure 7.4.4: Speedup of (batch-)dynamic algorithms over kEC for $k = 2, 8, 32$ on the synthetic instances, where fraction of updated edges = #updates / m.

of the edges are updated in a batch. For $k = 2$, the dynamic algorithms dyn-greedy-rpf and dyn-kEC-pf achieve a maximum speedup of 5.2 and 4.0, respectively, even if 70 % of the edges are updated. hybrid-kEC takes on average the same time as kEC unless only 10 % of edges are updated. Here, it is significantly faster as it uses the dynamic routine to update the solution. Similarly, hybrid-greedy-rpf is faster if $k = 2$ and 10 % of edges are updated, but slower if at least half of the edges are updated.

As k increases, the solution weights generally improve relative to kEC and, e. g. for $k = 8$, they are already within 6 % for all algorithms. The worst case arises for $k = 2$ and we discuss it next in more detail: batch-2apx produces solutions that differ from those of kEC by no more than 3 % on average. Both hybrid-kEC and hybrid-greedy-rpf match the solution weight of kEC on instances where they mainly employ the static algorithm. However, when only 10 % of the edges are updated the solution weight drops off by 10 % relative to kEC. This matches the performance of their dynamic counterparts: dyn-kEC-pf and dyn-greedy-rpf yield solutions that are at least 10 % lighter than those of kEC when only 10 % of edges are updated. With increasing batch size the solution weight improves, but for dyn-kEC-pf does not get within less than 8 % of kEC for $k = 2$.

In conclusion, while the dynamic algorithms are generally faster than kEC on instances with smaller batch size, the speedup comes at the cost of a somewhat decreased solution weight. For $k = 2$, this decrease can be quite significant, but it improves for larger k. batch-2apx produces high-weight solutions even for $k = 2$, which are within 3 % of kEC's. Thus, batch-2apx *is the recommended choice for small k*, whereas hybrid-kEC *and* hybrid-greedy-rpf *are faster for larger k* without sacrificing solution weight.

Recourse

We compare the best algorithms from the previous sections. As the static algorithms differ in recourse by no more than 2 %, we use kEC as reference.

REAL-WORLD INSTANCES. dyn-greedy-rpf and batch-2apx have similar recourse on FB and hpc, which is between 56 % and 69 % of kEC's. On pfab, batch-2apx is slightly better than dyn-greedy-rpf and clearly better than kEC. dyn-kEC-pf performs similarly well as dyn-greedy-rpf on hpc and pfab, and even better on FB with a recourse of at most 39 % relative to kEC.

The recourse of the hybrid algorithms tends to be between kEC and their dynamic subroutine. hybrid-kEC performs equal to kEC on FB, better on hpc, but worse on pfab. hybrid-greedy-rpf has lower recourse than kEC on all instances and matches that of dyn-greedy-rpf on pfab.

SYNTHETIC INSTANCES. We expect the recourse of the dynamic and hybrid algorithms to be less than that of kEC for small batch sizes. The results on the synthetic instances confirm this. dyn-greedy-rpf, dyn-kEC-pf, and batch-2apx show similar results in terms of recourse: for $k = 2$ the recourse ranges between 50 % and 95 % relative to kEC and between 22 % and 36 % for $k = 32$. As expected, the recourse increases with the batch size. Surprisingly on first sight, the recourse relative to kEC decreases as k increases. However, for larger k more edges are colored and, thus, a larger percentage of edges

keep their color during an update. The hybrid algorithms hybrid-kEC and hybrid-greedy-rpf have the same recourse as kEC if at least half of the edges are updated in a batch. If only 10 % of edges are updated, the recourse is similar to their dynamic counterparts. This behavior reflects the above discussion of running times and solution weights.

SPLIT INSTANCES. We observe a similar reduction in recourse for small batches on the split instances FB/s. The hybrid algorithms hybrid-kEC and hybrid-greedy-rp showed the same performance with respect to running time and solution weight, and they also produce the same recourse as kEC, regardless of batch size. The dynamic algorithms' recourse is on average at least 20 % lower than that of kEC. Post-processing has a significant impact on recourse: for dyn-greedy-rp has average recourse up to twice that of dyn-greedy-r. A similar trend can be seen for dyn-kEC-p compared to dyn-kEC, but the effect is not as pronounced. Filtering, on the other hand, has no significant impact on the recourse, which also matches our observations w.r.t. running time and solution weight. In general, the recourse relative to kEC increases for dynamic algorithms as the batches become larger. For $k = 2$ dyn-greedy-rp produces an average recourse between 51 % and 73 % and 44 % to 74 % for $k = 32$. dyn-kEC-p, dyn-greedy-rp and batch-2apx have a recourse roughly equal to dyn-greedy-rp.

SUMMARY. The (batch-) dynamic and hybrid algorithms generally have a smaller recourse than the static algorithms, which leads to fewer changes in the network configuration. On instances where the hybrid algorithms mainly use the static kEC, they have no advantage in terms of recourse.

Summary of Experiments

Our results show that dynamic algorithms are particularly good on instances with smaller batches, while static algorithms perform well on large batches. This suggests the following recommendations: (1) When dealing with small batches and small numbers of matchings, i.e., frequent reconfigurations of the network and a small number of switches, batch-2apx exhibits the best trade-off between running time and solution weight. (2) For a larger number of matchings, the dynamic algorithms dyn-greedy-rpf and dyn-kEC-pf provide faster running times and high-weight solutions. (3) When dealing with large batches, e.g. if the intervals between reconfigurations are longer, the hybrid algorithms hybrid-kEC and hybrid-greedy-rpf provide the best performance if running time, solution weight, and recourse are considered. They are also well-suited for other situations and thus are a very good general-purpose choice. (4) If the solution weight has the top priority and sacrifices w.r.t. running time and recourse are acceptable, the best choice is kEC. Generally, the dynamic and hybrid algorithms have a lower recourse than the static algorithms, with dyn-kEC-pf giving recourse as low as 30 % relative to kEC. Note that reduced recourse implies less reconfiguration cost in the optical network and, thus, less overhead.

7.5 Conclusion

We studied efficient algorithms to maximize the amount of traffic which can be offloaded to the optical topology in a demand-aware manner. In particular, we presented several dynamic algorithms which exploit the temporal structure of workloads, by computing the disjoint matchings offered by optical switches in an adaptive manner.

Our work opens several interesting avenues for future research. In particular, we have so far focused on algorithms running on a centralized controller, which matches the predominant architectural datacenter network model today [58, 140, 148]. Still, it would be interesting to explore distributed versions of our algorithms: emerging reconfigurable datacenter architectures support a distributed control, e.g., performed directly on the switches. Furthermore, we have so far focused on *online* algorithms which do not require any knowledge of future traffic demands. This is attractive, as the predictability of such traffic demands is application-specific. Nevertheless, it would be interesting to explore how a dynamic scheduler can benefit from headroom, as it may be available for repetitive applications (e.g., related to learning): such knowledge may be exploited to pre-compute solutions dynamically to some extent and hence to further improve the algorithms. Our algorithms may also be adapted to the related problem of scheduling connections in satellite-based communication networks, where colors correspond to time slots [75]. The problem differs in that the matchings are unweighted and bounded in cardinality.

[1] The world's most valuable resource is no longer oil, but data,
The Economist **423**(9039), 9 (2017).

[2] S. J. Little, S. L. Kosakovsky Pond, C. M. Anderson, J. A. Young, J. O. Wertheim, S. R. Mehta, S. May, D. M. Smith,
Using HIV networks to inform real time prevention interventions,
PLOS ONE **9**(6), 1–8 (2014).

[3] C. Auffray, R. Balling, I. Barroso, L. Bencze, M. Benson, J. Bergeron, E. Bernal-Delgado, N. Blomberg, C. Bock, A. Conesa, S. Del Signore, C. Delogne, P. Devilee, A. Di Meglio, M. Eijkemans, P. Flicek, N. Graf, V. Grimm, H.-J. Guchelaar, Y.-K. Guo, I. G. Gut, A. Hanbury, S. Hanif, R.-D. Hilgers, Á. Honrado, D. R. Hose, J. Houwing-Duistermaat, T. Hubbard, S. H. Janacek, H. Karanikas, T. Kievits, M. Kohler, A. Kremer, J. Lanfear, T. Lengauer, E. Maes, T. Meert, W. Müller, D. Nickel, P. Oledzki, B. Pedersen, M. Petkovic, K. Pliakos, M. Rattray, J. R. i Màs, R. Schneider, T. Sengstag, X. Serra-Picamal, W. Spek, L. A. I. Vaas, O. van Batenburg, M. Vandelaer, P. Varnai, P. Villoslada, J. A. Vizcaíno, J. P. M. Wubbe, G. Zanetti,
Making sense of big data in health research: towards an eu action plan,
Genome Med. **8**, 1–13 (2016).

[4] A. I. Paganelli, A. G. Mondéjar, A. C. da Silva, G. Silva-Calpa, M. F. Teixeira, F. Carvalho, A. Raposo, M. Endler,
Real-time data analysis in health monitoring systems: a comprehensive systematic literature review,
J. Biomed. Inform. **127**, 104009 (2022).

[5] H. Edelsbrunner, D. Letscher, A. Zomorodian,
Topological persistence and simplification,
Discrete Comput. Geom. **28**(4), 511–533 (2002).

[6] P. Bendich, J. S. Marron, E. Miller, A. Pieloch, S. Skwerer,
Persistent homology analysis of brain artery trees,
Ann. Appl. Stat. **10**(1), 198 (2016).

[7] Z. Gracia-Tabuenca, J. C. Díaz-Patiño, I. Arelio, S. Alcauter,
Topological data analysis reveals robust alterations in the whole-brain and frontal lobe functional connectomes in attention-deficit/hyperactivity disorder,
eNeuro **7**(3) (2020).

[8] Y.-M. Chung, C.-S. Hu, Y.-L. Lo, H.-T. Wu,
A persistent homology approach to heart rate variability analysis with an application to sleep-wake classification,
Front. Physiol. **12** (2021).

[9] G. Graff, B. Graff, P. Pilarczyk, G. Jabłoński, D. Gąsecki, K. Narkiewicz,
Persistent homology as a new method of the assessment of heart rate variability,
PLOS ONE **16**(7), e0253851 (2021).

[10] C. Biwer, A. Rothberg, H. IglayReger, H. Derksen, C. F. Burant, K. Najarian,
Windowed persistent homology: a topological signal processing algorithm applied to clinical obesity data,
PLOS ONE **12**(5), 1–14 (2017).

[11] M.-L. Dequéant, S. Ahnert, H. Edelsbrunner, T. M. A. Fink, E. F. Glynn, G. Hattem, A. Kudlicki, Y. Mileyko, J. Morton, A. R. Mushegian, L. Pachter, M. Rowicka, A. Shiu, B. Sturmfels, O. Pourquié,
Comparison of pattern detection methods in microarray time series of the segmentation clock,
PLOS ONE **3**(8), e2856 (2008).

[12] J. A. Perea, A. Deckard, S. B. Haase, J. Harer,
SW1PerS: sliding windows and 1-persistence scoring; discovering periodicity in gene expression time series data,
BMC Bioinformatics **16**(1), 257 (2015).

[13] J. M. Chan, G. Carlsson, R. Rabadan,
Topology of viral evolution,
PNAS **110**(46), 18566–18571 (2013).

[14] M. Gidea, Y. Katz,
Topological data analysis of financial time series: landscapes of crashes,
Physica A **491**, 820–834 (2018).

[15] G. E. Carlsson, T. Ishkhanov, V. de Silva, A. Zomorodian,
On the local behavior of spaces of natural images,
Int. J. Comput. Vis. **76**(1), 1–12 (2008).

[16] K. Xia, Z. Zhao, G.-W. Wei,
Multiresolution persistent homology for excessively large biomolecular datasets,
J. Chem. Phys. **143**(13), 134103 (2015).

[17] Z. Cang, G.-W. Wei,
Integration of element specific persistent homology and machine learning for protein-ligand binding affinity prediction,
Int. J. Numer. Methods Biomed. Eng. **34**(2), e2914 (2018).

[18] Z. Cang, G.-W. Wei,
TopologyNet: topology based deep convolutional and multi-task neural networks for biomolecular property predictions,
PLoS Comput. Biol. **13**(7) (2017).

[19] Y. Hiraoka, T. Nakamura, A. Hirata, E. G. Escolar, K. Matsue, Y. Nishiura,
Hierarchical structures of amorphous solids characterized by persistent homology,
PNAS **113**(26), 7035–7040 (2016).

[20] J. A. Perea, J. Harer,
Sliding windows and persistence: an application of topological methods to signal analysis,
Found. Comput. Math. **15**(3), 799–838 (2015).

[21] S. Zeng, F. Graf, C. D. Hofer, R. Kwitt,
Topological attention for time series forecasting,
in *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)* (2021).

[22] D. Cohen-Steiner, H. Edelsbrunner, J. Harer,
Extending persistence using Poincaré and Lefschetz duality,
Found. Comput. Math **9**(1), 79–103 (2009).

[23] D. Cohen-Steiner, H. Edelsbrunner, J. Harer,
Stability of persistence diagrams,
Discrete Comput. Geom. **37**(1), 103–120 (2007).

[24] H. Edelsbrunner, J. L. Harer,
Computational topology: an introduction,
American Mathematical Society (2022).

[25] R. Biswas, S. Cultrera di Montesano, H. Edelsbrunner, M. Saghafian,
Geometric characterization of the persistence of 1D maps,
J. Appl. Comput. Topology (2023).

[26] M. Glisse,
Fast persistent homology computation for functions on $\mathbb{R}$,
arXiv:2301.04745 [cs.CG] (2023).

[27] H. Edelsbrunner, D. Morozov,
Persistent homology,
in *Handbook of Discrete and Computational Geometry*, Chapman and Hall/CRC (2017).

[28] D. Cohen-Steiner, H. Edelsbrunner, D. Morozov,
Vines and vineyards by updating persistence in linear time,
in *Proceedings of the International Symposium on Computational Geometry (SoCG)* (2006).

[29] T. K. Dey, T. Hou, S. Parsa,
Revisiting graph persistence for updates and efficiency,
in *Proceedings of the Algorithms and Data Structures Symposium (WADS)* (2023).

[30] J. Vuillemin,
A unifying look at data structures,
Commun. ACM **23**(4), 229–239 (1980).

[31] C. Maria, J.-D. Boissonnat, M. Glisse, M. Yvinec,
The Gudhi library: simplicial complexes and persistent homology,
in *Proceedings of the International Congress on Mathematical Software (ICMS)* (2014).

[32] L. Sweeney,
Weaving technology and policy together to maintain confidentiality,
J. Law Med. Ethics **25**(2-3), 98–110 (1997).

[33] A. Tockar,
`https://agkn.wordpress.com/2014/09/15/riding-with-the-stars-passenger-privacy-in-the-nyc-taxicab-dataset/` (2014);
accessed: Aug. 14, 2024; originally published at `http://research.neustar.biz/2014/09/15/riding-with-the-stars-passenger-privacy-in-the-nyc-taxicab-dataset/`.

[34] A. Narayanan, V. Shmatikov,
Robust de-anonymization of large sparse datasets,
in *Proceedings of the Symposium on Security and Privacy (SP)* (2008).

[35] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, V. Shmatikov,
"You might also like:" privacy risks of collaborative filtering,
in *Symposium on Security and Privacy (SP)* (2011).

[36] A. Narayanan, J. Huey, E. W. Felten,
A precautionary approach to big data privacy,
in *Data Protection on the Move: Current Developments in ICT and Privacy/Data Protection*, Springer (2016).

[37] L. Sweeney,
K-anonymity: a model for protecting privacy,
Int. J. Uncertain. Fuzziness Knowl. Based Syst. **10**(5), 557–570 (2002).

[38] S. Chawla, C. Dwork, F. McSherry, A. D. Smith, H. Wee,
Toward privacy in public databases,
in *Proceedings of the Theory of Cryptography Conference (TCC)* (2005).

[39] S. Chawla, C. Dwork, F. McSherry, K. Talwar,
On privacy-preserving histograms,
in *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)* (2005).

[40] A. Machanavajjhala, D. Kifer, J. Gehrke, M. Venkitasubramaniam,
L-diversity: privacy beyond *k*-anonymity,
ACM Trans. Knowl. Discov. Data **1**(1), 3 (2007).

[41] N. Li, T. Li, S. Venkatasubramanian,
T-closeness: privacy beyond k-anonymity and l-diversity,
in *Proceedings of the International Conference on Data Engineering (ICDE)* (2007).

[42] C. Dwork, F. McSherry, K. Nissim, A. Smith,
Calibrating noise to sensitivity in private data analysis,
in *Proceedings of the Theory of Cryptography Conference (TCC)* (2006).

[43] C. Dwork,
Differential privacy,
in *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)* (2006).

[44] A. Machanavajjhala, D. Kifer, J. M. Abowd, J. Gehrke, L. Vilhuber,
Privacy: theory meets practice on the map,
in *Proceedings of the International Conference on Data Engineering (ICDE)* (2008).

[45] Ú. Erlingsson, V. Pihur, A. Korolova,
RAPPOR: randomized aggregatable privacy-preserving ordinal response,
in *Proceedings of the Conference on Computer and Communications Security (CCS)* (2014).

[46] K. Kenthapadi, T. T. L. Tran,
PriPeARL: a framework for privacy-preserving analytics and reporting at linkedin,
in *Proceedings of the International Conference on Information and Knowledge Management (CIKM)* (2018).

[47] N. M. Johnson, J. P. Near, D. Song,
Towards practical differential privacy for SQL queries,
Proc. VLDB Endow. **11**(5), 526–539 (2018).

[48] C. Dwork, A. Roth,
The algorithmic foundations of differential privacy,
Found. Trends Theor. Comput. Sci. **9**(3-4), 211–407 (2014).

[49] S. P. Vadhan,
The complexity of differential privacy,
in *Tutorials on the Foundations of Cryptography*, Springer (2017).

[50] K. Nissim, S. Raskhodnikova, A. Smith,
Smooth sensitivity and sampling in private data analysis,
in *Proceedings of the Symposium on Theory of Computing (STOC)* (2007).

[51] C. Dwork, J. Lei,
Differential privacy and robust statistics,
in *Proceedings of the Symposium on Theory of Computing (STOC)* (2009).

[52] C. Dwork,
Differential privacy in new settings,
in *Proceedings of the Symposium on Discrete (SODA)* (2010).

[53] S. Song, S. Little, S. Mehta, S. Vinterbo, K. Chaudhuri,
Differentially private continual release of graph statistics,
arXiv:1809.02575 [cs.CR] (2018).

[54] C. Dwork, M. Naor, T. Pitassi, G. N. Rothblum,
Differential privacy under continual observation,
in *Proceedings of the Symposium on Theory of Computing (STOC)* (2010).

[55] T.-H. H. Chan, E. Shi, D. Song,
Private and continual release of statistics,
ACM Trans. Inf. Syst. Secur. **14**(3) (2011).

[56] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, S. Vadhan,
On the complexity of differentially private data release: efficient algorithms and hardness results,
in *Proceedings of the Symposium on Theory of Computing (STOC)* (2009).

[57] M. Lyu, D. Su, N. Li,
Understanding the sparse vector technique for differential privacy,
Proc. VLDB Endow. **10**(6) (2017).

[58] M. Nance Hall, K.-T. Foerster, S. Schmid, R. Durairajan,
A survey of reconfigurable optical networks,
OSN **41**, 100621 (2021).

[59] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken,
The nature of data center traffic: measurements & analysis,
in *Proceedings of the Internet Measurement Conference (IMC)* (2009).

[60] C. Avin, M. Ghobadi, C. Griner, S. Schmid,
On the complexity of traffic traces and implications,
Proc. ACM Meas. Anal. Comput. Syst. **4**(1) (2020).

[61] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, Y. Chen,
OSA: an optical switching architecture for data center networks with unprecedented flexibility,
IEEE/ACM Trans. Netw. **22**(2), 498–511 (2014).

[62] F. Douglis, S. Robertson, E. den van Berg, J. Micallef, M. Pucci, A. Aiken, K. Bergman, M. Hattink, M. Seok,
FLEET—fast lanes for expedited execution at 10 terabits: program overview,
IEEE Internet Comput. **25**(3), 79–87 (2021).

[63] M. Y. Teh, Z. Wu, K. Bergman,
Flexspander: augmenting expander networks in high-performance systems with optical bandwidth steering,
J. Opt. Commun. Netw. **12**(4), B44–B54 (2020).

[64] C. Griner, J. Zerwas, A. Blenk, M. Ghobadi, S. Schmid, C. Avin,
Cerberus: the power of choices in datacenter topology design – a throughput perspective,
Proc. ACM Meas. Anal. Comput. Syst. **5**(3) (2021).

[65] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, A. Vahdat,
Helios: a hybrid electrical/optical switch architecture for modular data centers,
SIGCOMM Comput. Commun. Rev. **40**(4), 339–350 (2010).

[66] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, G. Porter,
Circuit switching under the radar with REACToR,
in *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI)* (2014).

[67] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, A. C. Snoeren,
Scheduling techniques for hybrid circuit/packet networks,
in *Proceedings of the Conference on Emerging Networking Experiments and Technologies (CoNEXT)* (2015).

[68] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, A. Vahdat,
Integrating microsecond circuit switching into the data center,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2013).

[69] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, M. Ryan,
c-Through: part-time optics in data centers,
SIGCOMM Comput. Commun. Rev. **40**(4), 327–338 (2010).

[70] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, D. Kilper,
ProjecToR: agile reconfigurable data center interconnect,
in *Proceedings of the 2016 ACM SIGCOMM Conference* (2016).

[71] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, A. Tanwer,
FireFly: a reconfigurable wireless data center fabric using free-space optics,
in *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014).

[72] S. Kandula, J. Padhye, P. Bahl,
Flyways to de-congest data center networks,
in *Proceedings of the Workshop on Hot Topics in Networks (HotNets)* (2009).

[73] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, H. Zheng,
Mirror mirror on the ceiling: flexible wireless links for data centers,
SIGCOMM Comput. Commun. Rev. **42**(4), 443–454 (2012).

[74] K. Hanauer, M. Henzinger, S. Schmid, J. Trummer,
Fast and heavy disjoint weighted matchings for demand-aware datacenter topologies,
in *Proceedings of the International Conference on Computer Communications (INFOCOM)* (2022).

[75] U. Feige, E. Ofek, U. Wieder,
Approximating maximum edge coloring in multigraphs,
in *Proceedings of the Conference on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)* (2002).

[76] Z.-Z. Chen, S. Konno, Y. Matsushita,
Approximating maximum edge 2-coloring in simple graphs,
Discrete Appl. Math. **158**(17), 1894–1901 (2010).

[77] M. Kamiński, Ł. Kowalik,
Beyond the Vizing's bound for at most seven colors,
SIAM J. Discrete Math. **28**(3), 1334–1362 (2014).

[78] L. M. Favrholdt, M. N. Nielsen,
On-line edge-coloring with a fixed number of colors,
Algorithmica **35**(2), 176–191 (2003).

[79] N. Grüttemeier, C. Komusiewicz, N. Morawietz,
Maximum edge-colorable subgraph and strong triadic closure parameterized by distance to low-degree graphs,
in *Proceedings of the Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)* (2020).

[80] A. Agrawal, M. Kundu, A. Sahu, S. Saurabh, P. Tale,
Parameterized complexity of maximum edge colorable subgraph,
Algorithmica **84**(10), 3075–3100 (2022).

[81] J. Misra, D. Gries,
A constructive proof of Vizing's theorem,
Inf. Process. Lett. **41**(3), 131–133 (1992).

[82] H. Fichtenberger, M. Henzinger, J. Upadhyay,
Constant matters: fine-grained error bound on differentially private continual observation,
in *Proceedings of the International Conference on Machine Learning (ICML)* (2023).

[83] P. Jain, A. Smith, C. Wagaman,
Time-aware projections: truly node-private graph statistics under continual observation*,
in *Proceedings of the Symposium on Security and Privacy (SP)* (2024).

[84] S. Raskhodnikova, T. A. Steiner,
Fully dynamic graph algorithms with edge differential privacy,
arXiv:2409.17623 [cs.DS] (2024).

[85] A. El-Hayek, K. Hanauer, M. Henzinger,
On b-matching and fully-dynamic maximum k-edge coloring,
arXiv:2310.01149 [cs.DS] (2023).

[86] S. M. Ferdous, B. Samineni, A. Pothen, M. Halappanavar, B. Krishnamoorthy,
Semi-streaming algorithms for weighted k-disjoint matchings,
in *Proceedings of the European Symposium on Algorithms (ESA)* (2024).

[87] G. Carlsson,
Topology and data,
Bull. Amer. Math. Soc. **46**(2), 255–308 (2009).

[88] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, H. A. Harrington,
A roadmap for the computation of persistent homology,
EPJ Data Sci. **6**(1), 17 (2017).

[89] D. Smirnov, D. Morozov,
Triplet merge trees,
in *Topological Methods in Data Analysis and Visualization V (TopoInVis)* (2020).

[90] Y. Luo, B. J. Nelson,
Accelerating iterated persistent homology computations with warm starts,
Comput. Geom. **120**, 102089 (2024).

[91] M. Piekenbrock, J. A. Perea,
Move schedules: fast persistence computations in coarse dynamic settings,
J. Appl. Comput. Topol. **8**(2), 301–345 (2024).

[92] R. Seidel, C. R. Aragon,
Randomized search trees,
Algorithmica **16**(4), 464–497 (1996).

[93] S. Cultrera di Montesano, H. Edelsbrunner, M. Henzinger, L. Ost,
Dynamically maintaining the persistent homology of time series,
in *Proceedings of the Symposium on Discrete Algorithms (SODA)* (2024).

[94] D. Morozov,
Dionysus,
`https://mrzv.org/software/dionysus/` (2023).

[95] D. Morozov,
Dionysus 2,
https://mrzv.org/software/dionysus2/ (2023).

[96] U. Bauer,
Ripser: efficient computation of Vietoris–Rips persistence barcodes,
J. Appl. Comput. Topol. **5**(3), 391–423 (2021).

[97] O. Krzikalla, I. Gaztanaga,
Boost.Intrusive C++ library,
http://www.boost.org/;
(version 1.74).

[98] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, H. E. Stanley,
PhysioBank, PhysioToolkit, and PhysioNet,
Circulation **101**(23), e215–e220 (2000).

[99] E. A. P. Alday, A. Gu, A. J. Shah, C. Robichaux, A.-K. I. Wong, C. Liu, F. Liu, A. B. Rad, A. Elola, S. Seyedi, Q. Li, A. Sharma, G. D. Clifford, M. A. Reyna,
Classification of 12-lead ECGs: the PhysioNet/Computing in Cardiology Challenge 2020,
Physiol. Meas. **41**(12), 124003 (2020).

[100] M. A. Reyna, N. Sadr, E. A P. Alday, A. Gu, A. J. Shah, C. Robichaux, A. B. Rad, A. Elola, S. Seyedi, S. Ansari, H. Ghanbari, Q. Li, A. Sharma, G. D. Clifford,
Will two do? varying dimensions in electrocardiography: the PhysioNet/Computing in Cardiology Challenge 2021,
in *Proceedings of Computing in Cardiology (CinC)* (2021).

[101] J. Donley, V. Tourbabin, J.-S. Lee, M. Broyles, H. Jiang, J. Shen, M. Pantic, V. K. Ithapu, R. Mehra,
EasyCom: an augmented reality dataset to support algorithms for easy communication in noisy environments,
arXiv:2107.04174 [cs.SD] (2021).

[102] A. Reiss,
PAMAP2 physical activity monitoring,
UCI Machine Learning Repository (2012).

[103] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, K. Talwar,
Privacy, accuracy, and consistency too: a holistic solution to contingency table release,
in *Proceedings of the Symposium on Principles of Database Systems (PODS)* (2007).

[104] F. McSherry, K. Talwar,
Mechanism design via differential privacy,
in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)* (2007).

[105] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, A. Smith,
What can we learn privately?,
in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)* (2008).

[106] A. Gupta, K. Ligett, F. McSherry, A. Roth, K. Talwar,
Differentially private combinatorial optimization,
in *Proceedings of the Symposium on Discrete Algorithms (SODA)* (2010).

[107] A. Blum, K. Ligett, A. Roth,
A learning theory approach to noninteractive database privacy,
J. ACM (2013).

[108] V. Karwa, S. Raskhodnikova, A. Smith, G. Yaroslavtsev,
Private analysis of graph structure,
Proc. VLDB Endow. **4**(11), 1146–1157 (2011).

[109] J. Blocki, A. Blum, A. Datta, O. Sheffet,
Differentially private data analysis of social networks via restricted sensitivity,
in *Proceedings of the Conference on Innovations in Theoretical Computer Science (ITCS)* (2013).

[110] S. Chen, S. Zhou,
Recursive mechanism: towards node differential privacy and unrestricted joins,
in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2013).

[111] S. P. Kasiviswanathan, K. Nissim, S. Raskhodnikova, A. Smith,
Analyzing graphs with node differential privacy,
in *Proceedings of the Theory of Cryptography Conference (TCC)* (2013).

[112] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, X. Xiao,
Private release of graph statistics using ladder functions,
in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2015).

[113] M. Hay, C. Li, G. Miklau, D. Jensen,
Accurate estimation of the degree distribution of private networks,
in *Proceedings of the International Conference on Data Mining (ICDM)* (2009).

[114] W.-Y. Day, N. Li, M. Lyu,
Publishing graph degree distribution with node differential privacy,
in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2016).

[115] S. Zhang, W. Ni, N. Fu,
Differentially private graph publishing with degree distribution preservation,
Comput. Secur. , 102285 (2021).

[116] Y. Wang, X. Wu, L. Wu,
Differential privacy preserving spectral graph analysis,
in *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2013).

[117] R. Arora, J. Upadhyay,
On differentially private graph sparsification and applications,
in *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)* (2019).

[118] M. Eliás, M. Kapralov, J. Kulkarni, Y. T. Lee,
Differentially private release of synthetic graphs,
in *Proceedings of the Symposium on Discrete Algorithms (SODA)* (2019).

[119] W. Lu, G. Miklau,
Exponential random graph estimation under differential privacy,
in *Proceedings of the Conference on Knowledge Discovery and Data Mining (KDD)* (2014).

[120] T. .-H H. Chan, M. Li, E. Shi, W. Xu,
Differentially private continual monitoring of heavy hitters from distributed streams,
in *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)* (2012).

[121] G. Kellaris, S. Papadopoulos, X. Xiao, D. Papadias,
Differentially private event sequences over infinite streams,
Proc. VLDB Endow. **7**(12), 1155–1166 (2014).

[122] J. L. Ny, G. J. Pappas,
Differentially private filtering,
IEEE Trans. Autom. Control **59**(2), 341–354 (2014).

[123] M. A. Erdogdu, N. Fawaz,
Privacy-utility trade-off under continual observation,
in *Proceedings of the International Symposium on Information Theory (ISIT)* (2015).

[124] Q. Wang, Y. Zhang, X. Lu, Z. Wang, Z. Qin, K. Ren,
Real-time and spatio-temporal crowd-sourced social network data publishing with differential privacy,
IEEE Trans. Dependable Secure Comput. **15**(4), 591–606 (2018).

[125] M. Hardt, G. N. Rothblum,
A multiplicative weights mechanism for privacy-preserving data analysis,
in *Proceedings of the Symposium on Foundations of Computer Science* (2010).

[126] A. Roth, T. Roughgarden,
Interactive privacy via the median mechanism,
in *Proceedings of the Symposium on Theory of Computing (STOC)* (2010).

[127] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, M. Yu,
HPCC: high precision congestion control,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2019).

[128] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, E. Ebrahimi,
SiP-ML: high-bandwidth optical network interconnects for machine learning training,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2021).

[129] J. C. Mogul, L. Popa,
What we talk about when we talk about cloud network performance,
SIGCOMM Comput. Commun. Rev. **42**(5), 44–48 (2012).

[130] M. Hampson,
Reconfigurable optical networks will move supercomputer data 100x faster,
IEEE Spectrum (2021).

[131] C. Avin, K. Mondal, S. Schmid,
Demand-aware network designs of bounded degree,
Distrib. Comput. **33**(3), 311–325 (2020).

[132] C. Avin, S. Schmid,
Toward demand-aware networking: a theory for self-adjusting networks,
SIGCOMM Comput. Commun. Rev. **48**(5), 31–40 (2019).

[133] A. Roy, H. Zeng, J. Bagga, G. Porter, A. C. Snoeren,
Inside the social network's (datacenter) network,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2015).

[134] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, G. Porter,
RotorNet: a scalable, low-complexity, optical datacenter network,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2017).

[135] J. Kulkarni, S. Schmid, P. Schmidt,
Scheduling opportunistic links in two-tiered reconfigurable datacenters,
in *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2021).

[136] A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang,
Proteus: a topology malleable data center network,
in *Proceedings of the Workshop on Hot Topics in Networks (HotNets)* (2010).

[137] S. B. Venkatakrishnan, M. Alizadeh, P. Viswanath,
Costly circuits, submodular schedules and approximate carathéodory theorems,
Queueing Syst. **88**(3-4), 311–347 (2018).

[138] C. Avin, S. Schmid,
ReNets: statically-optimal demand-aware networks,
in *Proceedings of the Symposium on Algorithmic Principles of Computer Systems (APOCS)* (2021).

[139] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, H. Williams,
Sirius: a flat datacenter network with nanosecond optical switching,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2020).

[140] M. Zhang, J. Zhang, R. Wang, R. Govindan, J. C. Mogul, A. Vahdat,
Gemini: practical reconfigurable datacenter networks with topology and traffic engineering,
arXiv:2110.08374 [cs.NI] (2021).

[141] J. Zerwas, C. Györgyi, A. Blenk, S. Schmid, C. Avin,
Duo: a high-throughput reconfigurable datacenter network using local routing and control,
Proc. ACM Meas. Anal. Comput. Syst. **7**(1) (2023).

[142] V. Addanki, C. Avin, S. Schmid,
MARS: near-optimal throughput with shallow buffers in reconfigurable datacenter networks,
Proc. ACM Meas. Anal. Comput. Syst. **7**(1) (2023).

[143] K. Hanauer, M. Henzinger, C. Schulz,
Recent advances in fully dynamic graph algorithms – a quick reference guide,
ACM J. Exp. Algorithmics **27**, 1.11:1–1.11:45 (2022).

[144] M. Bienkowski, D. Fuchssteiner, J. Marcinkowski, S. Schmid,
Online dynamic b-matching: with applications to reconfigurable datacenter networks,
SIGMETRICS Perform. Eval. Rev. **48**(3), 99–108 (2020).

[145] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, S. Shenker,
pFabric: minimal near-optimal datacenter transport,
in *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)* (2013).

[146] D. Chakrabarti, Y. Zhan, C. Faloutsos,
R-MAT: A recursive model for graph mining,
in *Proceedings of the International Conference on Data Mining (ICDM)* (2004).

[147] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, A. Pothen,
Approximate weighted matching on emerging manycore and multithreaded architectures,
Int. J. High Perform. Comput. Appl. **26**(4), 413–430 (2012).

[148] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, A. Vahdat,
Jupiter rising: a decade of Clos topologies and centralized control in Google's datacenter network,
SIGCOMM Comput. Commun. Rev. **45**(4), 183–197 (2015).