UNIVERSITY OF CALIFORNIA
Santa Barbara

Approximation Algorithms for Problems on Networks and Streams of Data

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Luca Foschini

Committee in Charge:

Professor S. Suri, Chair

Professor J. Gilbert

Professor T. Gonzalez

September 2012

The Dissertation of
Luca Foschini is approved:

Professor J. Gilbert

Professor T. Gonzalez

Professor S. Suri, Committee Chairperson

September 2012

Approximation Algorithms for Problems on Networks and Streams of Data

To my brother, and to all those who can’t wait to properly decorate this abundance of white space.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Subhash Suri, for his guidance during these years. I am very grateful for everything I have learned from him. I would also like to thank my committee whose feedback has been crucial for the development of this dissertation.

I greatly appreciated the mentorship of those who hosted me during my summer research internships, including Vahab Mirrokni and Eyal Carmi at Google, and Professor Widmayer at ETH Zurich.

Finally, I would like to acknowledge my former and current lab mates Pegah, Kyle, Hakan, and Sorabh for their assistance and support. Others who have provided significant input and guidance along the way include Prof. Giovanni and Sebastiano Vigna, Dr. Antonio Gulli and Dr. Antonio Savona. Without any of them, this dissertation would be incomplete.

Curriculum Vitæ

Luca Foschini

Education

2012 (Expected)	PhD in Computer Science, University of California, Santa Barbara.
2012	Master of Science in Computer Science, University of California, Santa Barbara.
2007	Master of Engineering in Computer Engineering, University of Pisa, Pisa, Italy. (Cum Laude)
2004	Bachelor of Engineering in Computer Engineering, University of Pisa, Pisa, Italy. (Cum Laude)

Experience

2012	Co-founder and Chief Scientist at The Activity Exchange Inc. (AchieveMint)
2011	Research Intern, Google Research NYC
2010	Visiting Scholar at ETH Zurich, Zurich.
2005 – 2009	Software Engineer at Ask.com
2004	Research Intern at CERN, Geneva.

Selected Publications

A. E. Feldmann, L. Foschini *Balanced Partitions of Trees and Applications*. Submitted to Algorithmica.

L. Foschini, J. Hershberger, S. Suri *On the Complexity of Time-Dependent Shortest Paths*. Submitted to Algorithmica.

A. E. Feldmann and L. Foschini *Balanced Partitions of Trees and Applications*, 29th Symposium on Theoretical Aspects of Computer Science (STACS12)

H. Yıldız, L. Foschini, J. Hershberger, S. Suri *Maintaining the Volume of the Union of Boxes and Related Problems*, 19th Annual European Symposium on Algorithms (ESA11)

L. Cavedon, L. Foschini, G. Vigna *Getting the Face Behind the Squares: Reconstructing Pixelized Video Streams*, 5th USENIX Workshop on Offensive Technologies (WOOT11)

S. Gauglitz, L. Foschini, M. Turk, T. Höllerer *Efficiently Selecting Spatially Distributed Keypoints for Visual Tracking*, IEEE International Conference on Image Processing (ICIP11)

F. Uyeda, L. Foschini, S. Suri, G. Varghese *Efficiently Measuring Bandwidth at All Time Scales*, 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI11)

L. Foschini, J. Hershberger, S. Suri *On the Complexity of Time-Dependent Shortest Paths*, 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA11)

C. Buragohain, L. Foschini, S. Suri *Untangling the Braid: Finding Outliers in a Set of Streams*, SODA Workshop on Algorithm Engineering and Experiments (ALENEX10)

S. Gandhi, L. Foschini, S. Suri *Space-efficient Online Approximation of Time Series Data: Streams, Amnesia, and Out-of-order*, 26th International Conference on Data Engineering (ICDE10)

L. Foschini, R. Grossi, A. Gupta, J. S. Vitter *When indexing equals compression: Experiments with compressing suffix arrays and applications*, ACM Transactions on Algorithms 2(4): 611-639 (2006)

L. Foschini, R Grossi, A. Gupta, J. S. Vitter *Fast Compression with a static model in High-Order Entropy*, 2004 IEEE Data Compression Conference (DCC04)

M. Cococcioni, L. Foschini, B. Lazzerini, F. Marcelloni *Complexity Reduction of Mamdani Fuzzy Systems through Multi-valued Logic Minimization*, 2008 IEEE International Conference on Systems, Man and Cybernetics (SMC08)

L. Valcarenghi, L. Foschini, F. Paolucci, F. Cugini, and P. Castoldi *Topology Discovery Services for Monitoring the Global Grid* Communications Magazine, Optical Control Plane for Grid Networks: Opportunities, Challenges and the Vision, March 2006.

L. Foschini, A. V. Thapliyal, L. Cavallaro, C. Kruegel, G. Vigna *A Parallel Architecture for Stateful, High-Speed Intrusion Detection*, Fourth International Conference on Information Systems Security (ICISS 2008)

A. Gulli, S. Cataudella, L. Foschini *TC-SocialRank: Ranking the Social Web*, The 6th Workshop on Algorithms and Models for the Web Graph (WAW09)

L. Valcarenghi, F. Paolucci, L. Foschini, F. Cugini, P. Castoldi *Centralized and Distributed Grid Topology Discovery Service Implementations* Accepted as a Poster at Hot Interconnects 13, (IEEE Symposium on High Performance Interconnects).

Patents

A. Savona, A. Gulli, L. Foschini, *Systems and methods for selecting and organizing information using temporal clustering* United States Patent: 20070260586

A. Savona, A. Gulli, L. Foschini, G. Deretta *Systems and methods for clustering information* United States Patent: 20090070346

Awards

UCSB Dean’s fellowship. UCSB 2011-2012

Sant’Anna Scholarship. Full scholarship (room and board) for the full duration of undergraduate and graduate studies (2001-2007) at the Sant’Anna School of Advanced Studies, Pisa, Italy as a winner of a nation-wide competition.

Abstract

Approximation Algorithms for Problems on Networks and Streams of Data

Luca Foschini

In this dissertation we investigate approximation algorithms for problems defined on networks and streams of data. The unifying theme of all problems discussed is that finding an exact solution to them is impractical. Such impracticality may stem from a provable time complexity characterization in a general computation model, or it may be dictated by constraints on the resources available to the algorithm.

The first problem concerns finding shortest paths in a network with time-variant transit times. We show that under minimal assumptions on the transit time functions, the complexity of the structure of the shortest paths between two network nodes over a time interval may overwhelm a decision maker; thus, it is crucial we devise strategies that approximate such structural complexity.

Another problem on networks presented considers coloring a graph on n nodes with k colors in such a way that all colors are used roughly the same number of times and the number of edges connecting two nodes assigned to different colors is minimized. We prove that the problem is hard to approximate even when restricted to tree instances. Despite this result, we were able to devise efficient approximation algorithms for a relaxed version of the problem on general graphs.

The second part of the dissertation studies problems on streams of data. In this model, a large stream of data is to be processed online using minimal time and memory per data item. These constraints are dictated by the restricted computational resources of the hardware processing the data, and by real time requirements. We first present a simple abstract framework for online approximation of time-series that yields a unified set of algorithms for the data stream model and other variations. Then, we implement our algorithms to solve the problem of identifying network traffic bursts at various and especially fine-grain time scales. The simplicity of our algorithmic framework allows our approximated burst detectors to be implemented directly in the operating system kernel or in constrained hardware such as routers.

Professor S. Suri
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Introduction

Dans ses écrits, un sàge Italien
Dit que le mieux est l'ennemi du bien.

Voltaire, La Bégueule (Contes, 1772)

The identity of the wise Italian who reportedly wrote that the best is the enemy of the good is not known. However, what we do know is that he or she was not the first in the history of mankind to advocate in favor of *approximated problem solving*.

Seemingly contrasting with the idea of mathematics being the epitome of the exact science, the notion of approximation has been entwined with mathematics since the ancient times. In the fifth century BCE pythagoreans made the eerie discovery, for the beliefs of the time, that the diagonal of a square is incommensurable with its side. That implies the impossibility of an exact finite representation of $\sqrt{2}$ in the decimal system—perhaps one of the first "negative results" in the history of mathematics. However, more than a thousand years before that, the Babylonians were already able to compute an *approximation* of $\sqrt{2}$ at six decimal figures. Since the dawn of time, mathematicians has been confronted with the impossibility of eradicating uncertainty wholly, and have thus developed ideas and tools to cope with uncertainty, by rigorously modeling and confining it.

In the last few centuries, techniques to provably confine the error of an approximated solution have become widespread, as the blossoming of astronomy and engineering fostered development of iterative numerical methods, ancestors of modern approximation algorithms. In the 17th century Kepler made extensive use of Simpson's rule to approximate definite integrals—more than a hundred years before Simpson was born. Numerical methods for root finding, derivation, and integration were common knowledge as early as the 18th century.

This is also the time when the idea emerges of quantifying the tradeoff between the accuracy of an approximation method and the number of steps the method requires to achieve such an accuracy. The driving concept is the notion of convergence: the optimal solution to the problem is still the goal longed for, but it can only be achieved in the limit.

With the advent of computers—approximated devices par excellence, right from their core of number representation in floating point—numerical methods flourished. However, as the possibilities of mechanizing computation brought about by computers expanded, it became increasingly clear that such expansion could not be limitless, for computers resources are known and finite. To make the matter worse, advances in the newborn discipline of computational complexity indicated that many problems of practical interests are most likely not solvable exactly, even when computational resources are allowed to be growing polynomially in the size of the problem instance.

This caused a paradigm shift in the field: In the last fifty years the focus has changed from asking *how long* it takes to optimally solve a problem, to the rather reverse perspective of *how well* a problem could be solved using the given amount of resources. The resources of interests are valuable quantities such as time, memory, and in recent years, energy. Such a change of perspective has reduced the importance of the notion of convergence: The optimal solution is no longer the ultimate goal and solving a *relaxation* of the original problem whose solution can be proved to be close to that of the original problem becomes sufficient, and sometime more desirable.

The field of computer science has thrived around these new ideas, and recent times have seen the study of approximation algorithms take center stage. The need for efficient approximation algorithms has furthermore been recently amplified due to fundamental changes that have reshaped the relationship between hardware, data, and applications.

First, applications have become more complex, and the focus of what can be computed has broadened from numbers (roots, integrals, derivatives) to more complex structures: routes, documents, images. What is the best document in a collection that matches a given query? What the best path to go from town A to town B when there is traffic? What is the image of an album that most closely matches a given picture? Analyzing and storing complex structures require more resources, thus calling for sketching and other approximated representations.

Second, the amount of data processed by applications increases at a restless pace: The document collection of web pages has topped one trillion pages, the traffic measures increase everyday as new sensors are being installed, and hundreds of new pictures are being taken every second and uploaded to the web.

Finally, the explosion on the amount and complexity of data and application is closely complemented by the rise of an array of hardware devices with limited capability, such as sensors and mobile devices. These devices are heavily constrained as far as computational power is concerned and they require trading optimal solution for more lightweight computational demands.

These three factors have all contributed create a breed of new problems, mostly of combinatorial nature, for which the computational power to compute an exact solution cannot be afforded and resorting to approximations is paramount.

This dissertation finds its place in the scenario depicted. Here, we analyze several problems coming from diverse domains of computer science. All these problems have in common the leitmotiv that they are “hard” to solve in the specific domain in which are defined, in the sense that the resources required to compute an exact solution in the general case cannot be afforded. The hardness might come from a provable disheartening time or space complexity characterization in a general model of computation, or it may be dictated by specific constrains in the resource available to solve the problem. The bottom line is that computing an optimal solution to any of the presented problem is impractical.

In what follows we present the problems and solutions discussed in this dissertation.

Content of the Dissertation

The dissertation is divided into two parts. The first part, composed of Chapter 1 and 2 studies problems on weighted graphs.

In Chapter 1, we study the problem of finding a time dependent shortest path in a n-node graph where delays on the edges are functions of time. Since delays are time-variant, shortest paths will change with time too, that is, different paths will be optimal at different times. We prove that if the delay functions are piecewise linear (PWL) with a polynomial (in n) number of breakpoints then the shortest path can change $n^{\Omega \log n}$ times and the bound is (polynomially) tight. This implies that if one wants to plot the arrival time to destination as a function of the departure time, the plotted function might change slope a super-polynomial number of times. To overcome this issue, we present an approximation algorithm to plot such function using much fewer breakpoints but retaining a relative-error type guarantee on the arrival time at destination.

Chapter 2 considers the problem of balanced partitions of graphs, where the problem is to color nodes of a n-node graph with k colors such that each color is used at most

$\lceil n/k \rceil$ times and the number of edges connecting nodes of different colors is minimized. We prove that the problem remains hard on unweighted trees even if either the diameter or the degree is restricted to be constant. On the positive side we give a PTAS for trees (in the bicriteria sense) that can be used in combination with tree-embedding results to improve on the best approximation factor known for the problem on general, weighted, graphs.

The second part of the dissertation, made of Chapters 3 and 4, considers problems defined on streams of data. The algorithms used in this model are required to compute statistics over a large amount of data, such as network packets, that can only be accessed online and one at a time. The algorithms are constrained to use a small memory footprint (usually sublinear in the stream size) and are required to process each element as quickly as possible (in time sublinear in the stream size).

In Chapter 3 we present an abstract framework for online approximation of time-series data that yields a unified set of algorithms for the data stream model and other variations. Our framework essentially develops a popular greedy method of bucket-merging into a more generic form, for which we can prove space-quality approximation bounds and offer simpler and unified algorithms.

The conceptual simplicity of our scheme translates into highly practical implementations, as borne out in our simulation studies: the algorithms produce near-optimal approximations, require very small memory footprints, and run extremely fast.

In Chapter 4 we apply the algorithms derived in Chapter 3 to the problem of identifying correlated traffic bursts at various and especially fine-grain, time scales. The combination of Giga-bit link speeds and small switch buffers have led to microbursts, which cause packet drops and large increases in latency. We present the design and implementation of an efficient and flexible end-host bandwidth measurement tool that can identify such bursts in addition to providing a number of other features. Managers can query the tool for bandwidth measurements at resolutions chosen after the traffic was measured. The algorithmic challenge is to support such a posteriori queries without retaining the entire trace or keeping state for all time scales. Our techniques can be implemented in routers and generalized to detect spikes in the usage of any resource at fine time scales.

Chapter 1

Shortest Paths inTime-Dependent Networks[*]

1.1 Introduction

Time-dependent networks are used to model situations in which the cost of traversing an edge varies with time. While the general framework has many applications, the everyday problem of route planning on road networks is easily the most compelling. Due to varying congestion on roads during the day, both the time to travel from a source node s to a destination d and the optimal path between them can change over time. In fact, traffic conditions often create temporary bottlenecks that cause one to reach the destination at more or less the same time despite leaving much later, at a significant

[*]Parts of this chapter appeared in the following publications: [54, 55]

reduction in travel time. By plotting the arrival time as a function of the departure time, one can plan optimal-*time* paths, instead of optimal-*length* paths, as is the current practice.

Besides their natural applications in the management and planning of complex networks such as highways and communication networks, time-dependent shortest paths are also a fundamental problem with non-trivial complexity issues and subtle implications of model parameters. The problem has been studied since 1969 when Dreyfus [40] observed that Dijkstra's algorithm can be used to find a time-dependent shortest path, given a starting time at the source node. Dreyfus implicitly assumed an unrestricted *waiting policy* at intermediate nodes to ensure that if waiting at a node results in an earlier arrival time (due to reduced congestion), then optimal wait times are utilized at nodes. It turns out, however, that the choice of waiting policies and the type of the edge cost functions have non-trivial implications on time-dependent shortest paths, which were explored more fully in the 1980's and 1990's by Orda and Rom [103]. Their findings include, among other results, that if waiting is *forbidden* anywhere in the network and the network is not FIFO (first-in-first-out), then the shortest path may be non-simple and violate the subpath optimality property. There may also be paths that traverse an infinite number of edges but still have finite delay. Thus, in these cases neither Dijkstra's algorithm nor the Bellman-Ford algorithm can find time-dependent shortest paths, and variants of these problems are known to be NP-hard [112].

In time-dependent networks, a non-FIFO edge is equivalent to a FIFO edge if waiting is allowed: one can always wait at the tail of an edge to optimize the arrival time at the head of the edge. In this chapter, therefore, we focus on the FIFO model, with the understanding that waiting at nodes is permitted to deal with any non-FIFO effects. The problem of shortest paths still exhibits surprising complexity, even when the edges are FIFO and their cost functions are piecewise linear and continuous. In particular, the complexity of the following natural problem, which is the focus of our work, has been open for some time: *Given a source node s, compute the* arrival time function *at another node d for all starting times at s.*

Orda and Rom [103] and many others have considered this problem due to its special significance for planning. Rather than asking for the best path starting at a particular time, which is easily computed by Dijkstra's algorithm, the goal here is to construct a *travel planning map* that gives, for any desired arrival time, the optimal time to depart so as to minimize the travel time. We should point out that in our problem time is considered a *continuous* variable, and so the problem has a distinctly different character from the *discrete time* version in which only a finite set of departure times is permitted [19, 21, 98]. In the discrete version, the complexity of the arrival time function is bounded by the discreteness of the departure times, but the arrival time function may suffer from severe and unnatural discontinuities.[1]

Orda and Rom [103] extended the shortest path finding paradigm of Bellman and Ford [53] to work with linear functions as edge costs, instead of scalar costs as in the non-time-dependent case. They left the *computational complexity* of the problem unresolved. Specifically, they presented the running time of their algorithm in terms of a *functional complexity* that "hides" the actual cost of composing and minimizing the arrival time functions. If the edge cost functions are piecewise linear, then their compositions and minimizations at intermediate nodes are also piecewise linear, but the *number of breakpoints* in these functions can grow, and this complexity is omitted from the analysis [103]. It is possible that Orda and Rom may have assumed that this complexity remains polynomially bounded, but no provable bound is known. In fact, in several subsequent papers, both in theoretical works as well as applied papers, the authors have simply presented the performance of their algorithms as if the *size of the arrival time function* is not much worse than linear [39, 70].

The most systematic study of the arrival time function's complexity was performed by Dean [28, 29], who after an initial erroneous claim that the complexity is linear in

[1]One can show that if the true travel time functions for edges are linear but are approximated discretely using fixed-size buckets, then the error in the travel time estimate can grow exponentially, depending on the *slope* of the linear functions.

the number of the edge function breakpoints [28] *conjectured* that this complexity may be *superpolynomial* [29]:

In a FIFO network with piecewise linear arc arrival functions, the arrival function at a destination node can have a superpolynomial number of pieces.

Our Results A resolution of this conjecture is our main result. In particular, we show that in an n-node graph with (polynomial-size) piecewise linear edge cost functions, the arrival time function has complexity $n^{\Theta(\log n)}$ in the worst case. More specifically, even with linear edge costs, there exists an n-node graph in which the shortest path from a source s to a destination d changes $n^{\Omega(\log n)}$ times, or equivalently the arrival time function has $n^{\Omega(\log n)}$ breakpoints. On the other hand, for any n-node graph whose piecewise linear edge cost functions have at most K pieces in total, the complexity of the arrival time function is at most $Kn^{O(\log n)}$. Both of these bounds assume that either the edges respect FIFO transit, or allow arbitrary waiting at nodes to deal with non-FIFO behavior of edges.

We analyze the complexity of the arrival time function by formulating connections to the *parametric* shortest paths problem [20, 64, 95]. The edge costs in both the parametric and the time-dependent shortest path problems are functions of the underlying parameter, but they interact in different ways. In the former, the cost of a path is the sum of edge costs parameterized by a parameter γ that has the *same value* throughout the network, while in the latter, the parameter is time, which varies over the path through compositions: the arrival time at the second edge depends on the transit time over the first edge, and is different from the initial departure time, and so on. In order to use the known lower bound on parametric shortest paths, we show how to transform an instance of the parametric shortest path problem into a time-dependent one while retaining all the breakpoints. The proof of the upper bound first reduces the problem to one with linear cost functions, and then adapts an inductive argument due to Gusfield [64].

Despite the complexity of computing the full arrival time function, some queries on it can still be answered efficiently in the worst case. We show that the *minimum delay* path over a given departure time interval can be computed in polynomial time, without having to compute the arrival time function explicitly. From a road network perspective, the minimum delay path can be seen as the path that requires the least amount of time spent on the road, given the option to freely choose the departure time within a specified interval. We give an algorithm to find the minimum delay path whose departure time lies in a specified interval in $O(K)$ runs of a standard (fixed-departure-time) shortest path algorithm.

Finally, we address the algorithmic question of computing the arrival time function. We propose a $(1+\varepsilon)$-*approximation* scheme that computes a representation of the arrival time function so that the travel time estimated using this approximation is at most $(1+\varepsilon)$ times the true value. If the maximum and minimum travel times are $D_{\max}$ and $D_{\min}$, the approximation has size $O(K\frac{1}{\varepsilon}\log(D_{\max}/D_{\min}))$ and can be computed using $O\left(K\frac{1}{\varepsilon}\log\left(\frac{D_{\max}}{D_{\min}}\right)\log\left(\frac{I}{K\varepsilon D_{\min}}\right)\right)$ calls to a standard shortest path algorithm, where I is the length of the (departure) time interval over which the arrival time function is computed. We also describe a pair of output-sensitive algorithms for computing the arrival time function exactly, with running time dependent on the complexity of that function. Finally, we show that the total number of breakpoints is polynomially bounded if each piece of the edge cost functions has the form $A(t) = at + b$, with slope $a \in \{0, \alpha^{-\beta}, \alpha^{-\beta+1}, \ldots, \alpha^{\beta}\}$, for some $\alpha > 1$ and $\beta \in \mathbb{N}$.

Related Work The study of shortest paths is a cornerstone of combinatorial algorithms. The classical algorithms, such as those by Dijkstra and Bellman-Ford [26, 53], compute shortest paths in a *static* network with fixed edge costs. Many applications, however, deal with a dynamic network in which the edge costs are subject to change. (Changes in the network topology, such as inclusion or exclusion of an edge, can also be modeled through edge costs.) Time-dependent networks are one way to model such dynamics, but there are others as well, which we briefly discuss below.

The *dynamic shortest paths* problem deals with insertion and deletion of edges in a graph, with a focus on maintaining a data structure for efficiently updating the shortest path tree from a single source or supporting fast shortest path queries between two nodes in response to such a change [34, 78, 111, 117]. The *stochastic shortest paths* problem attempts to capture uncertainty in edge costs by modeling them as random variables and computing paths with minimum expected costs [14, 101, 102]. The *weighted region shortest paths* [93, 94] problem models the dynamics of the environment in a continuous space: the space is divided into regions and each region has a different *speed* of travel, however, the speed is not a function of time.

The problem that bears most similarity to time-dependent paths is that of *parametric shortest paths*. In this problem, the cost of each edge e in the graph varies as a linear function of a parameter γ, i.e., $c(e) = a\gamma + b$. The shortest path from a source node s to a destination node d depends on the value of the parameter γ, and the goal in the parametric shortest path problem is to compute the shortest paths for all values of γ. A result by Carstensen [20] shows that in the worst case the shortest path from s to d can change $n^{\Omega(\log n)}$ times as γ varies; a simpler and more direct proof of this lower bound is given by Mulmuley and Shah [95]. An upper bound of $n^{O(\log n)}$ is given by Gusfield [64].

A more restricted form of the parametric function is used by Karp and Orlin [71] and Young, Tarjan, and Orlin [124]. In their model the edge costs are defined as $c(e) = c_e - \gamma$. This simpler form is far more tractable and generally leads to only a polynomial number of distinct shortest paths. Recently, Erickson [41] also used this form to give an efficient algorithm for parametric maximum flow in planar graphs. However, it is the more general linear form studied by Carstensen and Gusfield that is relevant to our time-dependent shortest paths.

In two recent papers, Dehne et al. [31, 30] studied the time dependent shortest path in a setting similar to that described here. For piecewise linear cost functions, they present an output-sensitive algorithm with time complexity proportional to the number of breakpoints in the arrival time function. They also give an *additive* approximation algorithm with polynomial complexity in the worst case.

Time-dependent shortest paths have also been studied from an *applied* perspective, with emphasis on empirical validation and applications [33, 39, 98, 99]. Our result shows that none of these algorithms can be expected to perform well in the worst case.

Chapter Organization The remainder of the chapter is organized as follows. Section 1.2 gives definitions and establishes basic facts about the time-dependent shortest path problem that are used in the following sections. Section 1.3 presents the $n^{\Omega(\log n)}$ lower bound to the complexity of arrival time functions. The result is complemented by the $n^{O(\log n)}$ upper bound given in Section 1.4. Section 1.5 describes our algorithmic results: a scheme to compute a minimum delay path, an algorithm that $(1+\varepsilon)$-approximates the arrival time function, output-sensitive algorithms to compute the arrival time function exactly, and an algorithm to compute the exact arrival time function efficiently when some conditions on the edge delay functions are satisfied. Section 1.6 concludes and discusses open problems.

1.2 Definitions and Preliminaries

We consider a directed graph G with n vertices and m edges, where the cost of each edge is a piecewise linear function of time, representing the time-dependent delay along it. The edge costs respect first-in-first-out (FIFO), meaning that each linear segment in the delay function has slope at least -1. Instead of reasoning with delays, we find it more convenient to work with *arrival time* functions, as did Orda and Rom [103]. In particular, if P is a set of paths with common first and last nodes u and v, then $A[P]$ is the *earliest arrival time* function for P. $A[P]$ is a function of the start time t at node u, written in full as $A[P](t)$, minimizing the arrival time at v over all paths in P, for each value of t. (We use square brackets around the path set argument of $A[P]$ to emphasize that it is discrete; we use parentheses around the continuous t argument.) We define simplified versions of the notation for three common cases:

$A[e]$ is the arrival time function for an edge $e = (u, v)$, that is, $A[e](t)$ gives the time of arrival at vertex v for travel on e departing from vertex u at time t. The function $D[e](t) \equiv (A[e](t) - t)$ is the *delay* or *travel time* along the edge e. Since travel times are nonnegative, $A[e](t) \geq t$ for all t.

$A[p](t)$ is the path arrival function for the path $p = (v_1, v_2, \ldots, v_q)$ defined by functional composition as $A[p](t) = A[(v_{q-1}, v_q)] \cdot A[(v_{q-2}, v_{q-1})] \cdots A[(v_1, v_2)](t)$.

$A[s, d](t) \equiv A[\mathcal{P}_{s,d}]$ is the earliest arrival time function for source s and destination d, minimizing over the set $\mathcal{P}_{s,d}$ of all paths in G from s to d, $A[s, d](t) = \min\big(A[p](t) \mid p \in \mathcal{P}_{s,d}\big)$.

We denote by K the total number of linear segments in all the edge arrival time functions $A[e]$. By a *breakpoint*, we mean a value of t at which the slope of the arrival time function changes. In a piecewise linear function, the number of breakpoints differs from the number of linear segments by only one, so for the sake of notational brevity, we use the two counts interchangeably in our analysis, depending on the context. The following easy lemma characterizes the composition of $A[e]$ functions.

Lemma 1.2.1. Suppose that $x(t)$ and $y(t)$ are monotone, piecewise linear functions. Then their composition $z(t) = y \cdot x(t) = y(x(t))$ is also monotone and piecewise linear. If $z(t)$ has a breakpoint at $t = t_0$, then either x has a breakpoint at t_0 or y has one at $x(t_0)$.

Proof: The composition of monotone functions is monotone. For every t_0 such that $t = t_0$ is not a breakpoint of x and $x(t_0)$ is not a breakpoint of y, x and y are linear functions $x(t) = at + b$ and $y(t) = ct + d$, for particular constants a, b, c, and d. Then $z(t) = y(x(t)) = c(at + b) + d$, which is linear in t. The functional form of $z(t)$ is fixed in the neighborhood of t_0, and so z cannot have a breakpoint there. Thus z is linear except at breakpoints derived from those of x and y. $\square$

1.2.1 Primitive and Minimization Breakpoints

Breakpoints of edge functions $A[e]$ are called *primitive breakpoints*. The function $A[p]$ for a path p is the composition of edge functions; breakpoints of $A[p]$ that arise from the primitive breakpoints of the composed functions are called *primitive images*. A primitive image b_F in the function $F = f_q \cdot f_{q-1} \cdots f_1$ that occurs at $t = t_F$ is the image of a primitive breakpoint b in some f_i if b occurs at $t = t_i$ in $f_i(t)$ and $t_i = f_{i-1} \cdot f_{i-2} \cdots f_1(t_F)$. The primitive breakpoint b whose image is b_F is called the *preimage* of b_F.

Breakpoints in $A[s, d]$ can be of two types, primitive images and *minimization breakpoints*. A minimization breakpoint occurs in $A[s, d]$ at some time t if the arrival time of the path p_1 that was optimal at time $t - \varepsilon$ becomes larger than the arrival time of another path p_2 at time $t + \varepsilon$, for an infinitesimal ε. A minimization breakpoint b occurs because of minimization at a particular node v (the first node in the last section of $p_1 \cap p_2$), somewhere after s and at or before d; each node x in $p_1 \cap p_2$ from v to d has a minimization breakpoint at $t = b$ that is an image of the one at v. Figure 1.1 illustrates the two types of breakpoints. Informally, minimization breakpoints are created by the pointwise minimum operation on the arrival time functions among all the paths $p \in \mathcal{P}_{s,d}$, while primitive images are the breakpoints that each path arrival function has as a consequence of being the functional composition of piecewise linear edge functions.

Lemma 1.2.2. If $x(t)$ and $y(t)$ are monotone, piecewise linear functions, then so is $\min(x(t), y(t))$.

It follows from Lemmas 1.2.1 and 1.2.2 that the function $A[s, d]$ is nondecreasing and piecewise linear.

Observation 1.2.3. $A[s, d]$ can be partitioned into compact intervals (with respect to the parameter t) in which one combinatorial simple path of G is optimal. These intervals are separated by minimization breakpoints in the piecewise linear function representing $A[s, d]$. (See Figure 1.1.)

The number of breakpoints of arrival functions is expressed as $B(\cdot)$. For instance, $B(A[s, d])$ is the number of breakpoints of the earliest arrival function between nodes s and d. We give separate bounds on the cardinality of the sets of primitive images and minimization breakpoints. We first focus on primitive images.

Lemma 1.2.4. Two distinct breakpoints b, b' in the monotone piecewise linear function $f_1 \cdot f_2 \cdots f_q$ are images of two distinct primitive breakpoints in the set of piecewise linear functions $\{f_1, f_2, \ldots, f_q\}$.

Proof: If b and b' are images of breakpoints from different functions f_i and f_j, they are clearly distinct. If they are images of breakpoints from the same function f_i, then by Lemma 1.2.1 and associativity of functional composition their preimages occur at different t values in the domain of f_i, and hence are distinct. □

The following key lemma shows that primitive breakpoints do not create too many images. In particular, while an exponential number of different paths may use an edge, a primitive breakpoint creates at most one image in the arrival time function at any node, and in particular in $A[s, d]$.

Lemma 1.2.5. Any two distinct primitive images b, b' in $A[s, d]$ are images of two distinct primitive breakpoints among the functions $A[e]$, for $e \in G$.

Proof: If b and b' are images of breakpoints from different edge functions, then their preimages must be distinct. Therefore, we need to consider only pairs of breakpoints whose preimages belong to $A[e]$ for the same edge $e \in G$. Recall that $A[s, d] = \min(A[p] \mid p \in \mathcal{P}_{s,d})$ where $\mathcal{P}_{s,d}$ is the set of all paths in G from s to d. Any breakpoints whose preimages come from $A[e]$ must belong to $A[p]$ for some $p \in \mathcal{P}_{s,d\,|\,e}$, where $\mathcal{P}_{s,d\,|\,e}$ is the subset of paths in $\mathcal{P}_{s,d}$ that contain the edge e. If $e = (u, v)$, then by distributivity of function composition over the $\min$ operation we can rewrite $A[\mathcal{P}_{s,d\,|\,e}] = \min(A[p] \mid p \in \mathcal{P}_{s,d\,|\,e})$ as

$$\min(A[p] \mid p \in \mathcal{P}_{vd}) \cdot A[e] \cdot \min(A[p] \mid p \in \mathcal{P}_{su}),$$

corresponding to the concatenation of the shortest path from s to u with edge $e = (u, v)$, followed by the shortest path from v to d. But the first and last terms in the composition are just $A[s, u]$ and $A[v, d]$, so we have

$$A[\mathcal{P}_{s,d\,|\,e}] = A[v, d] \cdot A[e] \cdot A[s, u].$$

All three of the composed functions are monotone and piecewise linear, so Lemma 1.2.4 applies: if two distinct breakpoints in $A[\mathcal{P}_{s,d\,|\,e}]$ are images of primitive breakpoints in $A[e]$, they must be images of distinct breakpoints in $A[e]$. $\square$

Corollary 1.2.6. In every $A[s, d]$ there are at most K primitive images.

The preceding lemma bounds only the number of primitive breakpoints. The minimization breakpoints are harder to analyze, and they occupy much of the rest of the chapter.

1.3 Lower Bound

In this section we prove a superpolynomial lower bound on the number of breakpoints in the arrival time function $A[s, d]$. Our construction uses only linear cost functions $A[e]$ for all the edges, and our analysis is based on a lower bound for parametric shortest paths.

We remind the reader that, in the parametric shortest path problem, edges of a directed graph G have weights that vary linearly with a parameter γ: for $e_i \in E$, $c(e_i) \equiv c_i(\gamma) = a_i\gamma + b_i$. The length of the shortest path from s to d, which we denote by $L[s, d]$ to parallel our notation for time-dependent paths, is a function of γ. Specifically, $L[s, d] = \min(L[p] \mid p \in \mathcal{P}_{s,d})$, the pointwise minimum of $|\mathcal{P}_{s,d}|$ linear functions $L[p]$. Let $B(L[s, d])$ denote the number of breakpoints in this pointwise minimum $L[s, d]$, which is the number of times the shortest path from s to d changes over the domain of γ. The following result is due to Carstensen [20] and Mulmuley and Shah [95].

Theorem 1.3.1 ([20, 95])**.** There exists a layered graph G with n nodes, linear parametric edge weights, and two nodes s and d such that the number of breakpoints of $L[s,d]$ is $B(L[s,d]) = n^{\Omega(\log n)}$.

In a layered graph vertices are arranged in groups, called layers, indexed in increasing order. The first layer consists of the single vertex s and the last consists of the vertex d. All edges of the layered graph connect consecutive layers and are directed from the lower indexed layer to the higher indexed one. A layered graph representation is convenient in reasoning about parametric shortest paths because the addition of a constant weight to all edges between two adjacent layers preserves the relative order of all s–d paths. This will be an important property for our lower bound construction.

The main idea in our lower bound is to transform an instance of the parametric shortest path problem into an instance of time-dependent shortest paths, by treating the parameter γ as the time parameter t. A key difference between the two models is that while the cost of a parametric shortest path is simply the sum of its edge costs for a fixed value of γ, a time-dependent shortest path involves a varying (monotonically growing) time parameter t. In order to map a fixed γ into a range of time parameters without distorting the breakpoints too much, we scale the edge delays down so that γ $(= t)$ does not change too much during the time it takes to travel from s to d. In order to calculate the scaling correctly, we need all the edge weights to be nonnegative. Since linear parametric weights can be negative for some values of γ, we add an appropriately large positive number to all edge weights. The following subsection describes the technical details of our reduction.

1.3.1 The Reduction

Let the set of breakpoints in the worst-case parametric shortest path instance correspond to the parameter values $\gamma_1 < \gamma_2 < \cdots < \gamma_N$, where $N = B(L[s,d])$. We compute the minimum edge weight in the interval $[\gamma_1, \gamma_N]$, namely $c_{\min} = \min_i \min(c_i(\gamma_1), c_i(\gamma_N))$, and then modify the weight of every edge by adding $W = \max(0, -c_{\min})$ to it. In the rest of this section we assume that the parametric costs $c_i(\gamma)$ have been translated up by W, so that $c_i(\gamma) \geq 0$ for $\gamma \in [\gamma_1, \gamma_N]$ and $L[s,d]$ is the path length in the parametric graph. Because the graph is layered, the addition of W to all edge weights preserves the relative order of all s–d paths: the function $L[s,d]$ is just translated upward by W times the number of layers minus one.

Let γ_j and γ_{j+1} be two successive breakpoints of $L[s,d](\gamma)$. Let p_j be the shortest path from s to d during the interval $[\gamma_j, \gamma_{j+1}]$. (Our construction works even if p_j is not unique, i.e., multiple paths have the same length function as p_j, but for simplicity assume p_j is unique.) Because the length of any path in G is a linear function of γ, it must be the case that for $\bar{\gamma}_j = (\gamma_j + \gamma_{j+1})/2$, every path in G not equal to p_j is strictly longer than p_j. Define $\bar{L}_j = L[s,d](\bar{\gamma}_j)$ and let $\bar{L}'_j$ be the length of the second-shortest path at $\gamma = \bar{\gamma}_j$. Define $\Delta_j = \bar{L}'_j - \bar{L}_j$, and $\Delta_{\min} = \min_j \Delta_j$.

The total number of edges in any s–d path is at most $n - 1$. Therefore, if we arbitrarily perturb the weights of edges at $\gamma = \bar{\gamma}_j$ by at most $\Delta_{\min}/(2n) \leq \Delta_j/(2n)$, the path p_j will continue to be the shortest s–d path.

We now compute an ε-interval around $\bar{\gamma}_j$ such that variation of γ within the interval keeps the weight perturbation small. Every cost function $c_i(\gamma) = a_i\gamma + b_i$ has finite slope a_i. If $a_i \neq 0$, we can choose a parameter $\varepsilon_i \equiv \Delta_{\min}/(2na_i)$ so that $|c_i(\bar{\gamma}_j) - c_i(\bar{\gamma}_j + \varepsilon)| < \Delta_{\min}/(2n)$ for any ε with $|\varepsilon| < \varepsilon_i$. If $a_i = 0$, $|c_i(\bar{\gamma}_j) - c_i(\bar{\gamma}_j + \varepsilon)| = 0 < \Delta_{\min}/(2n)$ for every ε, so ε_i can be set to infinity. Define $\varepsilon_{\min} = \min_i \varepsilon_i$. This is a constant that depends only on G and the cost functions $c_i(\gamma)$.

Observation 1.3.2. Given a directed graph with real edge weights, multiplying all edge weights by a constant κ does not change the combinatorial type of any point-to-point shortest path.

Lemma 1.3.3. Given a directed graph G with parametric edge weights, multiplying every edge weight function by a constant κ does not change the values of ε_i and $\varepsilon_{\min}$ defined above.

Proof: If edge weights are multiplied by κ, Δ_j and $\Delta_{\min}$ are also multiplied by κ. The constant that multiplies γ in the weight function $c_i(\gamma)$ changes from a_i to $\kappa \cdot a_i$. The value of ε_i is a ratio with $\Delta_{\min}$ in the numerator and a_i in the denominator, so the two κ's cancel, leaving ε_i unchanged. It follows that $\varepsilon_{\min}$ is also unchanged. $\square$

Let $L_{\max}$ be the maximum value of $L[s,d](\gamma)$ over the finite interval of γ bounded by breakpoints, i.e., $\gamma \in [\gamma_1, \gamma_N]$. We set the constant $\kappa = \frac{1}{2}\varepsilon_{\min}/(L_{\max} + \Delta_{\min})$. By Observation 1.3.2, if we multiply each input cost function by κ, the resulting parametric shortest path problem has the same breakpoints as the original one, and by Lemma 1.3.3 the values of ε_i and $\varepsilon_{\min}$ are unchanged.

We convert the parametric shortest path problem to a time-dependent one by setting the time-dependent delay function $D[e_i] \equiv D_i(t) = \kappa \cdot c_i(t)$, that is, we scale the costs by κ and equate the time t with the γ parameter. (Recall that $A[e_i] = D[e_i] + t$.)

1.3.2 The Analysis

We now show that the shortest s–d path that leaves from s at time $t = \bar{\gamma}_j$ is equal to p_j. We first compute the arrival time $A[s,k](\bar{\gamma}_j)$ for each node k on the path p_j. Without loss of generality assume that the nodes of p_j are numbered sequentially from $s = 1$ to $d \leq n$. (Equivalently, the nodes are numbered by layers in the layered graph.) To relate the time-dependent shortest path to the parametric shortest path, we introduce the shorthand $\sigma_k = \sum_{i<k} D_i(\bar{\gamma}_j) = \kappa \sum_{i<k} c_i(\bar{\gamma}_j)$, which is κ times the length of the $(k-1)$-edge prefix of the parametric shortest path for $\gamma = \bar{\gamma}_j$.

Lemma 1.3.4. The arrival time at the k^{th} node of the path p_j, $A[s,k](\bar{\gamma}_j)$, is at most $\bar{\gamma}_j + \varepsilon_{\min}\frac{k-1}{2n} + \sigma_k$.

Proof: The proof is by induction. The claim is trivial for $k = s = 1$. Assuming that the claim holds for some $k \geq 1$, we prove it for $k+1$. Because $D_k(t)$ is a linear function, the delay for the edge $e_k = (k, k+1)$ is between $D_k(\bar{\gamma}_j)$ and $D_k(\bar{\gamma}_j + \varepsilon_{\min}\frac{k-1}{2n} + \sigma_k)$. Note that $0 \leq \sigma_k \leq \sigma_d \leq \kappa \cdot L_{\max} < \varepsilon_{\min}/2$. Because p_j has at most $n-1$ edges, $k < n$ and hence $\varepsilon_{\min}\frac{k-1}{2n} < \varepsilon_{\min}/2$. This means that the delay for e_k is between $D_k(\bar{\gamma}_j)$ and $D_k(\bar{\gamma}_j + \varepsilon_{\min})$, i.e., the delay differs from $D_k(\bar{\gamma}_j) = \kappa \cdot c_k(\bar{\gamma}_j)$ by at most $\kappa \cdot \Delta_{\min}/(2n) < \varepsilon_{\min}/(4n)$. For convenience define $A_k \equiv A[s,k](\bar{\gamma}_j)$. Because $D_k(t)$ is nonnegative for all $\gamma_1 < t < \gamma_N$, $A_k \leq A_{k+1}$ for all $1 \leq k < n$. Then the arrival time at node $k+1$ is

$$\begin{aligned} A_{k+1} &= A_k + D_k(A_k) \\ &\leq \bar{\gamma}_j + \varepsilon_{\min}\frac{k-1}{2n} + \sigma_k + D_k(\bar{\gamma}_j) + \frac{1}{4n}\varepsilon_{\min} \\ &\leq \bar{\gamma}_j + \frac{k}{2n}\varepsilon_{\min} + \sigma_{k+1}. \end{aligned}$$

□

It follows from the lemma that the arrival time at d is $A[s,d](\bar{\gamma}_j) \leq \bar{\gamma}_j + \varepsilon_{\min}/2 + \sigma_d$. As noted in the proof, $\sigma_d < \varepsilon_{\min}/2$, and so $A[s,d](\bar{\gamma}_j) < \bar{\gamma}_j + \varepsilon_{\min}$. The cost (delay function) of each edge in G varies by at most $\kappa \cdot \Delta_{\min}/(2n)$ during the interval $t \in [\bar{\gamma}_j, \bar{\gamma}_j + \varepsilon_{\min})$, which is the time interval needed to traverse p_j starting at $t = \bar{\gamma}_j$. This means that, as argued above in the context of perturbing parametric shortest paths, p_j is the shortest s–d path for starting time $t = \bar{\gamma}_j$. This is true for each j between 1 and

$N-1$, and so there are at least $N-2$ combinatorial changes to the s–d shortest path in the time-dependent graph between $t=\gamma_1$ and $t=\gamma_N$. We have established the following theorem:

Theorem 1.3.5. There exists a graph G with n nodes and linear edge arrival functions $A[e]$ that contains a pair of nodes s, d such that $B(A[s,d]) = n^{\Omega(\log n)}$.

1.4 Upper Bound

We now show that the number of minimization breakpoints in any arrival time function in an n-node graph is $K \times n^{O(\log n)}$, where K is the total number of linear pieces among all the edges of G. The following crucial observation is key to the proof:

Lemma 1.4.1. Between any two adjacent primitive images, $A[s,d](t)$ forms a concave chain.

Proof: If a minimization breakpoint of $A[s,d]$ occurs at some t_0 where there is no primitive image, then $A[s,d] = A[p_1]$ for $t = t_0 - \varepsilon$ and $A[s,d] = A[p_2]$ for $t = t_0 + \varepsilon$, where $\varepsilon > 0$ is infinitesimal and $p_1, p_2 \in \mathcal{P}_{s,d}$. That is, $A[s,d] = A[p_1 \cup p_2]$ in the neighborhood $|t - t_0| \leq \varepsilon$. Both $A[p_1]$ and $A[p_2]$ are linear in that neighborhood, because neither contains a primitive image, and the two intersect at $t = t_0$. The minimum of two linear functions forms a concave corner at their intersection. Since every corner of $A[s,d]$ between two consecutive primitive images is concave, the lemma is proved. $\square$

A crude estimate that follows from Lemma 1.4.1 is that there can be as many minimization breakpoints between two adjacent primitive images as the number of different slopes in all possible $A[p]$, for $p \in \mathcal{P}_{s,d}$. Unfortunately, this observation does not yield useful upper bounds, since the number of different slopes among all the possible paths from s to d can be of the same order as the number of paths from s to d. As an example, consider the series-parallel graph shown in Figure 1.2 in which each of the $\Theta(n)$ edges e_i can be chosen independently to belong to an s–d path. If $A[e_i]$ is linear with slope equal to the i^{th} prime number, and all other edges e have $A[e](t) = t$, then the number of distinct slopes in all $A[p]$, for $p \in \mathcal{P}_{s,d}$, is $2^{\Theta(n)}$.

To focus on the complexity of $A[s,d]$ between primitive images, we consider the case in which $A[s,d]$ contains no primitive images at all, namely when each $A[e]$ is linear. Let us define $A_{\text{lin}}[s,d]$ to be the minimum arrival time function over all paths $p \in \mathcal{P}_{s,d}$, assuming that every $A[e]$ is linear. Then $B(A_{\text{lin}}[s,d])$ is the maximum number of minimization breakpoints in the s–d arrival time function for linear edge functions in the layered graph.

Lemma 1.4.2. The number of breakpoints in the arrival time function for s–d paths is at most K times the maximum number for the same function assuming linear edge functions:

$$B(A[s,d]) \;\leq\; K \times B(A_{\text{lin}}[s,d]).$$

Proof: Consider an edge $e = (u, v) \in E$. If $e \notin p$ for any $p \in \mathcal{P}_{s,d}$, we can delete it from E: its arrival time function $A[e]$ cannot contribute to $A[s, d]$. Otherwise, consider the restricted path set $\mathcal{P}_{s,d\,|\,e}$, which is the subset of paths in $\mathcal{P}_{s,d}$ that contain edge e, and the corresponding function $A[s, d\,|\,e] = A[v, d] \cdot A[e] \cdot A[s, u]$. If b is a primitive breakpoint of $A[e]$, let b' be its image in $A[s, d\,|\,e]$, that is, $b = A[s, u](b')$. (It is possible that discontinuities may hide a breakpoint, i.e., if $\lim_{t\uparrow b'} A[s, u](t) < b$ and $\lim_{t\downarrow b'} A[s, u](t) > b$, but this does not affect the breakpoint location in $A[s, d]$.)

Let PI be the union, over all edges e, of the images in $A[s, d\,|\,e]$ of the primitive breakpoints of $A[e]$. Then PI is a set of values of t, in the frame of reference of s, the start of all s–d paths. Let us break the function $A[s, d]$ at all $t \in PI$. Some of these values of t are primitive images in $A[s, d]$ and some are not, but all primitive images in $A[s, d]$ belong to PI. Therefore the function $A[s, d]$ is concave between two consecutive elements of PI, by Lemma 1.4.1. Between any two consecutive $b, b' \in PI$, a fixed set of paths $P \subseteq \mathcal{P}_{s,d}$ contributes to $A[s, d]$, i.e., $A[s, d] = \min(A[p] \mid p \in P)$ between b and b'. Let $E_P \subseteq E$ be the set of all edges that belong to paths in P. No edge $e = (u, v) \in E_P$ has a primitive breakpoint in $A[e]$ during the interval $(A[s, u](b), A[s, u](b'))$, i.e., (b, b') mapped to u's frame of reference. Thus we can replace the piecewise linear function $A[e]$ by the linear function that is active during the interval $(A[s, u](b), A[s, u](b'))$ without affecting the value of $A[s, d]$ during the interval $t \in (b, b')$. Likewise we can set $A[e] = \infty$ for all $e \notin E_P$ without affecting the value of $A[s, d]$ during $t \in (b, b')$. This gives a set of linear edge functions such that $A_{\text{lin}}[s, d]$ is equal to the original $A[s, d]$ during $t \in (b, b')$. But $|PI| \leq K$, and so $B(A[s, d]) \leq K \times B(A_{\text{lin}}[s, d])$. □

In the next theorem we bound $B(A_{\text{lin}}[s, d])$, the number of breakpoints in $A[s, d]$ if all $A[e]$ are linear.

Theorem 1.4.3. For any graph G with n nodes and linear arrival time functions $A[e]$ at all edges $e \in E$, $B(A_{\text{lin}}[s, d]) = n^{O(\log n)}$, for any pair of nodes s, d.

Proof: We define $M(k)$ to be the maximum number of linear pieces on the arrival function between any pair of nodes in an n-node graph if shortest paths are restricted to use at most k edges. Clearly, $B(A_{\text{lin}}[s, d]) < M(n - 1)$ as no path has more than $n - 1$ edges, and the number of breakpoints differs by only one from the number of linear pieces for a continuous piecewise linear function.

We prove that $M(k) \leq (2n)^{\lceil \log k \rceil}$. For purposes of this proof, we use base-2 logarithms: $\log \equiv \log_2$. Thus if k is a power of two, then the bound is

$$M(k) \leq (2n)^{\lceil \log k \rceil} = (2n)^{\log k} = k\, n^{\log k} = k^{1+\log n}.$$

The proof is by induction on $p = \lceil \log k \rceil$.

The base of the induction is $k = 1$ and $p = 0$. In that case $M(1) = 1$, because there can be at most one single-edge path from s to d, represented by one linear piece in the arrival function. But $(2n)^{\lceil \log k \rceil} = (2n)^0 = 1$, and so we have $M(1) = 1 \leq (2n)^{\lceil \log k \rceil}$.

Assuming that the claim holds for $k \le 2^p$, we now prove it for $k \le 2^{p+1}$. Any u–v shortest path with at most 2^{p+1} edges can be decomposed into two shortest paths, each with at most 2^p edges, passing through some middle vertex w. Consider the path set $\mathcal{P}_{u,v\,|\,w}$ for which w is the middle vertex. The arrival time function corresponding to $\mathcal{P}_{u,v\,|\,w}$ is the composition of arrival functions corresponding to $\mathcal{P}_{u,w}$ and $\mathcal{P}_{w,v}$, each containing paths using at most 2^p edges, hence each with at most $M(2^p)$ breakpoints. Therefore the number of breakpoints in the arrival time function corresponding to $\mathcal{P}_{u,v\,|\,w}$ is at most $2M(2^p)$ by Lemma 1.2.4. The set of u–v paths with at most 2^{p+1} edges is the union of $\mathcal{P}_{u,v\,|\,w}$ over all n possible middle vertices w, with the subpaths before and after w limited to at most 2^p edges apiece. Therefore, its corresponding arrival time function has at most $2n \cdot M(2^p)$ breakpoints. By the induction hypothesis, $M(2^p) \le (2n)^{\lceil \log 2^p \rceil} = (2n)^p$, and so

$$\begin{aligned} M(2^{p+1}) &\le 2n \cdot M(2^p). \\ &\le 2n \cdot (2n)^p \\ &= (2n)^{p+1} \\ &= (2n)^{\lceil \log 2^{p+1} \rceil}. \end{aligned}$$

Evaluating $M(k)$ for $k = n - 1$ completes the proof of the lemma. □

Combining Lemma 1.4.2 with Theorem 1.4.3, we obtain the following upper bound:

Theorem 1.4.4. For any graph G with n nodes, if the arrival time functions $A[e]$ at all edges $e \in E$ are piecewise linear with a total of K segments over all e, then $B(A[s,d]) = K \times n^{O(\log n)}$, for any pair of nodes s, d. If K is polynomial in n, then $B(A[s,d]) = n^{O(\log n)}$.

1.5 Algorithmic Results

The lower bound of Section 1.3 can be disheartening to anyone interested in computing the arrival time functions in a time-dependent graph. While one may hope that practical graphs are unlikely to exhibit such complexity, investigating the source of the complexity remains a challenging problem. In order to understand the scope of our lower bound, we can investigate important practical subclasses of graphs or design approximation schemes with provable bounds. We study four questions along these directions. First, we show that the minimum delay over a departure time query interval can be computed in polynomial time. This is somewhat surprising since the delay function over that time interval may have superpolynomial complexity. Second, we present an approximation scheme that constructs a small-size approximation of the travel time function. Third, we discuss output-sensitive algorithms to compute the arrival time function exactly. Finally, we show that the complexity of the arrival time function is much more well-behaved if we limit the *slopes* of the edge functions.

Fixed Time Shortest Path Probes In the next two section we will make extensive use of the concept of shortest path probes. A forward probe consists of running an instance of the shortest path algorithm (e.g., Dijkstra) for a given departure time t to determine the corresponding arrival time $A[s, d](t)$. Similarly, a *reverse probe* runs the shortest path algorithm on the reversed graph G^r obtained by reversing all edges of G and inverting the arrival time function for each edge[2], starting from a value A and determining the time t such that $A[s, d](t) = A$.

We measure the time complexity of our algorithms in terms of the number of calls to these shortest path algorithm probes.

1.5.1 Minimum Delay Queries

One important application of time-dependent shortest paths is finding the minimum delay for a departure time in a given query window. Let $D(t) = D[s, d](t) = A[s, d](t) - t$ be the *delay function* for the (s, d) node pair. Because the delay function is concave between primitive breakpoints, the minimum occurs at those breakpoints or at the endpoints of the query interval. This simple observation allows us to find the answer to minimum delay queries in polynomial time.

[2]Discontinuities in $A[e]$ map to horizontal edges in $A[e^r]$ and vice versa. For example, if an edge e has a discontinuous function $A[e]$, i.e., $A[e](\tau^-) = x$ and $A[e](\tau^+) = y$ with $x < y$, then in $A[e^r]$ each time in the range $[x, y]$ maps to τ: when traveling from s, one must arrive at edge e at or before $t = \tau$ in order to reach the other end of e at or before any time in the range $[x, y]$.

Theorem 1.5.1. For any query interval $[t_1, t_2]$, the minimum delay path can be found using $O(K)$ shortest path probes.

Proof: For each edge $e = (u, v)$ with a breakpoint at time b in u's frame of reference, we project b back to the temporal frame of reference of the source s. To do this we use reverse shortest path probes from u to s, computing the time b' in s's frame of reference at which one has to depart in order to hit the breakpoint b on edge e. This is the primitive image in $A[s, d]$, if any, that is the image of b. The union of all these breakpoint images, denoted PI, is a superset of the primitive images of $A[s, d]$; between any two consecutive times in PI, $A[s, d]$ is concave.

We perform forward probes at t_1, t_2, and all elements of PI inside $[t_1, t_2]$. This gives the corresponding values of $A[s, d]$, and hence the values of $D()$. (The values for elements of PI can be precomputed, if desired, or determined on the fly.) Because the function $D(t)$ is concave between the sampled t values, the minimum delay in $[t_1, t_2]$ is the minimum of the computed $D()$ values. $\square$

1.5.2 $(1 + \varepsilon)$-Approximation of the Delay Function

Although the worst-case complexity of $A[s, d]$ is $n^{\Theta(\log n)}$, in practice one may be satisfied with a lower-complexity approximation of the arrival time function. To quantify the approximation error, we focus on the delay function $D(t) = A[s, d](t) - t$. A function $D'(t)$ is a $(1 + \varepsilon)$-approximation of $D(t)$ if $|D(t) - D'(t)| \leq \varepsilon \cdot D(t)$.

As noted in Section 1.5.1, the function $D(t)$ is concave between any two consecutive primitive breakpoints. Since the number of primitive breakpoints is only $O(K)$, we can reduce the problem of approximating $D(t)$ to $O(K)$ approximation problems for concave chains. While we do not have access to the concave chains explicitly, we can *query* them efficiently, using shortest path probes.

The Approximate Representation

The first step of the approximation is to break $A[s, d]$ (and hence $D[s, d]$) at all primitive images. Instead of trying to identify exactly those primitive breakpoints with images in $A[s, d]$, we simply break $A[s, d]$ at the images of all primitive breakpoints, even those that do not propagate into $A[s, d]$.

As in the proof of Theorem 1.5.1, we use reverse probes to find a set PI of time values that is a superset of the primitive images of $A[s, d]$. For each pair of consecutive breakpoints $b, b' \in PI$, we use forward probes to find the corresponding values of $A[s, d]$ (and hence the values $D(b)$ and $D(b')$). The function $D(t)$ is concave between b and b'; that is, for every t_1, t_2, t_3 with $b < t_1 < t_2 < t_3 < b'$, the point $(t_2, D(t_2))$ lies on or above the line segment from $(t_1, D(t_1))$ to $(t_3, D(t_3))$. The minimum value of $D(t)$ in $[b, b']$ is $D_{\min} \equiv \min(D(b), D(b'))$, and the maximum value $D_{\max}$ is between $D(\bar{b})$ and $2D(\bar{b})$, where $\bar{b} = (b + b')/2$.

A conceptually simple approximation for $D(t)$ slices the function horizontally, obtaining the following result.

Lemma 1.5.2. If the edge arrival time functions have a total of K linear pieces and the maximum value of $D_{\max}/D_{\min}$ over all the concave intervals of $D[s,d]$ is at most R, then there exists a data structure of size $O(K\frac{1}{\varepsilon}\log R)$ that represents $D[s,d]$ with relative error ε.

Proof: For each of the $O(K)$ chains, we find the intersections of $D(t)$ with the lines $y = (1+\varepsilon)^k D_{\min}$, for each $k \geq 0$ such that $(1+\varepsilon)^k D_{\min} \leq D_{\max}$. Let the resulting sample points be $(t_i, D(t_i))$. Because $D(t)$ is concave in $[b, b']$, each slice generates at most two sample points, so the total number of samples is at most $2\log(D_{\max}/D_{\min})/\log(1+\varepsilon) \approx \frac{2}{\varepsilon}\log(D_{\max}/D_{\min})$. Linearly interpolating between consecutive sample points $(t_i, D(t_i))$ gives an approximation that is never greater than $D(t)$; the lower envelope of the supporting lines at all the sample points is a concave curve that is never less than $D(t)$. The worst-case error of either approximation is at most ε times the actual value of $D(t)$, giving a $(1+\varepsilon)$-approximation, as desired. $\square$

Computing the Representation

Finally, we discuss the time complexity of computing the $D(t)$ samples needed by our procedure. It is not possible to perform the horizontal slicing of $D(t)$ directly, because shortest path probes have access to $A[s,d](t)$, not to $D(t)$. However, we observe that we can obtain an adequate approximation without precisely locating the t_i such that $D(t_i) = (1+\varepsilon)^i D_{\min}$ for every $i \leq 2\log(D_{\max}/D_{\min})/\log(1+\varepsilon)$. It suffices to find approximate sample points within a finer partition of the range. That is, we imagine slicing $D(t)$ into fragments with the horizontal lines $y = (1+\varepsilon)^{k/2} D_{\min}$ (using twice as many values of k), and then find a sample point on each fragment. The values of $D(t)$ at consecutive sample points differ by at most a factor of $1+\varepsilon$, so this sampling satisfies the criteria needed to prove Lemma 1.5.2.

We consider the problem of finding approximate samples for a single concave chain delimited by two consecutive primitive breakpoint images b and b'. Let the horizontal span of the ith chain be denoted $I_i = b' - b$; the vertical range is bounded between $D_{\min}$ and $D_{\max}$. The total horizontal span of all K chains is defined as $I = \sum_{1 \le i \le K} I_i$. Note that the slope of $D(t)$ is monotone decreasing inside the interval (b, b'), and by the FIFO assumption, the slope of $D(t)$ at $t = b'$ is at least -1.

We use a combination of forward and reverse shortest path probes. Note that reverse probes have a rather loose connection to $D(t)$, since the value of t is not known *a priori*. We claim the following result.

Theorem 1.5.3. The total number of shortest path probes needed to compute the approximation of Lemma 1.5.2 is $O\left(K \frac{1}{\varepsilon} \log\left(\frac{D_{\max}}{D_{\min}}\right) \log\left(\frac{I}{K\varepsilon D_{\min}}\right)\right)$.

Proof: Consider the ith chain. We search for sample points on $D(t)$ using reverse probes on a prefix of $[b, b']$ and use forward probes on the remainder. Starting from $t = b$, so long as the slope of $D(t)$ is at least 1, we perform a reverse probe for $A = t + D(t) \cdot \sqrt{1+\varepsilon}$, then set t to the resulting time value t'. Because $t' \ge t$, we have

$D(t') = A(t') - t' \leq A(t') - t = D(t) \cdot \sqrt{1+\varepsilon}$; that is, $(t', D(t'))$ is in the current fragment or the next one. But if the slope of $D(t')$ is at least 1, then $A[s,d]$ has slope at least 2 between t and t', implying that $A = t + D(t)\sqrt{1+\varepsilon} \geq t + D(t) + 2(t' - t)$. Thus $t' \leq \frac{1}{2}D(t)(\sqrt{1+\varepsilon} - 1) + t$, which implies that $D(t') = t + D(t)\sqrt{1+\varepsilon} - t' \geq \frac{1}{2}D(t)(\sqrt{1+\varepsilon} + 1)$. But $(\sqrt{1+\varepsilon}+1)/2$ is a constant fraction of $\sqrt{1+\varepsilon}$, so this procedure performs a constant number of reverse probes in each fragment.

Once the reverse probes produce an interval $[t, b']$ with initial slope at most 1, we use bisection with forward probes to find the rest of the sample points. Observe that if the slopes of $D(t)$ at the ends of an interval of length ℓ differ by Δ, then the maximum possible vertical error in the interval is bounded by $\ell\Delta$. Suppose that the current interval has length ℓ, slope difference Δ, and minimum $D(t)$ value $\overline{D}$ (obtained at an interval endpoint). If $\ell\Delta > \overline{D}(1+\varepsilon)$, then we bisect the interval at its middle t value with a forward probe and recursively partition the two subintervals. This gives a worst-case approximation error of at most $1+\varepsilon$ between consecutive samples.

We can bound the number of bisection probes by considering the binary recursion tree generated by the bisection process described. The leaves of the tree correspond to the final intervals; the internal nodes correspond to the bisections. Each bisection halves the current interval. After k bisections, the current subinterval has length $I_i/2^k$, and since $\Delta \leq 2$ the maximum error in this interval is at most $2I_i/2^k$. This implies that the height of the recursion tree is at most $\log(2I_i/(\varepsilon D_{\min}))$. The internal nodes of maximal depth correspond to intervals with $\ell\Delta > \overline{D}(1+\varepsilon)$, therefore in *any* $(1+\varepsilon)$-approximation of the initial interval, at least one point must come from each such interval. The overhead cost of finding each approximation point using the bisection algorithm (Lemma 1.5.2 states that there are $O(\frac{1}{\varepsilon}\log(D_{\max}/D_{\min}))$ points in the worst case) is then the number of bisections on a root-to-leaf path of the bisection tree for a single chain. This is bounded by the height of the recursion tree. The log-sum inequality extends the result for K chains, since $\sum_{1\le i\le K}\log(I_i/\varepsilon D_{\min}) \le K\log(\frac{I}{K\varepsilon D_{\min}})$. This gives the claimed bound and concludes the proof. $\square$

1.5.3 Output-Sensitive Algorithms

This section describes two output-sensitive algorithms for computing the arrival time function. We give both algorithms because their running times are strictly incomparable.

Result-Directed Probing

An algorithm similar to the approximation described in Section 1.5.2 can be used to compute the arrival function $A[s,d]$ in an output-sensitive manner. That is, the time complexity is proportional to $B(A[s,d])$. Such a scheme is described in [30] and has complexity $O((K + B(A[s,d]))(|E| + |V|\log|V|)$ if a standard Dijkstra shortest path algorithm is used as a probe. To approximate a chain, the algorithm described in [30] starts from finding the endpoints of the chain, together with the linear segments incident to them belonging to the chain. Then, the intersection between the lines supporting the two segments is computed as a point (t_0, y_0) and a forward shortest path probe is made for the starting time t_0 at which the intersection occurs. If the probe returns a value $A[s,d](t_0) < y_0$ then a new segment of the chain has been discovered, and the algorithm recurses to the left and to the right of the new segment. Otherwise, if the probe returns a value $A[s,d](t_0) = y_0$, meaning that a segment already discovered has been encountered, the recursion stops. Since each segment can be encountered in at most two shortest path probes, the total number of probes made is proportional to the number of segments on the chain.

Kinetic Data Structures

An alternative output-sensitive algorithm is based on the paradigm of kinetic data structures [13, 63]. This algorithm builds a data structure that represents the execution of Dijkstra's algorithm for a fixed departure time. The algorithm then continuously varies the departure time, all the while updating the data structure to represent the execution of Dijkstra's algorithm for the varying departure time. As the algorithm varies t, it produces a representation of $A[s,u](t)$ for every node u in the graph, and hence it computes $A[s,d](t)$. The running time of this algorithm, described below is strictly incomparable with that of the algorithm of [30].

The idea is to run Dijkstra's algorithm for a particular departure time t and build a set of *certificates* (predicates that are linear functions of t) that guarantee the correctness of the shortest path tree that Dijkstra's algorithm computes. We then vary the departure time t while maintaining the shortest path tree and a set of certificates that guarantee its correctness. This approach produces all the combinatorially distinct shortest path trees that exist for all values of t, producing each one as a modification of a similar, temporally adjacent neighbor.

Certificates are of two types, *primitive certificates* and *minimization certificates*. Primitive certificates ensure that the linear function corresponding to a single path remains unchanged until the failure time of the certificate; the failure of a primitive certificate corresponds to a primitive breakpoint at some edge. Minimization certificates ensure that a certain path reaches a given node first; failure of a minimization certificate corresponds to a minimization breakpoint in the graph G. Each certificate has a failure time t_{fail} that is the earliest departure time after the current time t for which the certificate fails.

The certificates are stored in a priority queue, with the one with the earliest failure time at the head. When a certificate fails, at least one node or edge in G has a breakpoint, the combinatorial shortest path from s to one or more nodes may change, and some number of certificates will need to be updated. (Real kinetic data structures spend considerable effort to make sure that few certificates need to be updated in response to any event; the structure described here does not.)

For every node $v \in V$, we create a fixed binary tree whose leaves are the incoming edges (u, v). This tree represents a tournament among the incoming edges to determine which edge carries the path that first reaches v from s for any given departure time. For simplicity, we may modify G by expanding each node v with in-degree d into a binary tree with $d-1$ nodes, height $\lceil \log_2 d \rceil$, and all internal edges with zero delay. This gives a graph G' with $\Theta(m)$ nodes, each with in-degree at most two. We are now ready to claim our result.

Theorem 1.5.4. There is an algorithm that computes $A[s,v](t)$ for $t \in [t_0, t_1]$ in time $O(\log^2 n)$ times the total number of breakpoints of $A[s,v \,|\, e]$ over all edges $e = (u,v) \in G$.

Proof: We run Dijkstra's algorithm on G', starting from node s at time t_0. Dijkstra computes a linear function at each node v that represents $A[s,v]$ for the time interval containing t_0. Suppose that v has two predecessors u_1 and u_2, and the shortest path tree for departure time t_0 contains $e_1 = (u_1, v)$ but not $e_2 = (u_2, v)$. The correctness of Dijkstra's algorithm means that $A[s,v \,|\, e_1](t_0) < A[s,v \,|\, e_2](t_0)$, that is, the shortest path leaving s at time t_0 and reaching v via edge e_1 arrives before the shortest path with the same departure time and reaching v via e_2. We turn this into a minimization certificate by computing the time $t_{\min} > t_0$ (if any) such that the linear function $A[s,v \,|\, e_1]$ becomes equal to $A[s,v \,|\, e_2]$; the failure time of the certificate is $t_{\min}$, or ∞ if the two linear functions do not intersect after t_0. A primitive certificate for each edge $e = (u,v)$ is computed by using the inverse of the linear function $A[s,u]$ to project the next primitive breakpoint of $A[e]$ (if any) into the time domain of s; the resulting time t_e is the failure time of the certificate.

The primitive and minimization certificates of the nodes and edges of G' are placed in a priority queue ordered by the next certificate failure time. When the first certificate fails at some time t_{fail}, the linear function $A[s, v]$ changes for some set of nodes v. If a minimization certificate fails at v (the shortest path using (u_1, v) becomes more expensive than the one using (u_2, v)), then the shortest path tree changes at v. At the same time the functional form of $A[s, v]$ changes, along with $A[s, w]$ for all shortest-path-tree descendants w of v in G'. Likewise, if the primitive certificate for an edge (u, v) in the shortest path tree changes, the functional form of $A[s, \cdot]$ for v and all its shortest-path-tree descendants changes. If the primitive certificate for an edge (u, v) *not* in the shortest path tree changes, then the minimization certificate at v needs to have its failure time updated, but no other changes are required. In all cases, if the functional form of $A[s, v \mid e]$ changes for any edge $e = (u, v)$, then the minimization certificate at v needs to have its failure time updated.

When the linear function $A[s, v]$ changes for a node v but not for its predecessor u in the shortest path tree, then we need to update certificates for all shortest-path-tree descendants of v. We traverse the subtree rooted at v, updating the functional form of $A[s, w]$ for each descendant w. We recompute the failure times of the primitive certificates for edges reachable from v and the minimization certificates at nodes reachable from v and at the heads of non-tree edges reachable from v. We update the priority queue of certificates to account for new failure times. At the end of this procedure the shortest path tree is correct for paths leaving s at time $t = t_{\text{fail}} + \varepsilon$ for an infinitesimal ε, the functional form of $A[s, v]$ is correct for each node v and departure time $t = t_{\text{fail}} + \varepsilon$, and the priority queue identifies the next time greater than t_{fail} at which a change to one or more arrival time functions will occur. The total number of certificates is $O(m)$—at most one per node and one per edge. The time to respond to the failure of a certificate c is $O(|T_c| \log n)$, where T_c is the set of edges $e = (u, v)$ of G' whose arrival time functions $A[s, v \mid e]$ change due to the failure of c, and the logarithmic factor comes from priority queue operations. The edge set T_c consists of the shortest path subtree in G' rooted at the node or edge where c occurs, augmented with the non-tree edges directed out of that subtree. In the worst case $|T_c|$ may be $\Theta(m)$, but in cases when it is smaller, the output-sensitive algorithm exploits that fact. Expressed in terms of the original graph G, $|T_c|$ is equal to the number of outgoing edges reachable from the certificate node/edge's subtree, times a factor of at most $O(\log n)$ corresponding to the height of the tournament trees in which those edges participate in G'.

The worst-case running time of the algorithm is $O(\log^2 n)$ times the total number of breakpoints of $A[s, v \mid e]$ over all edges $e = (u, v) \in G$. $\square$

The bound achieved by the preceding output-sensitive algorithm is incomparable with that of [30]. It requires only $O(\log^2 n)$ time per breakpoint, but processes all the breakpoints in $A[s, v \,|\, e]$ over all $e = (u, v) \in G$, whereas [30] executes a full shortest path algorithm for each breakpoint, requiring $O(n \log n + m)$ time, but processes only those breakpoints that appear in $A[s, d]$. Depending on the relative sizes of these arrival functions, either algorithm may outperform the other.

1.5.4 Restricted Slopes

One of the simplest cases of time-dependent shortest paths arises when the slopes of the edge arrival time functions are restricted to 0, 1 and ∞: slope 1 implies constant speed travel, slope ∞ models a temporary complete stop of traffic flow (e.g., due to sudden congestion or a traffic signal), and slope 0 models release of the flow as the congestion clears. With these limited edge functions, the arrival function $A[s, d]$ does not suffer the superpolynomial complexity of the general case. In fact, the complexity of $A[s, d]$ is only $O(K)$. This fact was already noted in [30]. More generally, we show that the complexity is low (in fact, linear) if the edge functions have slopes in the set $\{0, \alpha^{-\beta}, \alpha^{-\beta+1}, \ldots, \alpha^{\beta-1}, \alpha^{\beta}\}$, for some $\alpha > 1$ and $\beta \in \mathbb{N}$. That is, each edge has a piecewise linear cost function and each piece has the form $A(t) = at + b$, with slope $a \in \{0, \alpha^{-\beta}, \alpha^{-\beta+1}, \ldots, \alpha^{\beta}\}$. (The special slope ∞ may also be included without altering the complexity.) We call such a graph an (α, β)-*slope graph*. We have the following theorem.

Theorem 1.5.5. In (α, β)-slope graph, any $A[s, d]$ has at most $2(\beta + 1)nK$ breakpoints.

Proof: The proof follows from the observation that the slope of a path arrival function along a path $p \in \mathcal{P}_{s,d}$ is the product of the slopes of the edge arrival functions $A[e]$ on the edges composing p. Since a path has at most $n-1$ edges, there can be at most $2(\beta+1)(n-1)$ possible distinct slopes appearing in any path arrival function. Between any two primitive images, $A[s,d]$ forms a concave chain (Lemma 1.4.1), and hence no two segments with the same slope can appear in it. This concludes the proof. $\square$

1.5.5 Average and Smoothed Bounds

In Section 1.3 we showed that the *worst case* complexity of $A[s,d]$ can be superpolynomial. However, when considering random cost functions drawn from a given distribution, it might still be the case that polynomial average complexity bounds could be provided. This is precisely the case for the parametric shortest path problem. Consider an instance of the parametric shortest path problem with edge costs in the form $a_e + \lambda b_e$ and define $\overrightarrow{a} = a_1, \ldots, a_m$ and $\overrightarrow{b} = b_1, \ldots, b_m$ the vectors of size $m = |E|$, then any path of the instance can be represented as $\overrightarrow{a} + \lambda \overrightarrow{b}$. In [102] the following average and smoothed bounds are given:

- if $\overrightarrow{a}$ and $\overrightarrow{b}$ are uniform random vectors in R^m then the expected number of points in the convex hull of all possible $\overrightarrow{a}$ and $\overrightarrow{b}$ is $O(m)$

- if $\overrightarrow{a}$ and $\overrightarrow{b}$ are ρ-perturbations (see Kelner and Spielman [73] for smoothed analysis reference) with $\rho = \frac{1}{\sqrt{2m}}$ of two given vectors $\overrightarrow{a'}, \overrightarrow{b'} \in R^m$ of points in the convex hull of all possible $\overrightarrow{a}$ and $\overrightarrow{b}$ is $O(m)$

The above results imply that the optimal frontier in the parametric shortest path problem (correspondent to $A[s,d]$ in the time-dependent shortest path problem) has only a polynomial (linear) number of breakpoints on average, for random input costs drawn from the distributions considered above. It would be desirable that similar polynomial bounds could be extended to the time-dependent shortest path problem considered in this chapter. However, the analysis provided in [102] heavily relies on the edge-separability path cost of the parametric shortest path problem. That is, the cost of a path in the parametric shortest path problem is just the sum of the costs on the edges along the path. On the other hand, In the time-dependent shortest path problem the path costs are not edge separable. In fact, if the cost of a path is $c + dt$ then c does not depends only on the constant terms of the edge costs on the path, but depends on the slopes as well, as it follows by cost composition along a path. For this reason, the results of [102] do not extent to the time-dependent shortest path problem considered here.

1.6 Closing Remarks

In this chapter we resolved a conjecture regarding time-dependent shortest paths, and showed that even with linear cost functions and FIFO edges, the arrival time function at a node in an n-node graph can have complexity $n^{\Omega(\log n)}$. We also presented an upper bound of $Kn^{O(\log n)}$ if the total number of linear pieces over all the piecewise linear edge cost functions is at most K. If K is polynomial in n, the arrival time function's complexity is $n^{\Theta(\log n)}$. Nevertheless, and somewhat surprisingly, we presented a polynomial-time algorithm to find the departure time in a query interval that minimizes delay.

The arrival time functions for time-dependent and parametric shortest paths have equivalent complexity if the edge cost functions are (piecewise) linear, but their behavior diverges for higher-order edge cost functions. In particular, if the edge costs are represented by polynomials of degree d, then the arrival time function for the parametric shortest path remains a polynomial of degree d, while the time-dependent shortest path arrival function may have degree d^n.

Several open problems naturally arise from our work: Are there natural but rich classes of graphs for which the arrival time complexity remains polynomial? Results in [52] imply that the complexity is polynomially bounded in graphs of constant treewidth, but no other results on minor-excluded classes are known. For instance, what is the complexity for planar graphs? Our lower bound proof easily extends to *constant degree* graphs, so sparsity alone cannot lead to a better bound. We showed that if the slopes of the cost functions belong to a restricted class (powers of α), then the complexity is polynomial. What other natural classes of linear functions lead to similarly reduced complexity? Another natural direction is to investigate the smoothed complexity: does a small random perturbation of the edge arrival functions lead to small expected complexity of the overall arrival time function?

Acknowledgements

We would like to thank the anonymous reviewer for pointing out a simpler proof to Theorem 1.4.4 than the one appeared in the preliminary version of this paper.

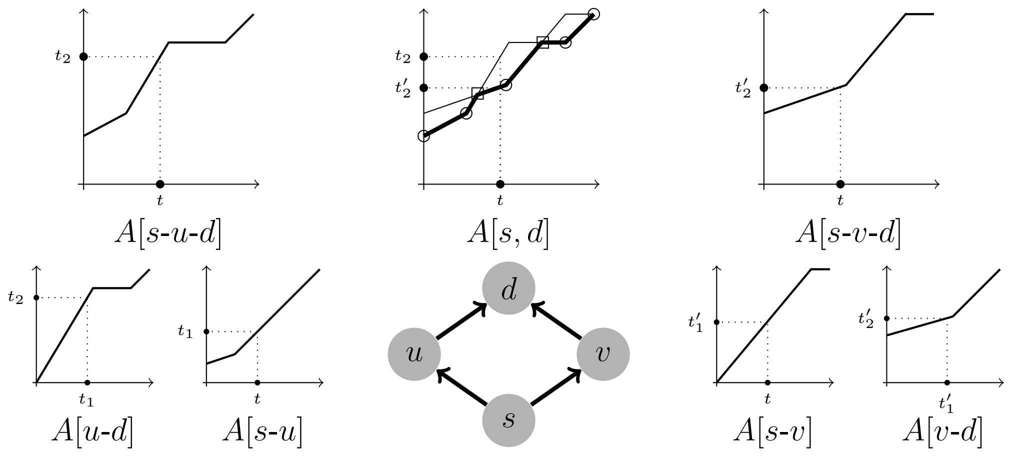


Figure 1.1: Illustration of primitive and minimization breakpoints. The four subfigures at the bottom show the arrival time functions for the four edges in the graph. The figures on the top show arrival time functions for the path (s, u, d) on the left, the path (s, v, d) on the right, and the result of their pointwise minimization at d, namely, $A[s, d]$. The figures also show the progression of time: when following the path (s, u, d), leaving at time t we reach u at time t_1, and reach d at time t_2. When following the path (s, v, d), the arrival times are t_1' at v and t_2' at d. Of these two paths, the latter is quicker for departure time t, as shown in $A[s, d]$. The subfigure for $A[s, d]$ also highlights primitive images (drawn as circles) and minimization breakpoints (drawn as squares).

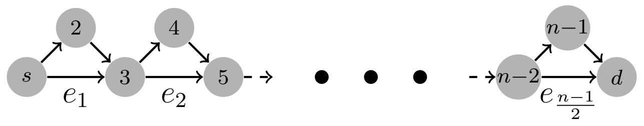


Figure 1.2: Each e_i can be chosen independently for membership in a s–d path.

Chapter 2

Balanced Graph Partitioning*

2.1 Introduction

In this chapter we study the k-`BALANCED PARTITIONING` problem. The problem asks for a partition of the n vertices of a graph into k sets of size at most $\lceil n/k \rceil$ each. At the same time the total number of edges connecting vertices in different sets, called the *cut cost*, needs to be minimised. k-`BALANCED PARTITIONING` is a notoriously hard problem. The special case of $k = 2$, commonly known as the `BISECTION` problem, is already NP-hard [57]. For this reason, approximation algorithms that find a balanced partition with a cut cost larger than optimal have been developed. We follow the convention of denoting the approximation ratio on the cut cost by α.

*Parts of this chapter appeared in the following publications: [47, 48]

Unfortunately, when k is not constant even finding an approximation of the minimum balanced cut still remains infeasible. In fact, no finite approximation for the cut cost can be computed in polynomial time, unless P=NP [8]. In order to circumvent the hardness results, researchers have investigated algorithms that compute solutions with relaxed balance constraints. By that we mean that the sets of the partitions are allowed to be of size at most $(1+\varepsilon)\lceil n/k \rceil$ for some factor $\varepsilon > 0$. In most cases the algorithms proposed are *bicriteria approximations*, which approximate both the balance and the cut cost of the optimal solution. The approximation ratio achieved by these algorithms is measured by comparing the cut cost of the computed solution in which the balance constraint is relaxed, with the optimal cut cost of a *perfectly balanced* partition. That is, one in which each set has size at most $\lceil n/k \rceil$.

The k-`BALANCED PARTITIONING` problem has received some attention for the case $\varepsilon = 1$. This means approximating the perfect balance within a factor of two. For this case, the best result is by Krauthgamer *et al.* [83] who give an algorithm with approximation factor $\alpha \in \mathcal{O}(\sqrt{\log n \log k})$ on the cut cost. However, it is not hard to imagine how the slack on the balance can be unattractive for practical applications. In parallel-computing, for instance, a factor of two on the balance in the workload assigned to each machine can result in a factor of two slowdown. This is because the completion time is solely determined by the machines with the highest workload. Therefore the case when ε can be chosen arbitrarily between 0 and 1, is of greater practical interest. We call

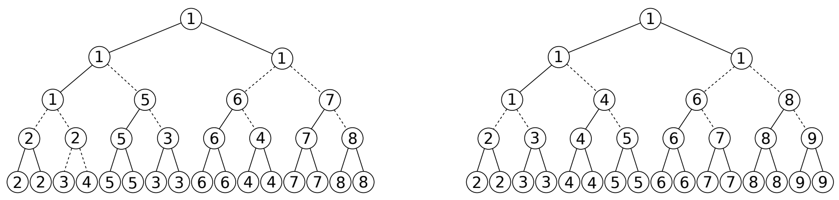


Figure 2.3: Two optimally partitioned binary trees. For the tree on the left $k = 8$ (with a cut cost of 10) whereas $k = 9$ (with a cut cost of 8) for the tree on the right. The numbers in the vertices indicate the set they belong to and the cut edges are dashed.

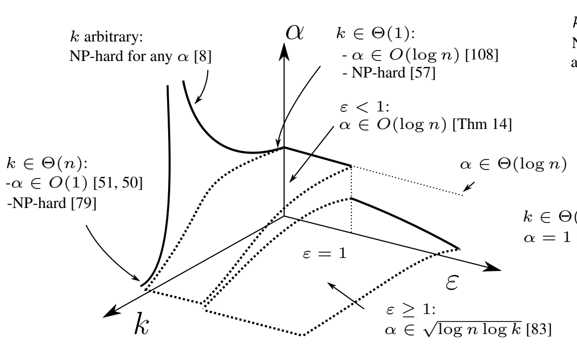
the resulting partitions *near-balanced*. No progress has been made on near-balanced partitions since Andreev and Räcke [8] gave an algorithm with $\alpha \in \mathcal{O}(\log^{1.5}(n)/\varepsilon^2)$—a significantly worse bound than the one for $\varepsilon = 1$.

As argued in [8], it is not surprising that the achieved ratio α is worse for near-balanced solutions than it is in the case $\varepsilon = 1$. This is because typically the algorithms for $\varepsilon = 1$ work in two phases. First the graph is partitioned into components of size at most $2\lceil n/k \rceil$ while minimizing the cut cost. Then the components are packed into k bins. The fact that $\varepsilon = 1$ implies that any such returned components can be packed into k bins using a greedy algorithm. However this approach cannot be used to find near-balanced solutions. In fact, as ε approaches 0 and the constraint on the balance becomes more stringent, the cutting phase must be adapted to return components that can later be packed into bins of the required size. This fact significantly complicates the cutting phase.

2.1.1 Our Contribution

As argued above, the restriction to near-balanced partitions poses a major challenge in understanding the k-BALANCED PARTITIONING problem. For this reason, we consider the simplest non-trivial instance class of the problem, namely connected trees. Even in this simple case the structure of the solution is not easily understood. Figure 2.3 gives an example of how balanced partitions exhibit a counter-intuitive behavior even on *perfect binary trees*, as increasing k does not necessarily entail a larger cut cost. Our results confirm this intuition when a perfectly balanced solution is required. Adapting an argument by Andreev and Räcke [8], we show that it is NP-hard to approximate the cut cost within $\alpha = n^c$ for any constant $c < 1$. This is asymptotically tight, since a trivial approximation algorithm can achieve a ratio of n by cutting all edges of the tree. Interestingly, the lower bound remains true even if the diameter of the tree (i.e. the length of the longest path between any two leaves) is equal to 4. Instances of diameter at most 3 on the other hand, are polynomially solvable.

By a substantially different argument, we show that a similar dichotomy arises when parametrizing the complexity with the maximum degree Δ. For trees with $\Delta = 2$ (i.e. paths) k-BALANCED PARTITIONING is trivial. However, if $\Delta = 5$ the problem becomes NP-hard and with $\Delta = 7$ we show it is APX-hard. Finding where exactly the dichotomy arises, i.e. the $\Delta \in \{3, 4\}$ at which k-BALANCED PARTITIONING


k arbitrary:
NP-hard for any α [8]
α
$k \in \Theta(1)$:
- $\alpha \in O(\log n)$ [108]
- NP-hard [57]
$\varepsilon < 1$:
$\alpha \in O(\log n)$ [Thm 14]
$k \in \Theta(n)$:
-$\alpha \in O(1)$ [51, 50]
-NP-hard [79]
$\alpha \in \Theta(\log n)$
$\varepsilon = 1$
ε
k
$\varepsilon \geq 1$:
$\alpha \in \sqrt{\log n \log k}$ [83]

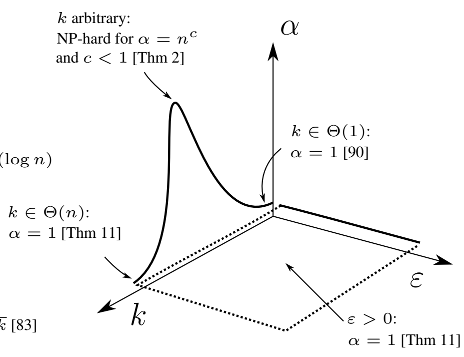


Figure 2.4: Illustrations of best approximation factor α known against k and ε, for general graphs (left) and trees (right). The plane (α, k) represents the case of perfectly balanced solutions $(\varepsilon = 0)$ and shows that the restriction to trees does not significantly change the asymptotic behavior. However, for $\varepsilon > 0$ much better approximations can be devised for trees. This remarkable behavior can be partially adapted to general graphs via tree decompositions, allowing us to reduce the gap between the case $\varepsilon < 1$ and $\varepsilon = 1$ visible in the plot on the left.

becomes hard, is an interesting open problem. These results should be contrasted with a greedy algorithm by MacGregor [90] that can be modified to find perfectly balanced partitions for k sets with $\alpha \in \mathcal{O}(\log(n/k))$, for trees of constant degrees.

On the positive side, we present approximation algorithms that compute near-balanced partitions for edge-weighted graphs. In this case the cut cost is measured by the total weight of the respective edge set of the solution. We show that when near-balance is allowed, trees exhibit a substantially better behavior compared to general graphs. We present an algorithm for edge-weighted trees that computes a near-balanced partition for any constant $\varepsilon > 0$ in polynomial time. It achieves a cut cost no larger than the optimal for a perfectly balanced partition, i.e. $\alpha = 1$. In this sense the presented algorithm is a PTAS w.r.t. the balance of the computed solution. In addition, such a PTAS can be shown to yield an optimal *perfectly balanced* solution for trees if $k \in \Theta(n)$. On general graphs the problem is NP-hard for these values of k [79].

In the last section of our chapter we capitalise on the PTAS we presented for trees to tackle the k−`BALANCED PARTITIONING` problem on general edge-weighted graphs. By decomposing a graph into a collection of trees with a cut distortion of $\mathcal{O}(\log n)$, we can use our PTAS for trees to get a solution for graphs. Since the PTAS has approximation factor $\alpha = 1$, the total approximation factor paid for the general graphs is due only to the distortion of the decomposition, that is $\alpha \in \mathcal{O}(\log n)$. Note that since the graph is decomposed into trees as a preliminary step, the decomposition is oblivious of the balance constraints related to solving k−`BALANCED PARTITIONING` on the individual trees. Hence the distortion does not depend on the balance factor ε. This is sufficient to simultaneously improve on the previous best result known [8] of $\alpha \in \mathcal{O}(\log^{1.5}(n)/\varepsilon^2)$, and answer an open question posed in the same paper whether an algorithm with no dependence on ε in the ratio α exists. Figure 2.4 summarizes the related work and our contribution.

2.1.2 Related Work

This chapter extends the results in [8], where it is shown that approximating the cut cost of the k-BALANCED PARTITIONING problem on general graphs is NP-hard for any finite factor α, if perfectly balanced partitions are needed. In [8] the authors also give a bicriteria approximation algorithm with $\alpha \in \mathcal{O}(\log^{1.5}(n)/\varepsilon^2)$ when solutions are allowed to be near-balanced. When more unbalance is allowed, for $\varepsilon = 1$ Even *et al.* [43] present an algorithm with $\alpha \in \mathcal{O}(\log n)$ that uses spreading metrics techniques. For the same value of ε, Simon and Teng [115] gave a method that can achieve a factor of $\alpha \in \mathcal{O}(\log k\sqrt{\log n})$ by recursively applying edge separators. Later Krauthgamer *et al.* [83] improved these results to $\alpha \in \mathcal{O}(\sqrt{\log n \log k})$ using a semidefinite relaxation which combines l_2^2 metrics with spreading metrics. For graphs with excluded minors[3] (such as planar graphs) solutions with $\alpha \in \mathcal{O}(1)$ and $\varepsilon = 1$ can be computed in $\widetilde{\mathcal{O}}(n^3)$ deterministic, or $\widetilde{\mathcal{O}}(n^2)$ expected, time by applying a spreading metrics relaxation and the results in [80]. A faster algorithm can be obtained [45] for the special case of grid graphs without holes. This is achieved by combining some structural insights [46] on the cut shapes with the above mentioned techniques of Simon and Teng [115]. This yields an algorithm running in $\widetilde{\mathcal{O}}(n^{1.5})$ time. While the algorithm also computes partitions with $\varepsilon = 1$, it trades the faster runtime with a worse ratio of $\alpha \in \mathcal{O}(\log k)$. For the problem under consideration, to the best of our knowledge these are the only results

[3]We thank an anonymous reviewer for pointing out this folklore result.

for restricted graph classes. However for a related problem on scheduling of pipelined operator trees, Bodlaender *et al.* [16] gave a PTAS. In this problem the objective is also to find a partition of the vertices into k sets. However instead of fixing an upper bound on the set sizes, the optimisation function to be minimised depends on the cut cost and the maximum set size.

The special case when $k = 2$, commonly known as the BISECTION problem, is well studied. The BISECTION problem is NP-hard in the general case [57] but approximation algorithms are known. Räcke [108] gives an algorithm with approximation ratio $\alpha \in \mathcal{O}(\log n)$ for perfectly balanced partitions. For near-balanced partitions, Leighton and Rao [85] show how to compute a solution using min-ratio cuts. In this solution the cut cost is approximated within $\alpha \in \mathcal{O}(\gamma/\varepsilon)$, where γ is the approximation factor of computing a min-ratio cut. In [85] it was shown that $\gamma \in \mathcal{O}(\log n)$, and this result was improved [10] to $\gamma \in \mathcal{O}(\sqrt{\log n})$. For (unweighted) planar graphs it is possible to compute the optimum min ratio cut in polynomial time [105]. If a perfectly balanced solution to BISECTION is required for planar graphs, Dìaz *et al.* [38] show how to obtain a PTAS. Even though it is known that the BISECTION problem is weakly NP-hard on planar graphs with vertex weights [105], whether it is NP-hard on these graphs in the unweighted case is unknown. For other special graph classes the problem can be solved optimally in polynomial time [37]. For instance an $\mathcal{O}(n^4)$ time algorithm for grid graphs without holes has been found [49], while for trees an $\mathcal{O}(n^2)$ time algorithm [90] exists.

In addition to the case $k = 2$, some results are known for other extreme values of k. For trees the algorithm from [90] is easily generalized to solve the $k-$BALANCED PARTITIONING problem for any constant k in polynomial time. At the other end of the spectrum, i.e. when $k \in \Theta(n)$, it is known that the problem is NP-hard [79] for any $k \leq n/3$ on general graphs. Feo and Khellaf [51] give an $\alpha = n/k$ approximation algorithm for the cut cost which was improved to $\alpha = 2$ [50] in case k equals $n/3$ or $n/4$.

We complete the review of related work by discussing the literature on hierarchical graph decompositions, which we leverage to extend our PTAS for trees to general graphs. Informally a hierarchical decomposition of a graph G is a set of trees for which the leaves correspond to the vertices of G, and for which the structure of their cuts approximate the cuts in G. Graph decompositions have been studied in the context of oblivious routing schemes (see [91] for a survey). Räcke [108] introduced an optimal decomposition with factor $\mathcal{O}(\log n)$, which we employ in the present work. In a recent work, Madry [91] shows that it is possible to generalize Räcke's insights so that any *cut based* problem (see [91] for more details) is solvable on graphs by computing solutions on trees resulting from a decomposition of the input graph. This result directly translates to our scenario and hence we use his notation in the present work.

2.2 Applications

The k-BALANCED PARTITIONING problem has numerous applications that can be found in VLSI circuit design [15], image processing [113, 123], computer vision [84], route planning [32], and divide-and-conquer algorithms [88, 114].

However, the most prominent application can be found in the field of parallel computing [9] where calculation can be naturally described by graphs in which vertices represent workload and edges encode data dependencies relationships. The goal is then to partition the n vertices of a graph among k parallel processors while minimizing the inter-processor communication. A few models have been proposed in order to capture the metric that better represents the communication sought to be minimized.

The most popular model found in the literature is the *standard graph partitioning model*. In this model, the vertices of the graph are to be partitioned into equally weighted sets so that the weight of the edges crossing the sets is minimized. This model is equivalent to the cut metric considered in this paper. Many heuristics to minimize such metric have been developed and implemented in software packages such as Chaco [66] and METIS [72].

More recently, it has been pointed out [65] that the edge cut metric minimized by the graph partitioning model is not a good proxy for the communication actually taking place during a parallel computation. Most prominently, edge cuts are not proportional to the total communication volume. Consider a simple example of a graph on three nodes u, v_1, v_2 where the only edges are $u - v_1$ and $u - v_2$ both with weight 1. If v_1 and v_2 are assigned to processor P_1 and u is assigned to processor P_2 then the total weight of cut edges is 2. However the data from node u on processor P_1 need only be communicated *once* to processor P_2. Thus, the actual communication volume is only 1.

Despite the limitations of the edge cut measure, the standard partitioning approach has proved successful for a restricted class of problems, including the parallel solution of differential equations and grid-based problems. However, the benefits on a partitioning strategy solely based on the edge cut metric fails to provide significant benefits in general problems with more complex data dependencies for the reason described above and more powerful models have been introduced. We now take into consideration an elegant emerging model that directly capture the volume of communication metric, the *hypergraph model* [35, 65].

A Hypergraph Partition Model For a graph G of n tasks to be partitioned,, a hypergraph H is created in the following way. H has a vertex for each vertex of G. Moreover, for each vertex v_i of G, H has a hyperedge e_i connecting v_i with $N(v_i)$, the set of neighbors of v_i in G.

The (unweighted) hypergraph partitioning problem is defined as follows. Given a hypergraph $H = (V, E)$ and an integer k, partition the vertex set V into k disjoint subsets $V_j, j = 0, ..., k-1$ of approximately equal sizes such that a cut metric is minimized. We refer to $P = \{V_0, ..., V_{k-1}\}$ as a partitioning and the subsets as partitions. A hyperedge is cut if it contains at least two vertices belonging to different partitions. We seek to minimize the cut metric:

$$cuts(H, P) = \sum_{i=0}^{|E|-1} (\lambda_i(H, P) - 1) \tag{2.1}$$

where $\lambda_i(H, P) \leq k$ is the number of partitions spanned by hyperedge i in the partitioning P. This metric is known as the $(k - 1)$-cut; it is important because it accurately reflects communication cost in parallel computing and, in particular, sparse matrix-vector multiplication. When $k = 2$, $cuts(H, P)$ is simply the number of hyperedges cut.

In [36] algorithms that compute a balanced partition that minimizes the cut metric (2.1) are presented. Such algorithms are mainly heuristic in nature and no algorithms with proven approximation guarantees are known for the partition metric (2.1).

Although not directly applicable to the hypergraph model, we argue that the algorithmic results presented in this chapter could provide insights for development of new heuristics attempting at minimizing cut metrics of more practical importance.

2.3 The Hardness of Computing Perfectly Balanced Partitions in Trees

We now consider the problem of finding a perfectly balanced partition with minimum cut cost for an unweighted tree. We prove hardness results in the case where either the diameter or the degree is restricted to be constant. All reductions are from the `3-PARTITION` problem, defined as follows.

Definition 1 (`3-PARTITION`)**.** Given $3k$ integers $a_1, \ldots, a_{3k}$ and a threshold s, such that $s/4 < a_i < s/2$ and $\sum_{i=1}^{3k} a_i = ks$, find a partition of the integers into k triples such that each triple sums up to exactly s.

The `3-PARTITION` problem is strongly NP-hard [57] which means that it remains so even if all integers are polynomially bounded in the size of the input. We call an instance to the `3-PARTITION` problem a ***YES instance*** if it is solvable and a ***NO instance*** otherwise.

We begin by showing that an approximation algorithm with factors $\alpha = n^c$ and $\varepsilon = 0$, for any constant $c < 1$, for k`-BALANCED PARTITIONING` on trees could be used to decide an instance of `3-PARTITION`. The idea for the reduction is similar to the one used by Andreev and Räcke [8] for general graphs. The result remains valid even if the diameter of the tree is bounded by a constant.

Theorem 2. *Unless P=NP, the k`-BALANCED PARTITIONING` problem on unweighted trees has no polynomial time approximation algorithm with ratios $\varepsilon = 0$ and $\alpha = n^c$, for any constant $c < 1$, even if the diameter is at most 4.*

Proof: The construction used in the proof is shown in Figure 2.5. Let $m = 3kn^c$ for some constant $c < 1$. For each a_i of a given instance I of the `3-PARTITION` problem,

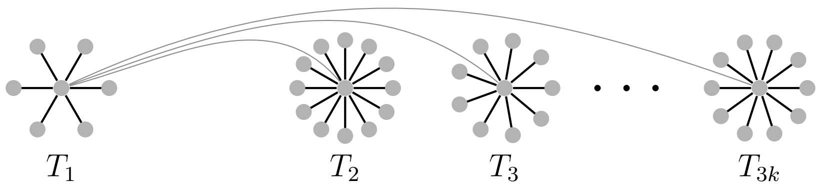


Figure 2.5: Construction for the reduction of Theorem 2. Thin grey edges link star centers, darkened edges connect centers to leaves.

define a corresponding gadget T_i as a star on $a_i m$ vertices. Construct a tree T where the centers of all T_i for $i \geq 2$ are connected to the center of T_1, as shown in Figure 2.5. The number of vertices of the resulting tree is $n = \sum_{i=1}^{3k} a_i m = 3k^2 s n^c$. Solving this equation for n gives $n = (3k^2 s)^{\frac{1}{1-c}}$. Since c is constant and s can be assumed to be polynomially bounded in k, the tree T can be constructed in polynomial time. We now argue that an optimal perfectly balanced partition has cut cost at most $3k - 1$ if and only if I is a YES instance. Otherwise it requires at least $m = 3kn^c$ edges. This would allow an approximation algorithm with approximation factor at most n^c to decide between a YES and a NO instance of the `3-PARTITION` problem. Since the latter is NP-hard this proves the theorem.

It is easy to see that if I is a YES instance then cutting at most the $3k - 1$ edges that connect the T_is suffices. Suppose now that I is a NO instance and that the cut set C^* of minimum size that partitions T into $\{V_1, \ldots, V_k\}$, where $|V_i| = ms$, has size strictly less than m. The set C^* can be expressed as $A \cup B$, where A contains only edges that link T_i centers, and B contains only edges separating a T_i's center from one of its leaves.

By the assumption on the cut cost it follows that $|A| < m$ and $|B| < m$. Also $|B| > 0$ as cutting only edges from A would separate complete T_is, thus implying a solution to I, and contradicting the fact that I is a NO instance.

Let V_l be a set of the partition induced by C^* that contains at least one isolated leaf v, which always exists since $|B| > 0$. Assume that V_l contains an incomplete T_i, that is, V_l contains the center of such a T_i but not one of its leaves w. Then w must be contained in some other V_j, where $j \neq l$. Thus swapping v and w allows putting both w and the incomplete T_i in V_l, rendering the cut that separate them superfluous. This contradicts the assumption that C^* is a cut set of minimum size. Hence V_l can contain only a (possibly empty) set of complete T_is in addition to one or more isolated leaves.

The proof is completed by noticing that the number of vertices of a set of complete T_is is a multiple of m. Therefore at least m additional isolated leaves are required for $|V_l|$ (recall $|V_l| = ms$) to be a multiple of m, contradicting the assumption that $|B| < m$ and establishing the result. $\square$

The reduction in the proof of the above theorem relies on the fact that the degree of the tree is unbounded. Therefore the natural next step is to investigate the complexity of bounded degree trees. As we will show next, the problem remains surprisingly hard for constant degree trees. We are able to prove two hardness results for this case. First we show that the problem of finding a perfectly balanced partition of a tree is APX-hard even if the maximum degree of the tree is at most 7. We prove this result by a reduction from the GAP-3-PARTITION problem. This is the 3-PARTITION problem in which, for a given ρ, either all or at most a ρ fraction of the a_is can be partitioned into the desired triples. A formal definition follows.

Definition 3 (GAP-3-PARTITION)**.** Let $3k$ integers a_1 to a_{3k}, a threshold s, and $\rho > 1$ be given, such that $s/4 < a_i < s/2$, $\sum_{i=1}^{3k} a_i = ks$, and the integers can either all be partitioned into k triples that sum up to exactly s or at most k/ρ of them can. Decide whether k or at most k/ρ such triples can be found.

There is a constant $\rho > 1$ for which the GAP-3-PARTITION problem is NP-hard. This is true even if all integers are polynomially bounded in k. These results follow from the original NP-hardness proof of 3-PARTITION by Garey and Johnson [57] and the results of Petrank [106].[4] The latter result introduces a constant sized gap for the 3D-MATCHING problem. By considering the reductions given by Garey and Johnson from 3D-MATCHING to 3-PARTITION it can readily be seen that they are gap-preserving (cf. [12]). These reductions also establish the strong NP-hardness of 3-PARTITION.

The technique we use to prove the hardness for constant degree trees is substantially different, and more involved, than the method used to prove Theorem 2. Rather than relying on the fact that many edges have to be cut in a gadget consisting of a high

[4]We thank Nikhil Bansal for pointing out the connection between the reductions in [57] and the results by Petrank [106].

degree star, we need gadgets with structural properties that guarantee a number of cut edges proportional to the number of integers that cannot be packed into triples in a GAP-3-PARTITION instance.

Theorem 4. *Unless P=NP, there exists a constant $\rho > 1$ such that the k-BALANCED PARTITIONING problem on unweighted trees with maximum degree at most 7 has no polynomial time approximation algorithm with ratios $\varepsilon = 0$ and $\alpha = 1 + (1 - \rho^{-1})/24$.*

Proof: Consider an instance I of GAP-3-PARTITION with polynomially bounded integers that are divisible by 12. Obviously all hardness properties are preserved by this restriction since GAP-3-PARTITION is strongly NP-hard and we may multiply each integer and the threshold parameter of an arbitrary instance by 12. As a consequence, all integers are divisible by 4 and $s > 20$, which will become important later in the proof. For each a_i in I, construct a gadget T_i composed by a path on a_i vertices (called an a_i-path) connected to the root of a tree on s vertices (referred to as an s-tree). The root of the s-tree branches into four paths, three of them with $s/4$ vertices each, and one with $s/4 - 1$ vertices. Additionally the roots of the s-trees are connected in a path, as shown in Figure 2.6. We define B to be the set of edges connecting different T_is and A the set of edges connecting an a_i-path with the root of the corresponding s-tree in each T_i.

At a high level, we set out to prove that if all k integers in I can be partitioned into triples that sum up to exactly s, then the constructed tree T can be split into $4k$ parts of a perfectly balanced partition with cut cost $6k - 1$. If however at most k/ρ such triples

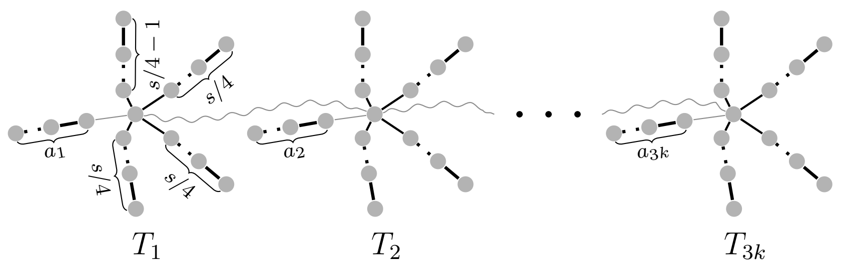


Figure 2.6: Construction for Theorem 4. Each gadget T_i is composed by an a_i-path connected to the root of an s-tree through an edge from A (straight grey). Each s-tree branches into four paths of (almost) the same length. Two adjacent gadgets in a path are connected through the roots of their s-trees with an edge from B (wiggled grey).

can be found, T requires at least $(1-\rho^{-1})k/4$ additional cut edges. This means that an algorithm computing an approximation within a factor smaller than $\frac{6k-1+(1-\rho^{-1})k/4}{6k-1}$ of the optimum cut cost, can decide the `GAP-3-PARTITION` problem, which proves the theorem.

It is easy to see that if all k integers of I can be partitioned into triples of size exactly s, cutting exactly the $6k-1$ edges in A and B suffices to create a valid perfectly balanced partition into $4k$ parts.

It remains to be shown that $(1-\rho^{-1})k/4$ additional edges are required when the integers in I can be partitioned into at most k/ρ triples of size exactly s. Let in this case C^* be an optimal set of edges that cuts T into $4k$ parts of a perfectly balanced partition. We argue that by incrementally repositioning cut edges from the set $C := C^* \setminus (A \cup B)$

to edges in $(A \cup B) \setminus C^*$, eventually all the edges in $A \cup B$ will be cut. However, the following lemma implies that a constant fraction of the edges initially in C will not be moved. We will then argue that the more triples of I cannot be packed into triples of size s, the more edges are left in C. Thus the more edges must additionally have been in C compared to those in $A \cup B$. We rely on the following technical lemma which we will prove later.

Lemma 5. *If* $s > 20$ *then* $|C| \geq 2|(A \cup B) \setminus C^*|$.

Consider the following algorithm $\mathcal{A}$ which repositions cut edges from a perfectly balanced partition into $4k$ parts. As long as there is an uncut edge $e \in A \cup B$, $\mathcal{A}$ removes a cut edge left in C and cuts e instead. At the end of the process, when all edges in $A \cup B$ are cut, $\mathcal{A}$ removes the set of cut edges still left in C denoted by C'. Then $|C'|$ is the minimum number of additional edges cut in the case at most k/ρ triples that sum up to exactly s can be formed from the integers. When repositioning a cut edge from C to $A \cup B$, or when removing the edges in C', $\mathcal{A}$ modifies the sizes of the sets in the partition induced by the cut set, and the balance might be lost. In particular, when a cut edge $e \in C$ is removed, the algorithm will join the two connected components induced by the cut set and incident to e to form a single component. The algorithm will then include the component in an arbitrary one of the sets that contained the two components. This changes the sizes of at most two sets in the partition. When a new cut is introduced by $\mathcal{A}$, a component is split into two and the two newly created components are retained in the same set, thus no set size is changed.

By Lemma 5 there are at least as many edges in C', as there are edges that are repositioned from C to $A \cup B$. Since each edge from C repositioned by $\mathcal{A}$ causes at most two changes in set sizes, the total number of set size changes performed by $\mathcal{A}$ is at most $4|C'|$.

When $\mathcal{A}$ terminates only edges from $A \cup B$ are cut. Therefore the remaining connected components correspond to the $3k$ integers a_i of I and $3k$ integers of size s. The integers in I can be partitioned into at most k/ρ triples of size exactly s. Hence at least $(1-\rho^{-1})k$ of the sets of the resulting partition do not have size exactly s. This means that $\mathcal{A}$ must have changed the size of at least $(1-\rho^{-1})k$ sets, since it converted a perfectly balanced partition of T into a solution to `GAP-3-PARTITION` with at least $(1-\rho^{-1})k$ unbalanced sets. This finally implies that $4|C'| \geq (1-\rho^{-1})k$, which concludes the proof given that $|C'|$ is the number of additional cuts required. $\square$

Proof:[Proof of Lemma 5]

It is sufficient to prove that the lemma holds for all disjoint subtrees T' of T that contain at least one edge from $(A \cup B) \setminus C^*$. That is, if E' is the edge set of T' we set out to prove $|C \cap E'| \geq 2|((A \cup B) \setminus C^*) \cap E'|$, for all pairwise disjoint trees T' such that their union contains $(A \cup B) \setminus C^*$. Consider all connected components cut out by a balanced partition of T that contain at least a root of an s-tree. Create a subtree T' from each such component W by re-attaching all the cut s-tree branches incident to it and all a_i-paths incident to the edges in $A \setminus C^*$ contained in W (see Figure 2.7). Note that T' contains the same edges from A and B as W. In particular no edges from $(A \cup B) \cap C^*$ are in T'. We distinguish two cases.

First, consider a T' that contains no edges from B. If T' also does not contain an edge from A then the lemma is trivially true. Otherwise, the tree T' is simply a gadget T_i and has a total size of $s + a_i$. Since $a_i > s/4$ and each branch of the s-tree in T' has at most $s/4$ vertices, there must be at least two edges from C in T' in order to cut away a_i vertices. This proves the lemma in the case T' contains no edges from B.

Consider now a subtree T' that contains at least one edge from B, and the connected component W from which T' originated. Let A' and B' denote the edges in W from A and B, respectively. Each branch of an s-tree and each a_i-path in T' has at least $s/4 - 1$ vertices. As $s > 20$, if 5 or more branches of s-trees or a_i-paths were fully included in W (before the extension to T') this connected component would contain more than s vertices. This is a contradiction since every connected component has size at most s. Therefore there are at most 4 such included branches. The branches that are not fully included in W but are in T', each contains an edge from C. Since T' contains at least $4(|B'| + 1)$ s-tree branches and $|A'|$ a_i-paths, we can conclude that the number of edges from C in T' is at least $|A'| + 4|B'|$. Notice that $|B'| \geq |A'| - 1$ since otherwise W

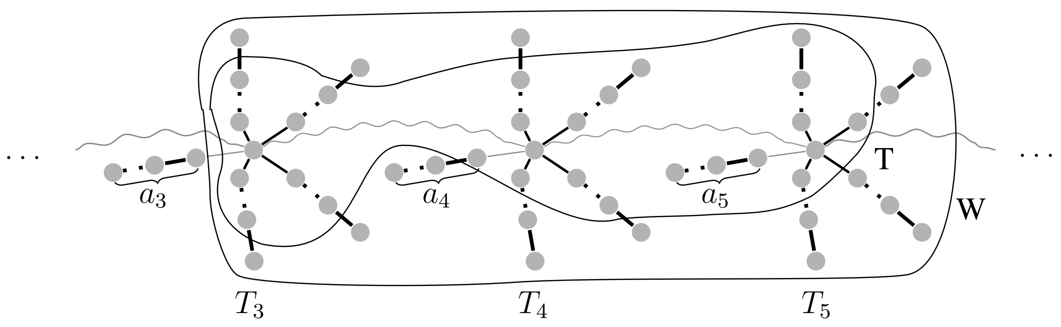


Figure 2.7: Part of the construction of Lemma 5 with components W and T' highlighted.

would be disconnected. Using the fact that $|B'| \geq 1$ we obtain

$$|A'| + 4|B'| \geq 2|A'| - 1 + 3|B'| \geq 2|A' \cup B'|.$$

This proves our claim in the case T' contains at least one edge from B, and concludes the proof of the lemma. □

Using similar ideas as in the proof of Theorem 4, if we restrict the degree to be at most 5 we can still show that the problem remains NP-hard. For this we again reduce from `3-PARTITION` and use a slightly different construction than the one shown in Figure 2.6. Instead of connecting the s-trees through their roots, the B edges connect the leaves of the shortest branches of the s-trees (Figure 2.8). It is then possible to show that exactly the $6k - 1$ edges in A and B are cut if all k integers in the instance I can be partitioned into triples of size exactly s, while otherwise at least $6k$ edges are cut. Since the `3-PARTITION` problem is NP-hard [57] this suffices to establish the following result.

Theorem 6. *Unless P=NP, the* k-BALANCED PARTITIONING *problem on unweighted trees with maximum degree at most* 5 *has no polynomial time algorithm.*

Proof: Consider an instance I of 3-PARTITION where s is a multiple of 4. Clearly the 3-PARTITION problem remains strongly NP-hard even if restricted to such instances since we may multiply all integers by 4. For each a_i in I, construct a gadget T_i composed by a path on a_i vertices (hereinafter, a_i-path) connected to the root of a tree on s vertices (hereinafter, s-tree). The root of the s-tree branches into four paths, three of them with $s/4$ vertices each and one with $s/4 - 1$ vertices. The construction is completed by connecting the leaves of the shortest branch of each s-tree in a path, as shown in Figure 2.8. We call A the set of edges connecting different T_is and B the set of edges connecting an a_i-path with the corresponding s-tree in each T_i.

At a high level, we want to argue that an algorithm that can find a perfectly balanced partition of T into $4k$ sets could be used to solve the 3-PARTITION problem on I. More precisely, we set out to prove that T can be split into $4k$ sets using exactly $6k - 1$ cut edges if and only if I is a YES instance. To this end, we call a partition *light* if it splits T into connected components each of size at most s, and leverage the following lemma.

Lemma 7. *The only cut set of minimum size that breaks* T *into a light partition contains all and only the* $6k - 1$ *edges in* $A \cup B$.

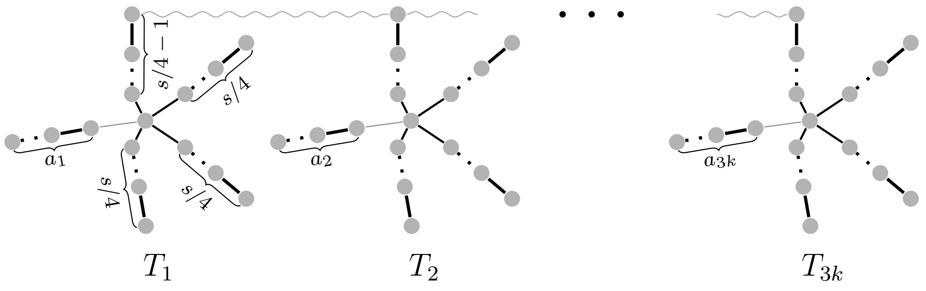


Figure 2.8: Construction for Theorem 6. Each gadget T_i is composed by a path on a_i nodes (a_i-path) connected to the root of a tree on s vertices (s-tree). Each s-tree branches from the root in three paths of length $s/4$ each and a shorter one of length $s/4-1$. The leaves of the shorter branch of each s-tree are connected by edges of the set A (wiggled thin grey) in a path. In each T_i, the a_i-path and the s-tree are connected by an edge belonging to the set B (straight thin grey).

Before proving Lemma 7 we see how Theorem 6 follows from it. Clearly, breaking T into connected components of size less than or equal to s is a necessary condition to obtain a perfectly balanced partition into $4k$ sets. It follows that no such partition can cut less than $6k-1$ edges. If I is a YES instance then cutting all the $6k-1$ edges in $A \cup B$ suffices to find a perfectly balanced partition of T into $4k$ sets. This is because the vertices of the a_i-paths can then be partitioned into sets of size s with no additional cuts. By Lemma 7 it also follows that if cutting $6k-1$ edges suffices to find a perfectly balanced partition of T into $4k$ sets, the cut set must coincide with $A \cup B$. Therefore no additional cut edges are required only if the integers corresponding to the a_i-paths can be packed into triples of size s each. This yields a solution to the `3-PARTITION` instance I, given that each s-tree separated by the $A \cup B$ cut set completely fills a set of size s. Hence if I is a NO instance then the cut cost is at least $6k$, which completes the proof. □

Proof:[Proof of Lemma 7.] First observe that cutting $A \cup B$ suffices to create a light partition. We show that *any* light partition that does not use some edge in $A \cup B$ can be transformed into a light partition that has the same number of cut edges and cuts all edges in $A \cup B$ plus at least one more. Therefore the only light partition of minimum cut cost is the one where all edges in $A \cup B$ are cut.

We begin by showing how to convert a light partition that does not use an edge $b \in B$, connecting an a_i-path to an s-tree, into one that uses all edges in B without increasing the cut cost. Suppose T_l is a gadget where an edge $b \in B$ is not cut. Since the size of T_l is $s + a_l$ at least a_l vertices have to be separated from T_l to form a light partition. Each branch of the s-tree is strictly smaller than a_l and hence, without using b, there must be at least two edges cut in two different branches of T_l. At least one of these edges c cannot be on the shortest branch of the s-tree, which is connected to a neighboring T_i. There are two cases. If c cuts some edge of the a_l-path (other than b), then c can be replaced with b without invalidating the light partition property. Else, if no cut edge is on the a_l-path, there is a cut edge c on one of the longest branches of the s-tree of T_l which can be replaced with b. This again results in a light partition as the a_l-path has size strictly larger than any branch of the s-tree of T_l.

Repeatedly applying the transformation described above for each gadget T_i, we can obtain a light partition of T where all edges in B are cut and the total size of the cut set has not increased. Also at least one additional edge in such a gadget T_i is cut. We are then left to prove that if any edge $a \in A$ is not cut then the cut set is strictly suboptimal.

Suppose not all edges from A are cut. Then there exist chains of length $p > 1$ of T_is connected by edges in A that are not cut. By the above transformation, we can assume that all T_is in such a chain have their corresponding a_i-path cut out. Furthermore, each T_i in the chain must have at least one more edge cut. If that was not the case the size of the connected component of the light partition containing a T_i that violates the property would be at least $s+1$. This is because the edge $a \in A$ from T_i to its neighboring gadget in the chain is not cut. Consider T_l, the first T_i of such a chain. The cut edge $c \notin B$ in T_l can be replaced by the edge $a \in A$ to the neighboring gadget without invalidating the light partition property. This follows from the fact that cutting a will completely separate T_l from the chain and the total size of T_l with its a_i-path removed is exactly s. The same process can be repeated for all the $p - 2$ remaining edges from A on the chain of gadgets. This implies that for each chain of T_is all but at least one cut edge can be replaced with edges in A.

Hence after replacing each cut edge not in $A \cup B$ with one from this set we obtain a light partition with the same number of cut edges. However in this light partition at least one additional edge is cut in addition to those in $A \cup B$ and hence the lemma follows. $\square$

2.4 Near-Balanced Partitions for Trees

The previous section shows that approximating the cut cost of k-`BALANCED PARTITIONING` is hard even on unweighted trees, if perfectly balanced partitions are desired. We showed that for the general case when the degree is unbounded there is no hope for a polynomial time algorithm with non-trivial approximation guarantee. Therefore, in this section we study the complexity of the problem when allowing the partitions to deviate from being perfectly balanced. In contrast to the negative results presented so far, we prove the existence of a PTAS for k-`BALANCED PARTITIONING` on trees with edge weights. It computes near-balanced partitions but returns a cut cost no larger than the optimum of a perfectly balanced solution.

Assume we are given an edge-weighted tree $T = (V, E, \omega)$ with weight function $\omega : E \to \mathbb{R}^+$. Conceptually one could find a *perfectly balanced* partition of T with minimum cut cost in two steps. First all the possible ways of cutting T into connected components are grouped into equivalence classes based on the sizes of their components. That is, the sets of connected components $\mathcal{S}$ and $\mathcal{S}'$ belong to the same equivalence class if they contain the same number of components of size x for all $x \in \{1, \ldots, \lceil n/k \rceil\}$. In a first step the set of connected components that achieves the cut of minimum cost for each class is computed and set to be the representative of the class. In a second stage only the equivalence classes whose elements can be packed into k sets of size at most $\lceil n/k \rceil$ are considered, and among those the representative of the class with minimum overall cut cost is returned. Clearly such an algorithm finds the optimal solution to the k-BALANCED PARTITIONING problem, but the runtime is exponential in n as, in particular, the total number of equivalence classes is exponential. To get around this problem we instead group sets of connected components into coarser equivalence classes. These are determined by subdividing the possible component sizes into intervals. A coarse class then consists of cuts for which each interval in total contains the same number of component sizes. By making the lengths of the intervals appropriately depend on ε, this reduces the equivalence classes to a polynomial number if ε is constant. However this also introduces an approximation error in the balance of the solution.

Definition 8. Let $\mathcal{S}$ be a set of disjoint connected components of the vertices of T, and $\varepsilon > 0$. A vector $\vec{g} = (g_0, \ldots, g_t)$, where $t = \lceil \log_{1+\varepsilon}(1/\varepsilon) \rceil + 1$, is called the *signature* of $\mathcal{S}$ if in $\mathcal{S}$ there are g_0 components of size in $[1, \varepsilon\lceil n/k \rceil)$ and g_i components of size in $[(1+\varepsilon)^{i-1} \cdot \varepsilon\lceil n/k \rceil, (1+\varepsilon)^i \cdot \varepsilon\lceil n/k \rceil)$, for each $i \in \{1, \ldots, t\}$.

The first stage of our algorithm uses a dynamic programming scheme to find a set of connected components of minimum cut cost among those with signature $\vec{g}$, for any possible $\vec{g}$. Let $\mathbb{S}$ denote the set containing each of these optimal sets that cover all vertices of the tree, as computed by the first stage. In the second stage the algorithm attempts to distribute the connected components in each set $\mathcal{S} \in \mathbb{S}$ into k bins, where each bin has a capacity of $(1+\varepsilon)\lceil n/k \rceil$ vertices. This is done using a scheme originally proposed by Hochbaum and Shmoys [68, 121] for the `BIN PACKING` problem. The final output of our algorithm is the partition of the vertices of the given tree that corresponds to a packing of a set $\widetilde{\mathcal{S}} \in \mathbb{S}$ that uses at most k bins and has minimum cut cost. Both stages of the algorithm have a runtime exponential in t. Hence the runtime is polynomial if ε is a constant.

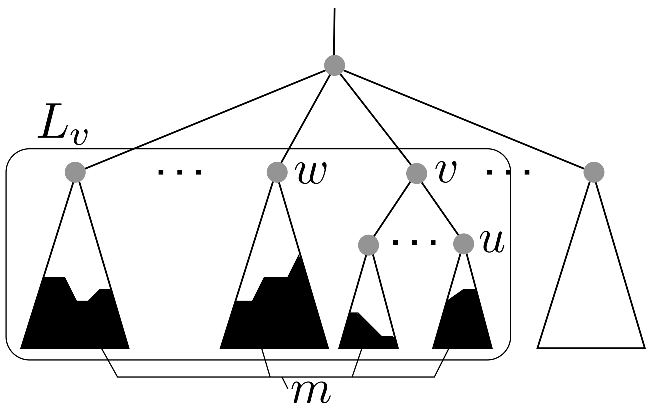


Figure 2.9: A part of a tree in which a vertex v, its rightmost child u, its predecessor w, the set of vertices L_v, and the m covered vertices by some lower frontier with signature $\vec{g}$ are indicated.

2.4.1 The Cutting Phase

We now describe the dynamic programming scheme to compute the set of connected components of minimum cut cost among those whose signature is $\vec{g}$, for every possible $\vec{g}$. We fix a root $r \in V$ among the vertices of T, and an ordering of the children of every vertex in V. We define the *leftmost* and the *rightmost* among the children of a vertex, the siblings *left of* a vertex, and the *predecessor* of a vertex among its siblings according to this order in the natural way. The idea is to recursively construct a set of disjoint connected components for every vertex $v \neq r$ by using the optimal solutions of the subtrees rooted at the children of v and the subtrees rooted at the siblings left of v. Let for a vertex $v \neq r$ the set $L_v \subset V$ contain all the vertices of the subtrees rooted at those siblings of v that are left of v and at v itself (Figure 2.9). We refer to a set $\mathcal{F}$

of disjoint connected components as a *lower frontier of* L_v if all components in $\mathcal{F}$ are contained in L_v and the vertices in V not covered by $\mathcal{F}$ form a connected component containing the root r. For every vertex v and every signature $\vec{g}$, the algorithm recursively finds a lower frontier $\mathcal{F}$ of L_v with signature $\vec{g}$. Finally, a set of connected components with signature $\vec{g}$ covering all vertices of the tree can be computed using the solutions of the rightmost child of the root. The algorithm selects a set having minimum cut cost in each recursion step. Let $C_v(\vec{g}, m)$, for any vertex $v \neq r$ and any integer m, denote the optimal cut cost over those lower frontiers of L_v with signature $\vec{g}$ that cover a total of m vertices with their connected components. If no such set exists let $C_v(\vec{g}, m) = \infty$. Additionally, we define $\mu := (1 + \varepsilon)\lceil n/k \rceil$, and $\vec{e}(x)$ for any integer $x < \mu$ to be the signature of a set containing only one connected component of size x. We now show that the function $C_v(\vec{g}, m)$ can be computed using a dynamic program.

Let $\mathcal{F}^*$ denote an optimal lower frontier associated with $C_v(\vec{g}, m)$. We will consider the four cases resulting from whether or not the vertex v is a leaf, and whether or not it is the leftmost among its siblings. First consider the case when both properties are met. That is, v is a leaf and the leftmost among its siblings. Then $L_v = \{v\}$ and hence the set $\mathcal{F}^*$ either contains $\{v\}$ as a component or is empty. In the latter case the cut cost is 0. In the former it is the weight $\omega(e)$ of the edge e incident to the leaf that is cut from the tree. Thus $C_v((0, \dots, 0), 0) = 0$ and $C_v(\vec{e}(1), 1) = \omega(e)$ while all other function values equal infinity. Now consider the case when v is neither a leaf nor the leftmost among its siblings. Let w be the predecessor among v's siblings and u the rightmost child of v. The set L_v contains the vertices of the subtrees rooted at v's siblings that are left of v and at v itself. The lower frontier $\mathcal{F}^*$ can either be one in which the edge from v to its parent is cut or not. In the latter case the m vertices that are covered by $\mathcal{F}^*$ do not contain v and hence are distributed among those in L_w and L_u since $L_v = L_w \cup L_u \cup \{v\}$. If x of the vertices in L_u are covered by $\mathcal{F}^*$ then $m - x$ must be covered by $\mathcal{F}^*$ in L_w. The vector $\vec{g}$ must be the sum of two signatures $\vec{g}_u$ and $\vec{g}_w$ such that the lower frontier of L_u (respectively L_w) has minimum cut cost among those having signature $\vec{g}_u$ (respectively $\vec{g}_w$) and covering x (respectively $m - x$) vertices. If this were not the case the lower frontier in L_u (respectively L_w) could be exchanged with one having a smaller cut cost—a contradiction to the optimality of $\mathcal{F}^*$. Hence in case v is a non-leftmost internal vertex and the edge to its parent is not cut,

$$C_v(\vec{g}, m) = \min \big\{ C_w(\vec{g}_w, m - x) + C_u(\vec{g}_u, x) \mid 0 \le x \le m \wedge \vec{g}_w + \vec{g}_u = \vec{g} \big\}. \quad (2.2)$$

If the edge connecting v to its parent is cut in $\mathcal{F}^*$, then all n_v vertices in the subtree rooted at v are covered by $\mathcal{F}^*$. Hence the other $m - n_v$ vertices covered by $\mathcal{F}^*$ must be included in L_w. Let x be the size of the component $S \in \mathcal{F}^*$ that includes v. Analogous to the case before, the lower frontiers L_u and L_w with signatures $\vec{g}_u$ and $\vec{g}_w$ in $\mathcal{F}^* \setminus \{S\}$ must have minimum cut costs. Hence the vector $\vec{g}$ must be the sum of $\vec{g}_u$, $\vec{g}_w$, and $\vec{e}(x)$.

Therefore in case the edge e to v's parent is cut,

$$C_v(\vec{g}, m) = \omega(e) + \min\big\{C_w(\vec{g}_w, m - n_v) + C_u(\vec{g}_u, n_v - x) \mid 1 \leq x < \mu \wedge \vec{g}_w + \vec{g}_u + \vec{e}(x) = \vec{g}\big\}. \quad (2.3)$$

Taking the minimum value of the formulas given in (2.2) and (2.3) thus correctly computes the value for $C_v(\vec{g}, m)$ for the case in which v is neither a leaf nor the leftmost among its siblings. In the two remaining cases when v is either a leaf or a leftmost sibling, either the vertex u or w does not exist. For these cases the recursive definitions of $C_v(\cdot, \cdot)$ can easily be derived from Equations (2.2) and (2.3) by letting all function values $C_u(\vec{g}, x)$ and $C_w(\vec{g}, x)$ of a non-existent vertex u or w be 0 if $\vec{g} = (0, \ldots, 0)$ and $x = 0$, and ∞ otherwise.

The above recursive definitions for $C_v(\cdot, \cdot)$ give a framework for a dynamic programming scheme that computes the wanted solution set $\mathbb{S}$ in polynomial-time if ε is a constant, as the next theorem shows.

Theorem 9. *For any tree T and any constant $\varepsilon > 0$ there is an algorithm that computes $\mathbb{S}$ in polynomial time.*

Proof: If the tree contains only one vertex the theorem obviously holds. Otherwise the optimum solution from $\mathbb{S}$ that has signature $\vec{g}$ must contain a connected component that includes the root r and has some size x. Clearly x is at least 1 and at most μ. Hence, if u denotes the rightmost child of the root r, the cut cost $C(\vec{g})$ of the optimal solution for

$\vec{g}$ can be computed in linear time using the formula

$$C(\vec{g}) = \min\{C_u(\vec{g} - \vec{e}(x), n - x) \mid 1 \leq x < \mu\}. \tag{2.4}$$

An optimal set of connected components with signature $\vec{g}$ can be computed using the dynamic program given by the above equation together with the recursive definition of C_v by keeping track of the set of connected components used in each recursion step.

To analyse the runtime let us first bound the number of signatures $\vec{g}$ that have to be considered for a vertex v in the dynamic program. Let $N_v = |L_v|$ denote the number of vertices in the subtrees rooted at the siblings left of v and at v itself. There are N_v vertices that can be distributed into connected components of different sizes to form a lower frontier $\mathcal{S}$ of L_v. Each entry g_i of $\vec{g}$ counts components of size at least the lowest value of the i-th interval as specified in Definition 8. Hence each g_i is upper-bounded by $N_v/((1+\varepsilon)^{i-1} \cdot \varepsilon n/k) \leq k/((1+\varepsilon)^{i-1} \cdot \varepsilon)$ if $i \in \{1, \ldots, t\}$, and N_v if $i = 0$. Therefore the total number of signatures $\vec{g}$ considered for a vertex v is upper-bounded by

$$N_v \cdot \prod_{i=1}^{t} \frac{k}{(1+\varepsilon)^{i-1} \cdot \varepsilon} = N_v \left(\frac{k}{\varepsilon}\right)^t \cdot \left(\frac{1}{1+\varepsilon}\right)^{\frac{(t-1)t}{2}} \leq N_v \left(\frac{k}{\sqrt{\varepsilon}}\right)^t,$$

where the inequality holds since $t - 1 = \lceil \log_{1+\varepsilon}(1/\varepsilon) \rceil$. The latter value can be upper-bounded by $\lceil 1/\varepsilon \cdot \log(1/\varepsilon) \rceil$ if $\varepsilon \leq 1$, since then $1 + \varepsilon \geq 2^{\varepsilon}$. Hence we can conclude that the number of signatures is γN_v, where $\gamma \in \mathcal{O}((k/\sqrt{\varepsilon})^{1+\lceil \frac{1}{\varepsilon} \log(\frac{1}{\varepsilon}) \rceil})$.

We bound the runtime as follows. For each vertex v we calculate the number of steps T_v that are needed to compute all entries $C_{v'}(\vec{g}, m)$ for all $v' \in L_v$. We claim that

$T_v \leq \frac{3}{2}\gamma^2 N_v^4$ for any vertex v. According to Formulas (2.2) and (2.3), in addition to the number of steps T_u and T_w to compute the tables for L_u and L_w, for each m and $\vec{g}$ the minimum value over two options is found by going through all possible x, $\vec{g}_u$, and $\vec{g}_w$. For any fixed x there are at most $\gamma N_u \cdot \gamma N_w$ many possibilities to combine vectors $\vec{g}_u$ and $\vec{g}_w$ to form a signature $\vec{g}$. Since m and x are both upper-bounded by N_v and $N_u + N_w \leq N_v$ we get

$$\begin{aligned} T_v &\leq T_u + T_w + 2\gamma^2 N_u N_w N_v^2 \\ &\leq \frac{3}{2}\gamma^2 N_v^2 \left(N_u^2 + N_w^2 + 2N_u N_w\right) \\ &\leq \frac{3}{2}\gamma^2 N_v^4. \end{aligned}$$

Since the time to compute Formula (2.4) for each signature is $\mathcal{O}(\gamma n)$, we conclude that the total runtime is $\mathcal{O}(\gamma^2 n^4) = \mathcal{O}(n^4(k/\sqrt{\varepsilon})^{2+2\lceil \frac{1}{\varepsilon}\log(\frac{1}{\varepsilon})\rceil})$, which is polynomial if ε is constant. □

2.4.2 The Packing Phase

The second stage of the algorithm attempts to pack each set of connected components $\mathcal{S} \in \mathbb{S}$, computed by the first stage, into k bins of capacity $(1+\varepsilon)\lceil n/k\rceil$. This means solving the well known `BIN PACKING` problem, which is NP-hard in general. However we are able to solve it in polynomial time for constant ε using a method developed by Hochbaum and Schmoys [68], which we briefly describe as presented in [121].

Let $\mathcal{S} \in \mathbb{S}$ be a set of connected components with signature $\vec{g} = (g_0, \ldots, g_t)$. First the algorithm constructs an instance I of the `BIN PACKING` problem containing only the components of $\mathcal{S}$ larger than $\varepsilon\lceil n/k \rceil$. In particular, the bin capacity is set to be $\lceil n/k \rceil$ and for every entry $1 \leq i \leq t$ of $\vec{g}$, g_i elements of size $(1+\varepsilon)^{i-1} \cdot \varepsilon\lceil n/k \rceil$ are introduced in I. That is, the size of each component is converted to the lower endpoint of the interval which contains it according to Definition 8. The number of elements in I is $\sum_{i \geq 1} g_i \leq n/(\varepsilon\lceil n/k \rceil) \leq k/\varepsilon$ since there are n vertices in V and the smallest size of an element in I is $\varepsilon\lceil n/k \rceil$. An optimal bin packing for I can be found in $\mathcal{O}((k/\varepsilon)^{2t})$ time, using a dynamic programming scheme (for more details see [68, 121]). That is, the runtime is exponential in the number t of different sizes of the elements. A packing of I into the minimum number of bins of capacity $\lceil n/k \rceil$ translates into a packing of the components of $\mathcal{S}$ larger than $\varepsilon\lceil n/k \rceil$ into bins of capacity $(1+\varepsilon)\lceil n/k \rceil$. This is because each element in I underestimates the size of the component in $\mathcal{S}$ that it represents by a factor of at most $1+\varepsilon$.

To complete the packing of $\mathcal{S}$ the algorithm distributes the remaining components of size less than $\varepsilon\lceil n/k \rceil$ by greedily putting them into bins without exceeding the capacity of $(1+\varepsilon)\lceil n/k \rceil$. A new bin is created if there is no room for a component in any of the open bins. Distributing the remaining components can be performed in $\mathcal{O}(n)$ time. Let $\varphi(\mathcal{S})$ denote the number of bins that this algorithm needs to pack $\mathcal{S}$. Note that for two sets of components having the same signature the components larger than $\varepsilon\lceil n/k \rceil$ will always be distributed in the same way by the algorithm. However the greedy distribution of the remaining small components may create more bins for one of the sets. We show next that if a set of components computed by the first stage has the same signature $\vec{g}^*$ as the set of components induced by an optimal perfectly balanced partition, then the second stage of the algorithm packs it into at most k bins with capacity $(1+\varepsilon)\lceil n/k \rceil$.

Lemma 10. *Let $\mathcal{S}^*$ having signature $\vec{g}^*$ be the set of connected components in an optimal perfectly balanced partition. For the set $\mathcal{S} \in \mathbb{S}$ with signature $\vec{g}^*$ it holds that $\varphi(\mathcal{S}) \leq k$.*

Proof: We distinguish two cases for the greedy distribution of the components of $\mathcal{S}$ that have size less than $\varepsilon\lceil n/k \rceil$ depending on whether or not new bins are created. If no new bins are created then $\varphi(\mathcal{S})$ is solely determined by the output of the bin packing algorithm, run with capacities $\lceil n/k \rceil$ on the instance I. Since $\mathcal{S}^*$ has the same signature $\vec{g}^*$ as $\mathcal{S}$, all elements $e_i \in I$ can be mapped to distinct components $S_i \in \mathcal{S}^*$ such that $e_i \leq |S_i|$. Hence any packing of $\mathcal{S}^*$ into bins of capacity $\lceil n/k \rceil$ requires at least $\varphi(\mathcal{S})$ many bins which is optimal for I. Since $\mathcal{S}^*$ requires at most k optimally packed bins by definition, this proves the claim in the case no new bins are opened.

If new bins are created by the greedy step, then at least the first $\varphi(\mathcal{S}) - 1$ bins of the final solution are filled beyond the extent of $\lceil n/k \rceil$. Otherwise small components of size at most $\varepsilon\lceil n/k \rceil$ could have been fit without requiring the creation of the last bin. Therefore the total number of vertices in $\mathcal{S}$ strictly exceeds $(\varphi(\mathcal{S}) - 1)\lceil n/k \rceil$. Since the total number of vertices contained in $\mathcal{S}^*$ equals that of $\mathcal{S}$, it follows that at least $\varphi(\mathcal{S})$ bins are required to pack $\mathcal{S}^*$ into bins of capacity $\lceil n/k \rceil$. This proves the claim in the case new bins were created by the greedy step. □

The final step of the algorithm is to output the packing of a set $\mathcal{S} \in \mathbb{S}$ of minimum cut cost among those with $\varphi(\mathcal{S}) \leq k$. The next theorem proves correctness and bounds the runtime of the algorithm.

Theorem 11. *For any tree T with positive edge weights, $\varepsilon > 0$, and $k \in \{1, \ldots, n\}$, there is an algorithm that computes a partition of T's vertices into k sets such that each set has size at most $(1+\varepsilon)\lceil n/k \rceil$ and its cut cost is at most that of an optimal perfectly balanced partition of the tree. Furthermore the runtime is polynomial if ε is a constant.*

Proof: Let $\widetilde{\mathcal{S}} \in \mathbb{S}$ be the set of connected components returned by the algorithm, i.e. if $C(\vec{g})$ denotes the cut cost of the set $\mathcal{S} \in \mathbb{S}$ with signature $\vec{g}$, then

$$\widetilde{\mathcal{S}} = \arg_{\mathcal{S} \in \mathbb{S}} \min\{C(\vec{g}) \mid \mathcal{S} \text{ has signature } \vec{g} \wedge \varphi(\mathcal{S}) \leq k\}. \tag{2.5}$$

By Lemma 10 we know that if $\mathcal{S} \in \mathbb{S}$ has signature $\vec{g}^*$ then $\varphi(\mathcal{S}) \leq k$. Thus the minimisation of (2.5) ensures that the cut cost of $\widetilde{\mathcal{S}}$ is at most that of a set of components $\mathcal{S} \in \mathbb{S}$ with signature $\vec{g}^*$. Since $\mathbb{S}$ retains the set of components with minimum cut cost among all those having the same signature, it follows that the cut cost of $\widetilde{\mathcal{S}}$ is at most that of $\mathcal{S}^*$, which concludes the proof of correctness.

To bound the runtime of the algorithm, recall from the proof of Theorem 9 that the total number of considered signatures $\vec{g}$ is γn, where $\gamma \in \mathcal{O}((k/\sqrt{\varepsilon})^{1+\lceil \frac{1}{\varepsilon}\log(\frac{1}{\varepsilon})\rceil})$. By Theorem 9 the set $\mathbb{S}$, whose size is at most γn, can be computed in time $\mathcal{O}(n^4\gamma^2)$. Each of the sets of components of $\mathbb{S}$ requires at most $\mathcal{O}((k/\varepsilon)^{2t} + n)$ time to be packed in the second stage of the algorithm. Hence the second stage can be performed in $\mathcal{O}(\gamma n((k/\varepsilon)^{2t} + n))$ total time. This means that the overall runtime of the algorithm is $\mathcal{O}(n^4(k/\varepsilon)^{1+3\lceil \frac{1}{\varepsilon}\log(\frac{1}{\varepsilon})\rceil})$, which concludes the proof. □

2.5 Extension to General Graphs

In this section we present an algorithm that uses the PTAS given in Section 4.3 to find a near-balanced partition of an edge-weighted graph G. The algorithm presented computes a cut cost of at most $\alpha \in \mathcal{O}(\log n)$ times that of an optimal perfectly balanced partition of G. It relies on using the PTAS of Section 4.3 to compute near-balanced partitions of a set of decomposition trees that well approximate the cuts in G. This decomposition can be found using the results by Räcke [108]. An in-depth discussion of decomposition trees can be found in [91, 108]. Here we only introduce the basic concepts needed for our purposes.

For a graph $G = (V, E, \omega)$ with weight function $\omega : E \to \mathbb{R}^+$, the quality of a *cut* $W \subseteq V$ is measured using its *cut cost* $C(W)$. It is defined to be the sum of the weights of the edges connecting vertices in W and $V \setminus W$. A *decomposition tree* of G is a tree

$T = (V_T, E_T, \omega_T)$, with weight function $\omega_T : E_T \to \mathbb{R}^+$, for which the leaves $L \subset V_T$ of T correspond to the vertices in G. More precisely there is a mapping $m_G : V_T \to V$ of all tree vertices to vertices in G such that m_G induces a bijection between L and V. Let $m_T : V \to L$ denote the inverse function of this bijection. We also define a *leaf cut* $K \subseteq L$ of a tree and its *cut cost* $C(K)$ is the minimum weight of edges in E_T that have to be removed in order to disconnect K from $L \setminus K$. We map cuts W on G to leaf cuts K on its decomposition tree and vice versa using the notation $m_T(W)$ and $m_G(K)$ to denote the image of W and K according to m_T and m_G respectively. We leverage the following properties of decomposition trees which are proved in [91, 108].

Theorem 12. *For any graph $G = (V, E, u)$ with n vertices, a family of decomposition trees $\{T_i\}_i$ of G and positive real numbers $\{\lambda_i\}_i$ with $\sum_i \lambda_i = 1$ can be found in polynomial time, such that for any cut W of G and corresponding leaf cuts $K_i = m_{T_i}(W)$,*

- *$C(K_i) \geq C(W)$ for each i (lower bound), and*
- *$\sum_i \lambda_i C(K_i) \leq \mathcal{O}(\log n) \cdot C(W)$ (upper bound).*

Since $\sum_i \lambda_i = 1$ the above theorem implies that for at least one tree T_i it holds that $C(K_i) \leq \mathcal{O}(\log n) \cdot C(W)$. That is, in at least one tree T_i the cut structure of G is not distorted too much. This allows for a fast approximation of balanced partitions in graphs using a modified version of the PTAS given in Section 4.3. To this end the PTAS must be adapted to compute near-balanced *leaf partitions* of each T_i. That is, the adapted PTAS computes a partition $\mathcal{L} = \{L_1, \dots, L_k\}$ of the l leaves L of a tree T into k sets of size at most $(1+\varepsilon)\lceil l/k \rceil$ each. The *cut cost* $C(\mathcal{L})$ in this case is the minimum capacity of edges that have to be removed in order to disconnect the sets in $\mathcal{L}$ from each other.

The adaptation of the PTAS given in Section 4.3 to compute near-balanced leaf partitions is straightforward. First, signatures need to count leaves instead of vertices in Definition 8. Also, we need to keep track of the number l_v of leaves at a subtree of a vertex v instead of the number n_v of vertices in Equations (2.2) and (2.3). The following result summarises the properties of the adapted PTAS.

Corollary 13. *For any weighted tree T, $\varepsilon > 0$, and $k \in \{1, \dots, l\}$, there is an algorithm that computes a partition of the l leaves of T into k sets such that each set includes at most $(1+\varepsilon)\lceil l/k \rceil$ leaves and its cut cost is at most that of an optimal perfectly balanced leaf partition. The runtime is polynomial in k and the number of vertices of T if ε is constant.*

Next we show our main result. It entails that the adapted PTAS can be used to compute near-balanced partitions for general graphs so that the cut cost deviates by only a logarithmic factor from the optimum.

Theorem 14. *Let $G = (V, E, \omega)$ be a graph with n vertices, $\varepsilon > 0$ be a constant, and $k \in \{0, \dots, n\}$. There is an algorithm that computes a partition of V into k sets such* *that each set has size at most $(1+\varepsilon)\lceil n/k \rceil$ and its cut cost is at most $\mathcal{O}(\log n)$ times that of the optimal perfectly balanced solution. The runtime is polynomial if ε is constant.*

Proof: We use Theorem 12 to compute a family of decomposition trees with the properties listed therein. This family has a polynomial number of trees since the runtime is polynomial. For each such tree we compute a partition of its leaves into k sets of size at most $(1+\varepsilon)\lceil n/k \rceil$ using Corollary 13. We select the computed leaf partition $\mathcal{L}^*$ of the decomposition tree T^* having the smallest cut cost when applied to G. That is, $\mathcal{L}^*$ minimises the quantity $C(m_G(\mathcal{L}))$ among all computed leaf partitions, where for a leaf partition $\mathcal{L} = \{L_1, \dots, L_k\}$ we define $m_G(\mathcal{L}) = \{m_G(L_1), \dots, m_G(L_k)\}$. The output of the algorithm is then the vertex partition $m_G(\mathcal{L}^*)$ of G.

By Theorem 12, for some decomposition tree T' and the corresponding leaf partition $m_{T'}(\mathcal{V}^*) = \{m_{T'}(V_1^*), \dots, m_{T'}(V_k^*)\}$ of the optimal perfectly balanced partition $\mathcal{V}^* = \{V_1^*, \dots, V_k^*\}$, we get the upper bound $C(m_{T'}(\mathcal{V}^*)) \leq \mathcal{O}(\log n) \cdot C(\mathcal{V}^*)$. This is due to the observation that for any (vertex or leaf) partition $\mathcal{X} = \{X_1, \dots, X_k\}$ it holds that $C(\mathcal{X}) = \frac{1}{2}\sum_{j=1}^{k} C(X_j)$. By Corollary 13 we know that the cut cost of the computed leaf partition $\mathcal{L}'$ in T' is at most the cut cost of the optimal perfectly balanced leaf partition in T', hence $C(\mathcal{L}') \leq C(m_{T'}(\mathcal{V}^*))$. From the lower bound in Theorem 12 we can conclude that $C(m_G(\mathcal{L}')) \leq C(\mathcal{L}')$. Finally, since we chose $\mathcal{L}^*$ to minimise the quantity $C(m_G(\mathcal{L}))$ among all computed leaf partitions, we get $C(m_G(\mathcal{L}^*)) \leq C(m_G(\mathcal{L}'))$. This implies $C(m_G(\mathcal{L}^*)) \leq \mathcal{O}(\log n) \cdot C(\mathcal{V}^*)$, and concludes the proof. $\square$

2.6 Conclusions

In this chapter, we studied the k-BALANCED PARTITIONING problem on tree instances. Typically, NP-hard graph problems become trivial when restricted to trees. For k-BALANCED PARTITIONING however we showed that trees are an interesting benchmark, for various reasons.

When a perfectly balanced solution is required, we showed that even if the maximum degree of the tree is constant the k-BALANCED PARTITIONING problem is hard to approximate. Even more disheartening, on unrestricted unweighted connected trees no approximation algorithm can substantially outperform the trivial one that cuts all edges of the tree. This behavior matches that of general graphs, for which no algorithm with bounded approximation ratio exists. In this sense, trees represent a simple unit that still captures the full complexity of the problem.

On the other hand, if one settles for near-balanced solutions, trees prove to be "easy" instances, which admit a PTAS with approximation $\alpha = 1$, the best possible in the bicriteria sense. This crucial fact enables our PTAS for trees to be extended into an algorithm for general graphs with approximation factor $\alpha \in \mathcal{O}(\log n)$, improving on the best previous [8] bound of $\alpha \in \mathcal{O}(\log^{1.5}(n)/\varepsilon^2)$. Hence, remarkably, the same approximation guarantee can be attained on the cut cost for the k-BALANCED PARTITIONING problem in case $k = 2$ (the BISECTION problem) and for unrestricted k, if we settle for near-balanced solutions in the latter case. This is to be contrasted with the strong inapproximability results for the case of unrestricted k when the solutions are to be perfectly balanced.

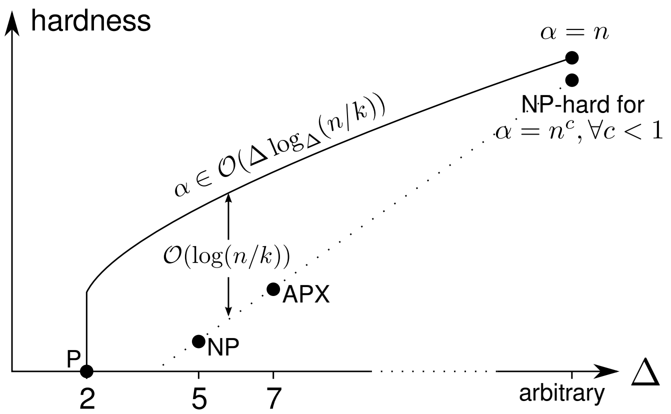


Figure 2.10: The hardness of trees versus their maximum degree Δ. The hardness results from Section 2.3 (large dots) indicate that the hardness grows with Δ (dotted line). A modification of MacGregor's [90] algorithm yields an approximation of $\mathcal{O}(\Delta \log_\Delta(n/k))$ (solid line). This means there is a gap of size $\mathcal{O}(\log(n/k))$ between the lower and upper complexity bounds in case Δ is constant.

Open Problems. For perfectly balanced partitions of trees it remains open to generalize our results to show a tighter dependency of the hardness on the degree (Figure 2.10). In addition, the possibility of an approximation algorithm for perfectly balanced partitions with a better ratio than $\alpha \in \mathcal{O}(\Delta \log_\Delta(n/k))$, as provided by the greedy scheme by

MacGregor [90], remains open. In particular, Theorem 4 does not rule out an algorithm that approximates the cut cost by the factor $\alpha = 25/24$ if $\Delta \leq 7$.

For near-balanced partitions it remains to be seen what other graph classes permit better approximation ratios on the cut size than those known for general graphs. It is worth mentioning though that the dynamic program of the PTAS presented in this chapter can be generalized to graphs with bounded tree-width, using standard techniques [116]. Hence also for these graphs a factor of $\alpha = 1$ is achievable.

Another performance metric that may be improved in our results is the runtime. We showed that we can achieve approximations on the cut cost that do not depend on ε, whereas the dependency on ε in the runtime is quite conspicuous. This begs the question whether an algorithm offering a better tradeoff in the dependency of ε between runtime and cut cost can be found. A recent result [44] however shows that there is no hope for a reasonable algorithm of this sort. More precisely it is shown that, unless P=NP, for trees there is no *fully* polynomial time algorithm with the following properties: The computed solution is near-balanced and the cut cost may deviate from the optimum of a perfectly balanced solution by $\alpha = n^c/\varepsilon^d$, for any constants c and d where $c < 1$.

Another main challenge we see for general graphs is to resolve the discrepancy in complexity between the case $\varepsilon = 1$ and the case $\varepsilon < 1$, studied in this chapter (recall Figure 2.4). For the case $\varepsilon = 1$ the algorithm by Krauthgamer *et al.* [83] achieves factor $\alpha \in \mathcal{O}(\sqrt{\log n \log k})$ and in the same paper it is shown that a dependency of

α on k is unavoidable. Proving similar results for the case $\varepsilon < 1$ seems difficult to achieve, as the spreading metric techniques generally used for $\varepsilon = 1$ do not extend to $\varepsilon < 1$. Furthermore, the graph decomposition results that we used to achieve an $\mathcal{O}(\log n)$ approximation do not seem amenable to leading to algorithms with $o(\log n)$ approximation factor. Therefore it is likely that radically new techniques need to be developed to resolve the discrepancy.

Chapter 3

Online Approximation of Time Series with Applications*

3.1 Introduction

Analyzing an online stream of measurements or transactions is a fundamental problem in database systems. Indeed, with the rapid growth and adoption of various tracking and monitoring technologies, database systems are under constant pressure to process in real-time the large volumes of data feeds that are generated by sensors embedded in environments, network infrastructures, financial markets, and so on. In many of these applications, the data is best thought of as a *time series*—an *ordered* sequence of measurements, where the order can be temporal as stock trades in financial markets,

*Parts of this chapter appeared in the following publication: [56]

series of sensor measurements, retail transactions, etc., or spatial as the linear order defined by isocontours in some physical sensor measurements, or some other externally imposed order.

The goal of an online time-series approximation scheme is to summarize the data in one-pass and construct an approximate representation that can support a variety of time-series queries with bounds on worst-case error. Given the importance of time-series data, many approximation methods are known [23, 74, 76, 82, 109], although research in *online* approximation and worst-case error bounds is relatively recent [17, 59, 60, 61, 62]. Some of these schemes such as those based on wavelets can require an expensive *post-processing* phase to reduce a larger set of wavelet coefficients to a B-bucket approximation. This includes the current theoretical best scheme due to Guha [59, 60] which achieves $(1+\varepsilon)$ approximation but requires $O(B^3\varepsilon^{-2}\log^2 n)$ time post-processing to cull down the $O(B \log n)$ wavelet coefficients into a B-bucket representation. In this work, we are particularly interested in combinatorial approximation schemes that maintain a *query-ready* synopsis in real-time—that is, at all times, the approximation is available for query processing with no additional computational overhead. In this regard, our approach is similar to the methods of [17, 104]. As discussed by Guha [59], when wavelets are used directly as a queriable structure [22], one only gets an approximation quality of the form $(O(1), \log n)$ in the standard (α, β) terminology; that is, α times the error of the optimal scheme using β times the optimal number of buckets. In order to achieve $(O(1), O(1))$ approximation, one requires the expensive post-processing mentioned earlier [59, 97].

Our main contribution is the introduction of a *generic* geometric method for approximating a one-dimensional signal that has the following properties: (1) at all times, it maintains a *query-ready* piecewise linear approximation of the time-series seen so far; and (2) as long as the error metric satisfies some weak conditions (non-negativity, monotonicity and additivity), the approximation provides (α, β)-type worst-case optimality bounds. More specifically, we show that a simple *greedy bucket-merge* method that has been used in specific settings (maximum error [17] and euclidean distance based histograms [75] and maximum error amnesic approximation [104]), sometimes without any accompanying worst-case error analysis, can be *abstracted* to a generic form and applied more broadly with worst-case approximation bounds where none were known before [75, 104], and simpler and unified algorithms in other cases. In particular, we show that, besides the data stream model, the new generic scheme is easily adapted to two other variants of time-series approximation that have received significant attention lately: *out-of-order* streams and *amnesic* approximation.

The *out-of-order* stream model is motivated by the asynchrony in many distributed and networked data processing applications [27], such as sensor networks. When the data is produced at dispersed locations and then transmitted to a collection or processing node, data values may not arrive in the strict time-order in which they were recorded due to various networking delays and asynchronous communication. An approximation scheme for the out-of-order stream should be tolerant of a small number of *gaps* in the arrival sequence. We show that our approximation scheme works for the out-of-order time series model, increasing the memory requirement only by an additive term $O(k)$, where k is the number of gaps in the time series at the current time.

An *amnesic* approximation is motivated by the fact that in most applications of time-series, *the value of data decreases with age* [100, 104]. An analyst is likely to be most concerned with the recent data, and as the data gets old, his tolerance for the error of approximation grows. Thus, given a fixed memory budget, a natural way to optimize its use in the time-series approximation is to grow the error tolerance with age, so that progressively smaller fraction of the representation is spent on older data. While there have been several models for aging the data (exponential time decay, polynomial time decay, etc), our approach is general, and works with any given *monotone* amnesic function. Specifically, given a piecewise constant monotone *amnesic function*, describing the error tolerance over the entire history, our scheme achieves a constant factor approximation. We begin with some notation to introduce our problems, followed with a statement of our results.

3.1.1 Preliminaries and Contributions

We assume that a time series is given as a sequence of value $S = < v_1, v_2, \ldots, v_n >$, where each value v_i is associated with a *timestamp* t_i. For our purposes, the timestamps are an abstract entity with monotonically increasing values. We are not concerned with whether the timestamps reflect the time at which the data values are generated (source) or the time at which they are processed (sink)—any consistent view of timestamps will suffice. In the stream model, we only require that before the item with timestamp t_i arrives, all items with timestamps $t_i < t_j$ have arrived, but this assumption is relaxed in the out-of-order stream model.

We consider the problem of maintaining an approximation of the time series. A popular approximation method is to partition the data into consecutive blocks called *buckets* and represent every bucket by a single value, called a piecewise constant (PC) approximation. Instead, if we approximate the values of each bucket by a linear function (line segment), we get the piecewise linear (PL) approximation. The quality of an approximation is measured by error between the true time series and its approximated version, and L_p norm is a popular form of measuring the error. Specifically, the L_p norm error of the approximation is defined as follows:

$$E_p = \left(\sum_{i=1}^{n} |v_i - \hat{v}_i|^p \right)^{1/p} \tag{3.6}$$

where $\hat{v}_i$ is the approximation for value v_i. The L_∞ norm is defined as:

$$E_\infty = \max_{i=1}^{n} |v_i - \hat{v}_i| \tag{3.7}$$

Our primary focus is on L_∞ and L_2 norms.

The performance of our algorithms are expressed using the (α, β) approximation guarantees, meaning we achieve α times the error of the optimal scheme using β times the optimal number of buckets. Our main results are summarized below.

1. We show that a generic bucket-merge scheme achieves a $(2, 4)$ approximation for a broad class of error metrics that satisfy non-negativity, monotonicity and additivity.

2. For commonly used metrics such as L_2 and L_∞ and piecewise constant or piecewise linear approximations, our scheme admits a simple and highly efficient streaming implementation that compares favorably with the existing methods. In particular, for the L_2 error, our algorithm maintains a B bucket approximation using $O(B)$ working space and $O(\log B)$ update time. While our bounds are worse than the best known theoretical result [59, 60], our scheme delivers comparable approximation quality in simulations, *with the advantage of being significantly faster.*

3. We show that our scheme easily adapts to the amnesic model of time series [104]. In particular, this adaptation called AMNESIC-MERGE achieves $(1 + \varepsilon, 3)$ approx-

imation using $O(\varepsilon^{-1/2}B\log(1/\varepsilon))$ memory and $O(\log B + \varepsilon^{-1/2}\log(1/\varepsilon))$ per item update time for the piecewise linear approximation. For piecewise constant approximation AMNESIC-MERGE achieves a $(1, 3)$ approximation using $O(B)$ memory and $O(\log B)$ per item update time. (This problem is called the UAA problem in [104], using piecewise constant monotonic amnesic functions.)

4. We also show that our scheme extends easily to out-of-order streams, where the data values may not arrive in their intended (stamped) order. Our scheme maintains piecewise approximations (PC or PL) for only the data that have arrived while still ensuring worse-case bounds on the error of approximation. (The need to bound approximation error requires that no bucket should include a "missing value" because otherwise an adversary can force an unbounded error by assigning arbitrarily large values to the missing data in the future.) Our variant, called the FRAGMENT-MERGE achieves $(\sqrt{2}, 5)$ approximation for the L_2 error using $O(B)$ space and $O(\log B)$ update time, and $(1 + \varepsilon, 3)$ approximation for L_∞ error using $O(\varepsilon^{-1/2}B\log(1/\varepsilon))$ space and $O(\log B + \varepsilon^{-1/2}\log(1/\varepsilon))$ update time.

5. We perform extensive simulations to validate the empirical performance of our schemes on a variety of synthetic and real-world datasets. Our results confirm the space and time efficiency of our algorithms, which together with its simplicity and generality make it well-suited for many applications.

This chapter is organized as follows. In Section 4.2, we discuss related work. In Section 3.3, we describe the GENERIC-MINMERGE algorithm and its error analysis. Section 3.4, 3.5, and 3.6 discuss applications of this generic technique: stream histograms, amnesic approximations and out-of-order streams. In Section 3.7, we discuss our simulation results.

3.2 Related Work

There is a large body of research on streaming histograms in the database community. A non-streaming dynamic programming scheme for constructing a B-bucket histogram was originally proposed by Jagadish et al. [69], with time complexity $O(n^2B)$ and space $O(nB)$, which was subsequently improved by Guha [59] to $O(n^2B)$ time and $O(n)$ space. Currently, the best worst-case bounds for streaming histogram construction are due to Guha and his colleagues [58, 59, 60, 61, 62]. Their techniques tend to maintain a summary structure of small memory, require $O(1)$ processing time per item, and then require some post processing at the end to compute the final approximation. Specifically, for the L_2 metric, they achieve a $(1+\varepsilon, 1)$ approximation in $O(n + B^3\varepsilon^{-2}\log^2 n)$ time with $O(B\varepsilon^{-2}\log n \log \varepsilon^{-1})$ space [59]. A very recent result of Guha [60] removes the dependence on $\log n$ from the space bounds.

Since many real-time applications deal with an intermixed sequence of updates and queries, an expensive post-processing before each query can be a bottleneck. As a result, we seek representations that dynamically maintain an approximation, always ready to process queries. In [17], Buragohain et al. propose fast lightweight algorithms for query-ready approximations, but their schemes only works for the L_∞ error metric. For other error metrics we point the reader to [59] as a good survey for existing techniques. Many time-series approximation scheme use a natural version of greedy bucket merging [104, 75, 17] in which the algorithm repeatedly greedily merges two buckets whenever it needs to allocate a new bucket. In particular, the GRAP-R algorithm proposed for the L_2 metric in [104] is very similar to the merging scheme proposed in this chapter, but it lacks a worst-case error analysis. In the computational geometry community, several algorithms are known for curve simplification using Hausdorff and Frétchet metrics, but their focus is not on time series.

The amnesic approximation problem was proposed in [104], where the authors perform an extensive study and show very good results in practice. But they offer no worst case guarantees. In fact, our merge scheme is a minor variation (controlling when the merge happens) but we can prove worst-case approximation bounds for our version.

In the streaming setting, given B buckets as the input parameter, Abam et al. [6] propose an algorithm which achieves a $(3, 2)$ approximation for xy monotone curves in $O(B)$ space and $O(\log B)$ update time and the algorithm works for Hausdorff error metric. For arbitrary paths, it achieves a $(4\sqrt{2} + \varepsilon, 2)$ approximation for Fr$\acute{e}$tchet error metric in space $O(B^2\varepsilon^{-1/2}\log^2(1/\varepsilon))$ with update time $O(B\varepsilon^{-1/2}\log^2(1/\varepsilon))$ per value. For more references we point the reader to [6].

Out-of-order stream processing has been a topic of interest in many recent works in the database community. There have been new architectures proposed for handling out of order processing [86, 87] for a variety of problems, with the main concern being memory footprint. Many researchers have proposed heuristics [11, 118] and guaranteed algorithms [18, 27] for answering various types of queries for out-of-order data including quantiles, heavy hitters, sliding window counts. We solve the problem of maintaining histograms over out-of-order streams which is a well known aggregation technique for answering queries, and we show that the memory footprint required for our technique is very small.

Finally, Nath et al. [100] discussed the pros and cons of amnesic approximations for storing data on flash memory. They use lazy compression, which compresses data only when it is full, whereas our focus is to maintain an approximation with guarantee at all times.

3.3 The Generic-Minmerge

We want to compute a B bucket approximation of a time series of values $S = < v_1, v_2, \ldots, v_n >$, where B is an input parameter. The piecewise constant (resp. linear) approximation means that the values in each bucket are approximated by a single value (resp. linear function). The *error of the approximation* is typically measured by the L_p norm distance between the original time series and its approximated version. We will use the notation $e(S, b)$ to denote the optimal error in approximating the time series S using b buckets; the specific L_p norm and the approximation type (constant or linear) will be clear from the context. We begin with an abstract formulation of the problem that attempts to capture the essence of a *greedy bucket-merge* that is used in practice. We say that the piecewise approximation (constant or linear) for an error metric is *well-behaved* if the following three conditions hold:

1. *Nonnegativity*: The error is always non-negative.

$$e(S', b) \geq 0, \quad S' \subseteq S, \quad b \geq 1 \tag{3.8}$$

2. *Monotonicity*: The error of approximating $S_1 \cup S_2$ using one bucket is at least as large as the error in approximating S_1 and S_2 separately with one bucket each.

$$e(S_1 \cup S_2, 1) \geq e(S_1, 1) + e(S_2, 1) \tag{3.9}$$

   Figure 3.11 illustrates this property with an example.

3. *Additivity*: The error of a B-bucket approximation is the sum of the errors of the individual buckets. (Please see the remark below.)

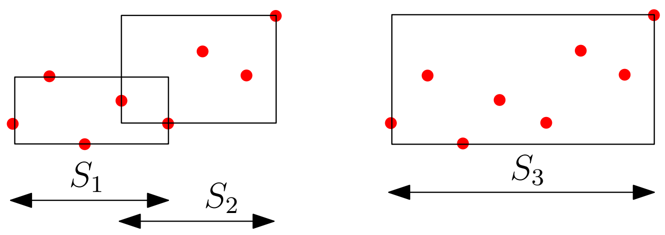


Figure 3.11: Illustrating monotonicity, using L_∞ error. The error for the combined set with one bucket is at least as large as the error of two subsets.

Remark: One can verify that PC or PL approximations using all L_p norms (including L_∞) are well-behaved—simply use the pth power of the L_p norm as the error e to see that additivity and monotonicity hold. In the limit as $p \to \infty$, this holds for the L_∞ norm as well, but one can also replace sum with $\max$ in appropriate places in the proofs to get results for the L_∞ norm as well. The Hausdorff metric is also well-behaved, but the Frétchet is not [6].

3.3.1 The Algorithm

The generic merge scheme, denoted GENERIC-MINMERGE is explained in pseudo-code in Algorithm 1. It takes as input a time series S and B, the number of buckets. At any time, the buckets impose a partition of S into intervals, where two buckets are *adjacent* if they represent two consecutive intervals of this partition. Our approximation maintains a partition with at most $4B$ buckets. When a new item arrives, it is placed in its own (newly allocated) bucket. *If this new bucket causes the number of buckets to exceed 4B, we shrink the number of buckets by one by merging an appropriate pair*

Algorithm 1: GENERIC-MINMERGE (S, B)

```
1: $\mathcal{M} = \emptyset$
2: for $next\ v_i \in S$ do
3:     Allocate new bucket $u$
4:     $\mathcal{M} = \mathcal{M} \bigcup \{u\}$
5:     if $|\mathcal{M}| \geq 4B$ then
6:         $\{v_1, v_2\}$ = BEST-ADJPAIR($\mathcal{M}$)
7:         $v :=$ MERGE($v_1, v_2$)
8:         $\mathcal{M} = \mathcal{M} \setminus \{v_1, v_2\}$
9:         $\mathcal{M} = \mathcal{M} \bigcup \{v\}$
10:    end if
11: end for
```

of adjacent buckets. In particular, we merge those two adjacent buckets that result in a bucket with smallest error. This is achieved by functions BEST-ADJPAIR and MERGE. An illustration of one iteration of the algorithm is shown in Figure 3.12. In the next section we prove that this scheme achieves the $(2, 4)$ approximation.

3.3.2 Analysis of Generic-Minmerge (GM)

We begin by stating a crucial property, called *Min-Merge* maintained by the buckets produced by the GM algorithm at any time. *An approximation P is said to obey the*

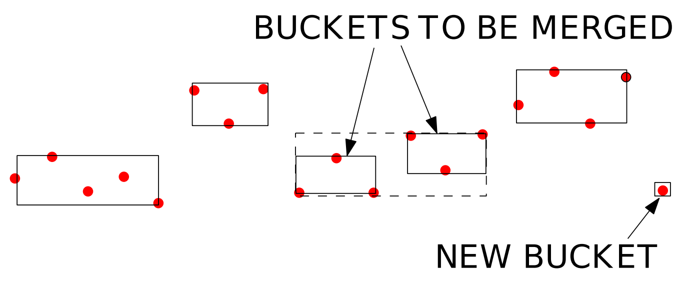


Figure 3.12: Illustrating Min-Merge. The new item is put in a new bucket and the two buckets whose merger leads to the least error bucket are merged (shown with a dashed box).

Min-Merge property if the approximation error of a bucket formed by merging any two adjacent buckets in P is greater than (or equal to) the approximation error of any of the buckets before the merge. Then by Algorithm 1 we have the following.

Lemma 3.3.1. The buckets in the GENERIC-MINMERGE algorithm obey the Min-Merge property.

Proof: The proof is by induction on i, the index of the last input value arrived. The base case is established by noticing that when $i \leq B$ then each bucket b_j has error $e(b_j) = 0$.

For the inductive step, we assume that the condition holds after v_i $(i > B)$ has been processed and prove it continues to hold after v_{i+1} is processed by the GENERIC-MINMERGE algorithm.

If $i > B$ the GENERIC-MINMERGE algorithm must merge two adjacent buckets into a new bucket b' after receiving v_{i+1}. We consider two cases. Either the GENERIC-MINMERGE merges the Bth and the $(B+1)$th bucket (i.e., the bucket newly allocated for the single v_{i+1}) or it merges some other pair of buckets present before v_{i+1} had been inserted. In the former case, the newly created $b' = merge(b_B, b_{B+1})$ bucket has error $e(b')$ greater than or equal to $e(b_B)$ before inserting v_{i+1} therefore, since merge involving the Bth bucket before v_{i+1} was inserted obeyed the min-merge property by the inductive hypothesis, so it will *a fortiori* when b_B is extended to include $b_{B+1} = v_{i+1}$. Suppose otherwise that two buckets b_j, b_{j+1}, $j \neq B$, are merged following the insertion of v_{i+1}. This means that $e(merge(b_B, b_{B+1}))$ is greater than the error of the two merged buckets (otherwise b_j and b_{j+1} would have been merged by the algorithm, instead), therefore it holds that $e(merge(b_j, b_{j+1})) \geq e(merge(b_B, b_{B+1})) \geq e(b_x) \forall x$ where the last part of the chain of inequalities is given by the inductive hypothesis. □

We now prove the main result of this section.

Theorem 3.3.2. For any well-behaved error metric, GENERIC-MINMERGE achieves a (2, 4) approximation.

Proof: We present our proof when the objective is to minimize the *sum* of the bucket errors; the proof for the *maximum* error follows by changing summation to max. Let E^* denote the optimal error of approximation using B buckets, and let E denote the error achieved by the GM algorithm using $4B$ buckets. We will show thaot

$$E \leq 2E^*. \tag{3.10}$$

For ease of reference, we will denote the buckets in the optimal solution as optimal buckets, and the buckets of the latter as the merge buckets: in both solutions, the buckets partition the set into intervals. Now, the B optimal buckets can *intersect* at most $B-1$ merge buckets, where the intersection means that an endpoint of the optimal bucket lies in the interior of the corresponding merge bucket. Let us call a merge bucket intersected by the optimal bucket a *split bucket*, and let $b_s \leq B-1$ be the number of split buckets. Let E_s be the total error associated with these b_s split buckets. We call the remaining merge buckets the *non-split buckets*, and let E_n be the total error associated with these buckets. Now, since the split and non-split buckets account for all the buckets of the GM algorithm, we have

$$E \leq E_s + E_n \tag{3.11}$$

(This is an equality for L_p norms, for finite p, and the max can be substituted for the sum for the L_∞ norm.)

We now argue that

$$E_n \leq E^* \tag{3.12}$$

Let $\{T_1, T_2, \ldots, T_B\}$ denote the partition of S by the B optimal buckets. That is, T_i is the subset approximated by the ith bucket in the optimal representation. Similarly, let $\{T_1', T_2', \ldots, T_{b_n}'\}$ denote the sets approximated by the b_n non-split buckets of GM. Observe that for every j, we must have an i such that $T_j' \subseteq T_i$, although some T_i sets may not have any T'_j as subset. Let Y be the set of indices such that $i \in Y$ iff for some j, $T'_j \subseteq T_i$. Thus, we have

$$\sum_{i \in Y} e(T_i, 1) \geq \sum_{j=1}^{b_n} e(T'_j, 1) \tag{3.13}$$

This follows from the monotonicity property because each non-split bucket lies inside some optimal bucket. Finally, since $E^* \geq \sum_{i \in Y} e(T_i, 1)$, we get that $E_n \leq E^*$.

Next, we argue that $E_s \leq E^*$. If $\{T''_1, T''_2, \ldots, T''_{b_s}\}$ are the subsets approximated by the split buckets, then we clearly have that

$$E_s = \sum_{i=1}^{b_s} e(T''_i, 1) \tag{3.14}$$

Call a pair of adjacent non-split buckets a *witness pair* if those buckets lie in a common optimal bucket. Call a set of witness pairs *disjoint* if no two pairs share any (non-split) buckets. We will argue shortly that there always exists a set of b_s disjoint witness pairs, but for now let's assume that and complete the proof that $E_s \leq E^*$.

Let us denote the set of values approximated by the ith disjoint witness pair by the set X_i. Since the buckets obey the Min-Merge property (Lemma 3.3.1), we must have

$$\sum_{i=1}^{b_s} e(X_i, 1) \geq \sum_{i=1}^{b_s} e(T''_i, 1) \tag{3.15}$$

Each of these X_i sets is a subset of some set T_j, where some sets might have multiple X_is as subsets. Let Z be the set of indices j such that $j \in Z$ iff T_j has some X_i set as its subset. Then, using monotonicity we get,

$$\sum_{j \in Z} e(T_j, 1) \geq \sum_{i=1}^{b_s} e(X_i, 1) \tag{3.16}$$

By the non-negativity and additivity of the error metric, we have

$$E^* \geq \sum_{i \in Z} e(T_i, 1) \tag{3.17}$$

Using equations 3.14, 3.15, 3.16, 3.17 we get,

$$E_s \leq E^* \tag{3.18}$$

Remember that b_s is at most $B-1$. Now we show that our assumption of there being at least b_s disjoint witness pairs is true. Let us say not, and without loss of generality let us say there are exactly $B-2$ disjoint witness pairs. There are a total of $4B$ buckets, among which there are at most $B-1$ split buckets. Hence we are left with at least $3B+1$ non-split buckets. Among these there are only $B-2$ disjoint witness pairs (by assumption). Hence we are left with at least $3B+1-(2B-4) = B+5$ buckets. But, we have only B optimal buckets, hence by the pigeon hole principle, there must be at least one more disjoint witness pair, which makes our assumption incorrect.

Equations 3.11, 3.12 and 3.18 give us the 2 approximation on the error, we used $4B$ buckets, where B is the optimal, and this completes the proof that the GM algorithm gives a $(2,4)$ approximation if the error metric is well behaved. $\square$

The error analysis of the generic merging scheme holds for all well-behaved error metrics, which include all L_p norms as well Hausdorff distance. In fact, for the L_p norms by substituting E for its p-th power in the proof above, we can show the following improved bound.

Theorem 3.3.3. For the L_p error metric, GENERIC-MINMERGE achieves a $(2^{1/p}, 4)$ approximation.

Remark: A streaming implementation of GM requires a space-efficient implementation of the two key functions: BEST-ADJPAIR and MERGE. In the following, we show that for the two most popular norms (L_2 and L_∞), these functions can be implemented in the streaming model with small memory footprints and can also be extended to amnesic and out-of-order models. On the other hand, we do not know how to realize these functions for other L_p norms in the streaming model: for instance, computing the optimal L_1 approximation of a time series requires maintaining (and updating) the *median* of streaming subsets. Whether one can use *approximations* of the bucket errors and still obtain the global error bound for the generic minmerge remains an interesting open question.

3.4 Histograms for L_2 and L_∞

To evaluate the functions BEST-ADJPAIR and MERGE in the streaming model, we need to compute the L_2 or L_∞ error for the union of two buckets without explicit knowledge of the values in those buckets. For the L_∞ error, Buragohain et al. [17] show how to implement the greedy bucket-merge in the streaming setting. However, their technique and analysis is limited to L_∞ error only, which can be computed by maintaining the axis-aligned bounding box for PC histogram and (approximate) convex hulls for the PL histogram. Because of their specialized application, Buragohain et al. [17] are able to obtain the following tighter analysis for the greedy bucket-merge:

Theorem 3.4.1 ([17]). For the L_∞ error metric,

- the GM algorithm achieves a $(1, 2)$ approximation for the streaming PC histogram using $O(B)$ space and
  $O(\log B)$ update time for each value.

- the GM algorithm achieves a $(1+\varepsilon, 2)$ approximation for the streaming PL histogram using $O(\varepsilon^{-1/2} B \log(1/\varepsilon))$ memory and $O(\log B + \varepsilon^{-1/2} \log(1/\varepsilon))$ update time for each value.

In the following, we show that our generic framework allows us to extend these results to L_2 norm. In particular, we show that the L_2 error for each bucket can be encoded in $O(1)$ space, and this representation allows us to compute the L_2 error of the bucket obtained by merging two adjacent buckets in $O(1)$ time. Specifically, let us consider two adjacent buckets, representing the sets of items S_1 and S_2, where $S_1 \cap S_2 = \emptyset$. Suppose the sequence of values is $v_1, v_2, \ldots, v_n$, with associated time stamps $t_1, t_2, \ldots, t_n$. Then, it is well-known that the optimal linear approximation under the L_2 error for the sequence $v_1, v_2, \ldots, v_n$ is achieved by the *least squares line* fitting the two-dimensional set of values $(t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n)$. The slope and the intercept of this line can be computed by the following four sums: $\sum v_i$, $\sum v_i^2$, $\sum t_i$ and $\sum v_i t_i$, which can be maintained in $O(1)$ space and updated for the union of two adjacent buckets in $O(1)$ time. The proof of the following can be found in [24] and [104].

Lemma 3.4.2. Given two adjacent intervals sets S_1 and S_2 and their optimal least square approximations, we can obtain the optimal least square approximation for $S_1 \cup S_2$ in $O(1)$ time and $O(1)$ space. The exact value of the L_2 norm error of $S_1 \cup S_2$ can be computed from the errors of the buckets for S_1 and S_2 in $O(1)$ time and $O(1)$ space.

With this result, we can evaluate the key functions of the GM algorithm efficiently in the streaming model. Specifically, MERGE can be implemented $O(1)$ time by storing $O(1)$ information about each bucket, while BEST-ADJPAIR function can be implemented using a heap in $O(\log B)$ update time per value. Thus, we have the following result.

Theorem 3.4.3. Given a time-series, we can compute a streaming $(\sqrt{2}, 4)$ approximation of its B-bucket PC or PL histogram under the L_2 norm using $O(B)$ space and $O(\log B)$ update time for each value.

3.5 L_∞ Amnesic Approximation

In an *amnesic* (or, time decaying) approximation, recent data are approximated more faithfully than older data. In other words, the approximation error is allowed to grow with the age of the data. The streaming amnesic approximations have received considerable attention lately as histogram summaries have become an important tool for online analysis of data [100, 104]. We show that our generic minmerge framework is well-suited for amnesic approximation as well.

An amnesic representation is controlled by dividing the data into *age groups*, where each age group is assigned an upper bound on the approximation error. We take a general approach and assume that the amnesic representation is specified as a piecewise constant function, which is monotone, meaning that the approximation errors grow with age. In particular, a monotone piecewise constant amnesic function partitions the time series into time-intervals $\{[1, \tau_1], (\tau_1, \tau_2], \ldots, (\tau_{m-1}, \tau_m]\}$, with each interval $(\tau_i, \tau_{i+1}]$ associated with an error tolerance e_{i+1}, where $e_i \leq e_{i+1}$. Thus, the amnesic function $f(x)$ can be described as:

$$f(x) = \left\{ \begin{array}{l} e_1, \ \ 1 \le x \le \tau_1 \\ \\ e_2, \ \ \tau_1 < x \le \tau_2 \\ \vdots \\ \\ e_m, \ \ \tau_{m-1} < x \le \tau_m \end{array} \right\} \tag{3.19}$$

See Figure 3.13 for an example—the top part shows an example stream approximated within the error bounds specified by the amnesic error function shown at bottom.

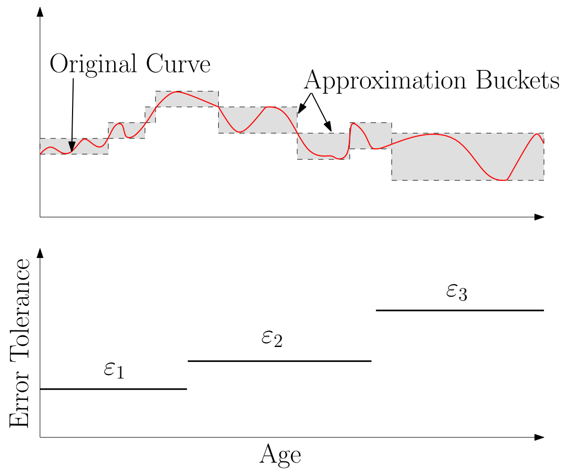


Figure 3.13: Amnesic approximation. The top figure shows increasing bucket heights (L_∞ error) with increasing data age, and the bottom figure shows a monotone amnesic error function, with 3 age groups.

Next, we describe the amnesic version of our generic merge algorithm and prove its error bounds. We consider only the L_∞ norm.

3.5.1 Amnesic-Merge (AM) Algorithm

Let the number of age groups in the amnesic function be m, where we assume that m is a small constant and $m = o(B)$. Then, our problem is the following: *Given a monotone piecewise constant amnesic function $f()$, and a stream $S =< v_1, v_2, \ldots, v_n >$ of values, maintain an approximation minimizing the number of buckets used such that $|\hat{v}_i - v_i| \leq f(x_i)$ for all i, where $\hat{v}_i$ is the approximated value for v_i.* The main conceptual difficulty in this *amnesic approximation problem* arises from the fact that the age of each item changes every time instant. The simple but key observation in our solution is the fact that only those buckets that cross the age group boundaries require special care; the remaining ones behave as in the standard streaming model. This requires only a minor change to the generic minmerge, and we call this variant the AMNESIC-MERGE (AM) algorithm:

1. Allocate a bucket for the next value in the stream.
2. For each age group with two or more buckets in it, if merging any two adjacent buckets produces error less than or equal to the allowable error in the age group, merge the two buckets.
3. Go to Step 1.

A bucket is in an age group if its starting timestamp lies in the age group. This algorithm is nearly identical to the algorithm proposed in [104] with one minor difference: in that algorithm, the merge *precedes* the allocation of the new bucket, while in our algorithm these steps are reversed.

Analysis of the Amnesic-Merge (AM)

We first point out the problem formulation of the amnesic approximation is quite different from our earlier problem. There, we wished to minimize the error *given a number B of buckets*. In the AM algorithm, we want to minimize the number of buckets *given error bounds*. We show that if the optimal algorithm uses B^* optimal buckets for a given amnesic function, then the AM algorithm uses no more than $3B^*$ buckets, while achieving the same error guarantees for each age group. Thus, AM is a $(1, 3)$ approximation.

Lemma 3.5.1. The bucket approximation produced by the
AMNESIC-MERGE algorithm satisfies the L_∞ error bounds for each age group, and uses at most 3 times the optimal number of buckets.

Proof: Let us denote the number of buckets in age group i by b'_i, where age group i is the set of values with age in the range $(\tau_{i-1}, \tau_i]$. As mentioned earlier, the buckets can be thought of as a linear partition of the age axis. There are at most $m-1$ buckets which intersect with the partition obtained by the age groups defined by the amnesic function.

Every age group i has at least $b'_i - 1$ non-intersecting buckets. Let us denote the number of non-intersecting buckets associated with age group i by b''_i. Then,

$$b''_i \geq b'_i - 1 \tag{3.20}$$

Assume for now that $b''_i \geq 2$. Since we merge buckets only if they produce error less than the required error for the age group, the buckets in every age group are within the required error. Let us denote the set of values in age group i approximated by non-intersecting buckets by T''_i. Let us denote the maximum error in the non-intersecting buckets formed by AM algorithm in age group i by $e_m(T''_i, b''_i)$, where b''_i is the number of buckets used. Also, let us denote the maximum error in the buckets produced by the optimal algorithm in approximating set T''_i using b_i buckets by $e(T''_i, b_i)$. From the AM algorithm we know that if there was even one more merge in the b''_i non-intersecting buckets in age group i, the error will be more than the allowable error. Hence,

$$e(T''_i, \lfloor (b''_i - 1)/2) \rfloor \geq e_m(T''_i, b''_i - 1) > e_i \tag{3.21}$$

The first part of this equation follows from the fact that the buckets obey the Min-Merge property. This equation implies that even if one bucket is shared across every neighboring age group by the optimal algorithm, with less than $(\sum_i \lfloor (b'_i - 1)/2 \rfloor) - m$ buckets optimal algorithm cannot achieve the desired error in every range. Let $B = (\sum_i \lfloor (b''_i - 1)/2 \rfloor) - m$. We also know that the merge algorithm with at most $\sum_i (b'_i)$ buckets achieves the desired error in every range. Let $B' = \sum_i (b'_i)$. Using this and equation 3.20 we can show that

$B' \leq 2B + O(m) \leq 3B$. (The $O(m)$ terms also takes care of the case when $b'_i \geq 3$ does not hold, and since $m = o(B)$, the approximation still holds.) □

To implement the merging of two buckets and computing the L_∞ error of a bucket, we need to maintain the convex hull of the values in each bucket. In the worst case, the size of the convex hull for a set of n values is $O(n)$, but we can use the techniques presented in [17] to reduce this size to a user specified constant. We omit the details and point the reader to [17] for details, and summarize our result in the following theorem.

Theorem 3.5.2. The AMNESIC-MERGE algorithm achieves a $(1 + \varepsilon, 3)$ approximation for the amnesic approximation problem and uses $O(\varepsilon^{-1/2} B^* \log(1/\varepsilon))$ space and $O(\log B^* + \varepsilon^{-1/2} \log(1/\varepsilon)$ update time for each value, where B^* is the number of buckets in the optimal solution.

Remark: Palpanas et al. [104] also propose the dual version of amnesic approximation problem described above. In the dual version, we are given an input parameter B, and the constraints on the ratios of the tolerable errors in each age group. It is difficult to achieve bounds on this problem as the number of buckets that needs to be assigned to a contiguous piece of data might change in a non-monotonic manner depending on the incoming data. We believe that there is a negative result for this problem and intend to address this issue in a future study.

3.6 Out-of-Order Streams

We now discuss an extension of our GENERIC-MINMERGE that holds for approximating out-of-order streams. The *out-of-order* stream model is motivated by the asynchrony in many distributed and networked data processing applications [27], where data is sensed at dispersed locations and then transmitted to a central node for processing. Due to various networking delays and asynchronous communication, data values may not arrive in the strict time-order in which they were recorded. An approximation scheme for the out-of-order stream should be tolerant of a small number of *gaps* in the arrival sequence. A gap is maximal block of timestamps $[t_i, t_j]$ such that items with timestamps $t_i - 1$ and $t_j + 1$ have arrived but none of the items with timestamp t, for $t_i \leq t \leq t_j$, have arrived. We call the maximal block of timestamps between two consecutive gaps a *fragment*. Figure 3.14 shows a snapshot of an out-or-order stream with gaps and fragments.

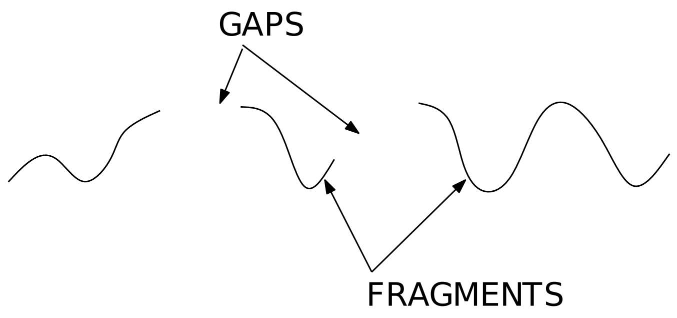


Figure 3.14: Snapshot of an out-of-order stream, showing gaps and fragments.

We are interested in PC or PL bucket approximations using L_2 or L_∞ error. A bound on the approximation error means that no bucket can include a gap—the missing value can be arbitrarily far off from the approximation. Thus, it is easy to see that if there are k *gaps* in the data seen so far, then an guaranteed error approximation requires at least $\Omega(min(k, n-k))$ space. In practice, we assume that the number of gaps is small, and show that our approximation uses space only linear in the number of gaps. Our algorithm for the out-of-order streams is called FRAGMENT-MERGE and is described below.

3.6.1 The Fragment-Merge (FM) Algorithm

Algorithm 2 gives the pseudocode for the FM algorithm. The value for the constant c is set to 2 for the L_∞ norm, and 4 for the L_2 norm. At a high level, the algorithm works as follows. On the arrival of a new value, a new bucket is allocated. Suppose the timestamp associated with the new value is t_i. Depending on how t_i relates to the gaps and fragments, we can have three cases.

1. A new fragment is formed if the values with timestamps t_{i-1} and t_{i+1} have not been seen so far.

2. The new value is adjacent to exactly one existing fragment—that is, either t_{i-1} or t_{i+1} have been seen, but not both.

3. The new value closes the gap between two fragments—that is, both t_{i-1} and t_{i+1} have been seen.

These three cases are shown in Figure 3.15. Depending on which category the new value falls in, the FM algorithm performs 0, 1 or 2 merges. The FRAG-BEST-ADJPAIR function returns the two buckets whose merge results in a bucket with least error among all adjacent pairs of buckets in all fragments. We analyze this algorithm in the next section.

Algorithm 2: FRAGMENT-MERGE(S, c, B)

```
1: $\mathcal{M} = \emptyset$, $k = 0$
2: for all $v_i$ do
3:    Allocate new bucket $u$
4:    $\mathcal{M} = \mathcal{M} \bigcup \{u\}$
5:    if $|\mathcal{M}| \geq cB + k$ then
6:       if Exactly one of the neighboring values of $v_i$ has been seen then
7:          $\{v_1, v_2\}$ = FRAG-BEST-ADJPAIR($\mathcal{S}$)
8:          $v$ = MERGE($v_1, v_2$)
9:          $\mathcal{M} = \mathcal{M} \setminus \{v_1, v_2\} \bigcup \{v\}$
10:      else if Two neighboring values of $v_i$ have been seen then
11:         $counter = 0$
12:         while $counter < 2$ do
13:            $\{v_1, v_2\}$ = FRAG-BEST-ADJPAIR($\mathcal{S}$)
14:            $v$ = MERGE($v_1, v_2$)
15:            $\mathcal{M} = \mathcal{M} \setminus \{v_1, v_2\} \bigcup \{v\}$
16:            $counter = counter + 1$
17:         end while
18:         $k = k - 1$
19:      else
20:         $k = k + 1$
21:      end if
22:   end if
23: end for
```

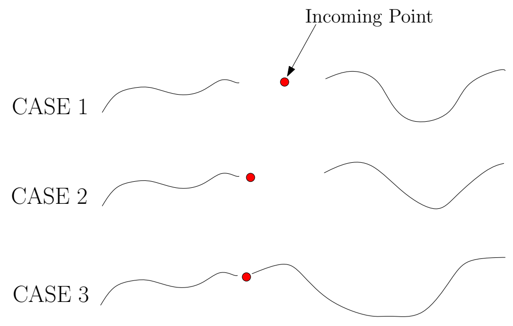


Figure 3.15: Three different configurations on the arrival of a new item: creation of a new fragment, extension of an existing fragment, and the merging of two existing fragments.

3.6.2 Analysis of Fragment-Merge (FM)

We detail the analysis for the L_∞ norm, and mention briefly the case of the L_2 norm. Because of the out-of-order arrival of items, there are two different notions of time: the timestamp, which represents the time at which data is generated (at the source), and the time at which it is received at the processing node (sink). We use the notation t to denote the source-time and τ to denote the processing or sink-time. Let k_i be the number of fragments at time instant τ_i. The analysis of FM rests on two lemmas: the first one proves that the buckets satisfy the Min-Merge property and bounds the size of approximation; the second does the error analysis.

Lemma 3.6.1. At any time τ_i, the FRAGMENT-MERGE algorithm maintains an approximation of size *at most* $2B + k_i$ buckets, where k_i is the number of fragments at time

τ_i. After each completed iteration of the algorithm, the buckets obey the Min-Merge property.

Proof: Our proof is by induction, on the length of the time series. For the base case, we have one value, so for any B, we have less than $2B + k_i$ buckets, where $k_i = 1$. We show that the lemma is true for all iterations of the algorithm after the first time instant τ_j where we have $2B + k_j$ buckets. Before the first such time instant the number of buckets is less than $2B + k_i$ for any instant τ_i and the Min-Merge property holds because there have been no merges. We assume that the claim is true for time instant τ_i and show that the claim is true for τ_{i+1}. We separately analyze the three cases mentioned earlier.

- Case 1. *The new value seen creates a new fragment of size 1*: The number of fragments $k_{i+1} = k_i + 1$. For the new value a separate bucket is allocated by the algorithm which has zero error. The number of buckets at this point is $2B + k_i + 1$, which is $2B + k_{i+1}$ and this proves the first part of the claim. Since the Min-Merge property was satisfied at time instant τ_i, it is also satisfied at time instant τ_{i+1} as the new bucket is not merged with any bucket.

- Case 2. *The new value increases the length of exactly one existing fragment by 1*: The number of fragments $k_{i+1} = k_i$. One merge operation is performed, so the number of buckets first increases to $2B + k_i + 1$ (allocation of bucket for new value), and then decreases with the merge to $2B + k_i$, which is the same as $2B + k_{i+1}$. There are two sub cases that can occur:

- Case a: In the merge operation the new bucket gets merged with adjacent bucket, say v. Let us say that the bucket formed is u. Then, the error of bucket u is no less than the error of bucket v (monotonicity of L_∞ error). Hence, the error in the bucket formed by further merging of u will be no less than the resultant error if v had been merged. But merging v again would itself result in a bucket with highest error. This is true as Min-Merge property holds at time τ_i. Hence the Min-Merge property holds at time τ_{i+1}.
- Case b: In the merge operation, the new bucket is not merged. Then let us say buckets u' and v' get merged and the error in the resultant bucket is e. But this implies that any further merge (including merge involving new bucket), will give a bucket with error greater than e, which is the maximum error in the approximation after $(i + 1)$th iteration. Hence after this one merge Min-Merge property is satisfied.

- Case 3. *The new value joins two previously disjointed fragments*: This case is similar to Case 2 and its proof is omitted.

This completes the proof. □

The second lemma establishes the quality of the approximation formed by FM algorithm with respect to optimal.

Lemma 3.6.2. At any time τ_i, the approximation error of the FRAGMENT-MERGE algorithm using $2B + k_i$ buckets is no worse than the optimal approximation error using B buckets, where k_i is the number of fragments at time τ_i.

Proof: Let us denote the number of buckets used by the optimal algorithm for fragment j by b_j, where $1 \leq j \leq k_i$ and $\sum_{j=1}^{k_i} b_j = B$. Let the error associated with the optimal approximation using B buckets be E^*. Let us denote the set of values in fragment j by F_j. Let the error associated with optimal approximation of fragment F_j using b_j buckets be $e(F_j, b_j)$, and hence $E^* = \max_{j=1}^{k_i} e(F_j, b_j)$.

Let us denote the number of buckets used by the FM algorithm for the jth fragment by b'_j, where $1 \leq j \leq k_i$ and $\sum_{j=1}^{k_i} b'_j \leq 2B + k_i$. Let the corresponding error associated with fragment j be $e_m(F_j, b'_j)$. The overall error of the approximation is $E = \max_{j=1}^{k_i} e_m(F_j, b'_j)$. We want to show that $E \leq E^*$. We will show the result by contradiction, and hence let us assume the following,

$$E^* < E \tag{3.22}$$

For the above equation to be true, there must exist at least one u, such that $b'_u < 2b_u$. This follows from the fact that every fragment follows the Min-Merge property (Theorem 3.4.1). Without loss of generality, let this be the fragment which results in the maximum error i.e.

$$E = e_m(F_u, b'_u) \tag{3.23}$$

Then, there must also be at least one fragment v, such $b'_v > 2b_v + 1$ (Pigeon hole principle). Now, consider the last algorithm iteration which resulted in the number of buckets for u to go from $b'_u + 1$ to b'_u. Note that there could be multiple such merges and we consider the last such merge before time instant τ_i, say at time instant τ_j. At time instant τ_j, fragment v has either more than $2b_v$ buckets, or less (equal to) than $2b_v$ buckets. We handle the arguments for these two cases separately.

- Case 1: If v has more than $2b_v$ buckets, say it has b''_v buckets, and let the fragment itself at that time be F'_v where $F'_v \subseteq F_v$, then

  $$e_m(F'_v, b''_v - 1) \leq e(F_v, b_v) \leq E^* \tag{3.24}$$

  This follows from the fact that the fragments obey Min-Merge property and the buckets obeying Min-Merge gives a $(1, 2)$ approximation for L_∞ error (Theorem 3.4.1). But, FM algorithm merges u before v, so error in fragment v after one merge, is no less than the error in fragment u when it has b'_u buckets,

  $$e_m(F'_v, b''_v - 1) \geq e_m(F_u, b'_u) \tag{3.25}$$

  Combining equations 3.23, 3.24 and 3.25, we get $E^* \geq E$, which contradicts our assumption.

- Case 2: At time instant τ_j, fragment v has less than (or equal to) $2b_v$ buckets. Then there must be a new value at some other time instant, say τ_l, where τ_l is after τ_j which results in v having more than $2b_v + 1$ buckets (after completed iteration of FM merge). This value can be either one which is next to one end of a fragment (in which case the proof is similar to that of Case 2, Lemma 3.6.1) or it can be the value which joins two fragments (in which case the proof can be derived similarly to Case 3, Lemma 3.6.1).

This concludes the proof. $\square$

With the natural assumption that $k_i \leq B$, we get that $2B + k_i \leq 3B$. The analysis above requires us to maintain the convex hull of the values in each bucket in order to determine the approximation error in the bucket. Using the approximate convex hulls as in [17], we can get an ε approximation. The main result for this section follows.

Theorem 3.6.3. The FRAGMENT-MERGE algorithm achieves a $(1+\varepsilon, 3)$ approximation for the piecewise linear approximation of an out-of-order stream under the L_∞ norm, using $O(\varepsilon^{-1/2} B \log(1/\varepsilon))$ space and $O(\log B + \varepsilon^{-1/2} \log(1/\varepsilon))$ update time for each value.

For piecewise constant approximation, the space and time bounds are $O(B)$ and $O(\log B)$ respectively, for a $(1, 3)$ approximation for the L_∞ norm. We can show a similar result for the L_2 approximation.

Theorem 3.6.4. The FRAGMENT-MERGE algorithm gives a $(\sqrt{2}, 5)$ approximation for the piecewise constant (or piecewise linear) approximation of an out-of-order stream under the L_2 norm using $O(B)$ space and $O(\log B)$ update time for each value.

3.7 Simulation Results

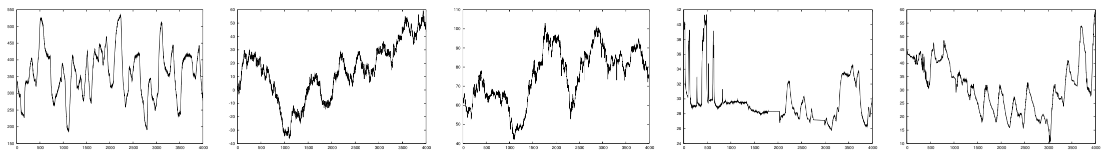


Figure 3.16: Left to Right: STEAMGEN, RANDOM, DJIA, HUM and TEMP

In this section, we report on the simulation-based performance evaluation of our three main algorithms, for computing histogram for streams (L_2 approximation), histograms for amnesic approximation (L_∞ approximation), and the out-of-order stream approximation. In particular, we implemented the following algorithms, including some optimal schemes for benchmarking. (All the code is written in Python or C++, and is available at www.cs.ucsb.edu/˜foschini/files/ooodelay.zip, along with the instructions to run it. All the benchmarks reported in this chapter are run on an Intel Core Duo Processor (@ 2.00GHz) equipped with 2GB RAM.)

- GENERIC-MINMERGE, AMNESIC-MERGE, FRAGMENT-MERGE: We implemented all three main variants of our generic scheme (GM, AM and FM).

- *Dynamic Programming*: We implemented the dynamic programming scheme described in [69] for computing the optimal error histogram approximation. We only use the optimal histogram for error comparison, and so techniques for improving the space bounds are irrelevant in this case [59].

- SPACEAPXWAVEHIST *Algorithm*: We also implemented Guha's algorithm described in [59], which currently has the best theoretical approximation bound[5]. (More details about this can be found in Section 3.7.1. For performance reasons, this algorithm is implemented in C++.)

- *Optimal Amnesic Approximation*: We implemented the optimal (offline) amnesic approximation scheme. This algorithm processes values in order of decreasing (source) timestamps. Initially the algorithm starts with an empty bucket. Values are added to the bucket as long as the error in the bucket is less than the tolerable error for the highest timestamp value in the bucket. If not, a new bucket is started, and so on.

In our experiments, we used a variety of datasets from various applications. From these, 5 datasets are shown in Figure 3.16.

[5][60] provides similar bounds as [59], the postprocessing time is nearly the same for both the algorithms and this is what we compare with GM and hence we implement only SpaceApxWaveHist.

- STEAMGEN [77]: The set has 9600 measurements of the drum pressure, taken every 3s , from a model of a steam generator at Abbott Power Plant in Champaign IL.

- RANDOM [77]: This is the classical random walk data used in many papers on time series indexing. This set consists of 65536 values.

- DJIA: This set has the value of the Dow Jones Industrial Average index starting from year 1900. We obtained this dataset from StatLib and it consists of 25737 values.

- HUM: This set consists of measurements from humidity sensors monitoring servers in a large data center. It has 69652 values, representing humidity measurements every 30 seconds.

- TEMP: This set of 93861 values consists of temperature readings taken once every 5 minutes from the Davis weather station in Amherst, Massachusetts.

We begin by evaluating the quality of approximations produced by the GM algorithm and demonstrating the scalability of the scheme in Section 3.7.1. In Section 3.7.2 we compare the performance of AM algorithm with the optimal offline algorithm. And finally, in Section 3.7.3 we demonstrate the quality of approximations produced by the FM algorithm for out of order streams.

3.7.1 Minmerge for the L_2 Histograms

We evaluate the quality and scalability of L_2 histogram approximation by our generic minmerge algorithm GM. (A minmerge type algorithm for the L_∞ histogram was already presented in [17].) Curiously, algorithms that use greedy bucket-merges for the L_2 streaming histograms have been used widely before, *but with no worst-case error bounds* [75, 104]. In fact, there is only one minor difference in the details of our GM and the GRAP-R algorithm [75]: we do the merge *after allocating the new bucket* for the new item, while GRAP-R does the merge first. Empirically, this minor change has little effect on the performance of the algorithm, but it does allow us to prove a worst-case error bound for GM.

The Approximation Quality

We compare the quality of the approximations produced by the GM algorithm with the dynamic programming scheme. Because dynamic programming has quadratic time complexity, it is extremely slow for large data sets, so we ran our test using $n = 4000$ values. The results are shown in Figure 3.17, which plots the L_2 approximation error vs. the number of buckets.

Our results were virtually identical across all 5 data sets, so this figure shows a representative sample for the HUM data set. In the experiment, we vary the buckets from 50 to 200. (Since the slowness of the dynamic programming limited us to $4K$ size data sets, there did not seem to be a point in increasing the number of buckets to larger values. In later experiments on larger data sets, we allow larger values of B.)

The total error predictably decreases with an increasing number of buckets. Recall that Theorem 3.4.3 gives a $(\sqrt{2}, 4)$ approximation guarantee, meaning that the error of the GM histogram using $4B$ buckets is no worse than $\sqrt{2}$ times the error of the optimal B-bucket histogram. Thus, Figure 3.17 shows 3 bars for each value of B: the leftmost bar is the optimal error; the middle bar is the error of the GM using $4B$ buckets; and the rightmost bar is the *error of the GM using just B buckets.* The left part shows the results for PC histograms and the right part shows the results for PL histograms. These results show that GM's performance is much better than the worst-case guarantee: with $4B$ buckets, the error is about a third of the optimal error, and even with the same memory as optimal (B buckets), the error of the GM approximation is only slightly worse (never more than 30%).

In fact, for the rest of the empirical evaluations, even for AM and FM algorithms, we always use only B buckets for our algorithms, and not the constant-factor-times B buckets required by our theorems.

Scalability of GM algorithm

In this experiment we evaluate the scalability (processing time) of the GM algorithm as a function of the size of the stream. We use a large dataset to show the scalability, we call this dataset PRECIP. The archive [122] contains the average monthly and annual

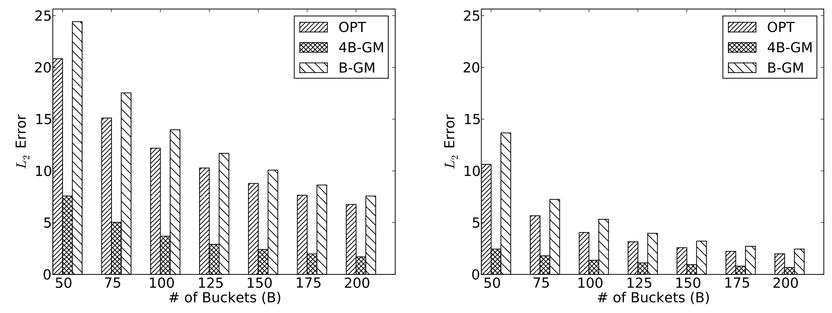


Figure 3.17: The approximation quality as a function of the number of buckets, for the HUM data. The results shown are for PC histograms (left) and PL histograms (right). B-GM refers to the B-bucket approximation, while $4B$-GM refers to the $4B$-bucket approximation (as needed for the theorem).

gage-corrected precipitation recorded by $26,858$ stations worldwide. Station averages of precipitation are interpolated to a 0.5 degree by 0.5 degree of latitude/longitude grid, where the grid nodes are centered on 0.25 degree, resulting in 259200 geographically distinct measurements. The dataset PRECIP here analyzed contains $1M$ values from the previous archive obtained by concatenating the measurements for all stations for several months after having sorted the archive on (latitude, longitude) of the stations.

We vary the size of the dataset from $200K$ to $1M$ in intervals of $200K$ and plot the time taken per data point by the GM algorithm for different number of buckets: $B = 1000, 5000, 9000$. The results are shown in Figure 3.18 for PC approximation. As predicted by theory, the GM algorithm scales linearly with the size of the stream and

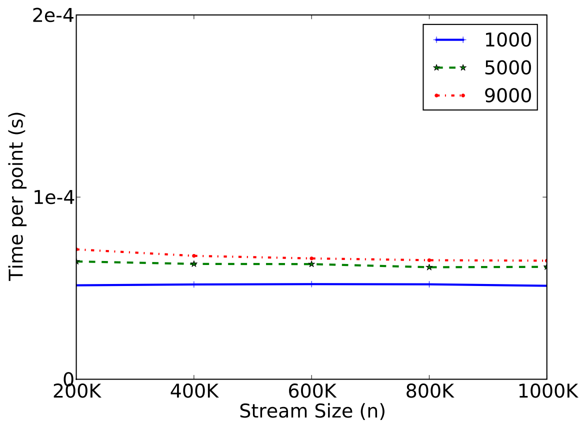


Figure 3.18: Per-item processing time vs. the size of the time stream, for $B = 1000, 5000, 9000$ for the PRECIP dataset.

shows a negligible effect of the number of buckets. The results for PL approximations are the same, just that it takes more processing time per data point (roughly a factor of 3 more).

Comparison with SpaceApxWaveHist

The current best theoretical bound for the L_2 histogram approximation is due to Guha [59]: his SpaceApxWaveHist algorithm achieves $(1+\varepsilon, 1)$ approximation in $O(n + B^3\varepsilon^{-2}\log^2 n)$ time using $O(B\varepsilon^{-2}\log n \log \varepsilon^{-1})$ space. Our GM algorithm only guarantees a $(\sqrt{2}, 4)$ approximation, but has the advantage of utter simplicity, as well as generalizability to models such as amnesic and out-of-order streams. A second, and perhaps, more significant advantage of our scheme may be that it maintains a *query-ready* indexable structure at all times. In other words, our time series approximation is directly represented as a histogram that can be queried at any time, with no additional overhead. This is in contrast to schemes such as Guha's [59] whose synopses consist of wavelet coefficients that must be processed to build a B bucket histogram. The overhead of this post-processing requirement for the guarantee can be significant—for instance, the technique in [59] requires $O(B^3 \log^2 n)$ time to build the B bucket histogram before a query can be answered. Thus, the two schemes, ours and Guha's, have different advantages—the latter has the best provable worst-case error guarantee while the former maintains an online query-ready histogram—so it is worth comparing the two in a simulation.

Since a public implementation of SPACEAPXWAVEHIST was no available, we implemented it ourselves. After some optimizations and parameter tuning, the observed performance seems to match with what is reported in [59], so we feel that our comparisons should be fair. Figures 3.19 shows the main results of our experiments (for HUM dataset): the error in the approximation of the GM algorithm (using the same amount of memory) is close to that achieved by SPACEAPXWAVEHIST (the left figure), however the running time of that algorithm is significantly worse than GM (as predicted by theory), as shown in the right figure. Results for other datasets are virtually the same. The SPACEAPXWAVEHIST scheme uses memory in two places, wavelet synopsis and partial dynamic programming table. *Rather than optimize the memory distribution* *across these two components, we allocated to each part as much memory as our GM scheme requires, thereby potentially giving* SPACEAPXWAVEHIST *twice the memory.*

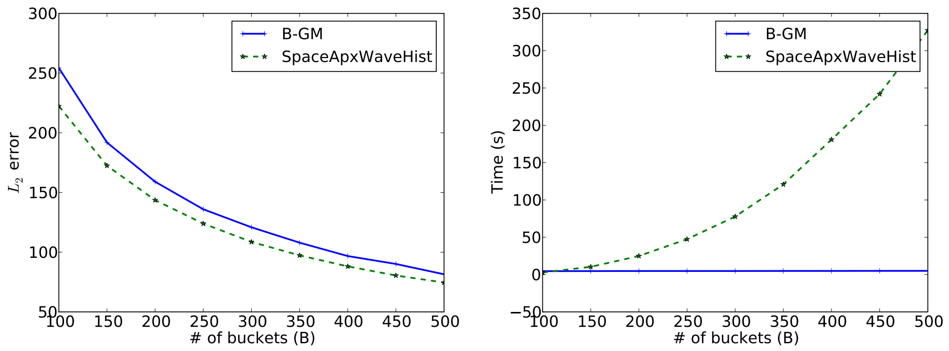


Figure 3.19: GM vs. SPACEAPXWAVEHIST. The approximation error (left) of B-bucket GM is comparable to the B-bucket SPACEAPXWAVEHIST, with the latter given twice the working space. The running time of GM is significantly faster, due to the $O(B^3)$ term in SPACEAPXWAVEHIST.

3.7.2 The Amnesic Time Series Approximation

In order to set up a natural aging function, we divided the data into 4 groups, with approximation error set to 1%, 2%, 3% and 4%, respectively, of the maximum data range, where the data range is defined as the difference between the maximum and the minimum value. The spread of the groups is exponential. In other words oldest $n/2$ are allowed error 4%, the next $n/4$ values are allowed error 3% and so on. Thus, we have a monotone piecewise constant amnesic function with 4 steps. We also used a scaling function to adjust the values of the allowable errors, while keeping their ratio the same.

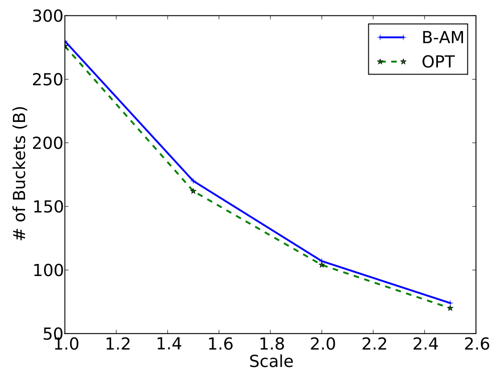
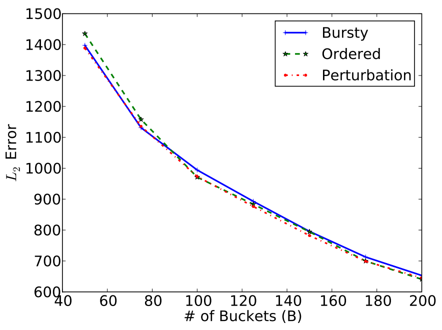


Figure 3.20: The number of buckets required by the AM (B-AM) algorithm as compared to optimal for different scaling factors for the DJIA dataset.

In other words, by using a scaling factor of 2, we get the approximation errors for the 4 age groups to be 2.0%, 4.0%, 6.0% and 8.0%.

The approximation quality for the amnesic histogram is given by the number of buckets used by the AM algorithm, so we compare this to the optimal amnesic approximation. We show the results of the streaming amnesic approximation for one representative data set (DJIA), which contains $25K$ data points. Our results for PC approximations are shown in Figure 3.20. One can see that the number of buckets used by the AM algorithm closely track the optimal number of buckets and typically within a factor 1.2 of the optimal number of buckets. The results for the other datasets for both PC and PL approximations are similar; the factors go up to 1.3.



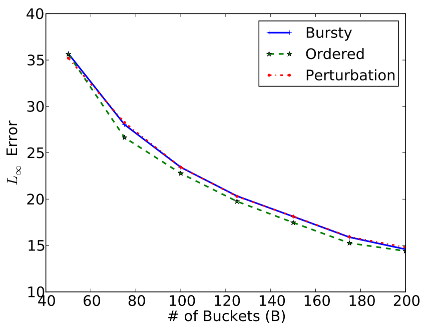


Figure 3.21: The quality of approximation produced for the Bursty, Perturbation and the Ordered stream models for L_2 (left) and L_∞ (right) error metrics. The FM algorithm adapts very well for out of order arrivals.

3.7.3 Performance of the Fragment-Merge (FM)

Our last set of experiments analyze the performance of the FRAGMENT-MERGE algorithm for approximating out-of-order streams. We first describe our data model for generating a realistic out-of-order stream. There are two aspects to this: how frequently an item appears out of order, and how far from its true position it reappears. Let us first consider the second question. Using the Internet as a guide, we envision two different models for packet delay: a *bursty* model, which may account for buffer overflow at a node causing many data values to be lost and retransmitted, and a *perturbation* model, which may account for the multi-path routing phenomenon in networking, causing packets to arrive slightly out of order. For the first question, we use a correlated random model to decide which data value to delay.

In more details, we construct an out-of-order stream from an ordered stream statistically in the following manner. The ith item in the stream is reordered (delayed) with probability p_i, given as follows:

$$p_i = \frac{(1-\chi)\cdot\rho + \chi \cdot I_{i-1}}{I_{i-1}+\rho}, \tag{3.26}$$

where ρ is the independent probability for an item to be reordered, χ is the correlation probability that accounts for the previous item being reordered, and I_{i-1} is the indicator variable which is either 1 or 0 depending on whether the $(i-1)$th value was reordered or not. In order to simulate this, for the ith value we flip a coin biased with probability p_i. Once an item is taken out, it is reinserted later in the stream with a delay uniformly distributed between 0 and L (lag). Thus, our reordering strategy is completely described by the parameters χ, ρ and L. The bursty model is simulated with small value of ρ, and large values for χ and L. The perturbation model is simulated with large value of ρ, and small values of χ and L.

We evaluate the error performance of the FM algorithm by comparing it with the approximation obtained by the GM algorithm for the ordered version of the stream. Figure 3.21 shows the results of this experiment for the DJIA data, where the out-of-order streams are generated both with the bursty and the perturbation models, and the plot shows the error of the approximation vs. the bucket parameter B.

The figure on the left shows the L_2 approximation, while the one on the right shows the L_∞ approximation. In both cases, it is easy to see that the error of the FRAGMENT-MERGE approximation is very close to the error of the GM approximation for the ordered version of the stream. In some cases, the FM error approximation is actually smaller than the GM approximation, but that is explained by the fact the FM approximation uses a maximum of $B + k$ buckets, where k is the number of gaps in the stream, while the ordered stream uses only B buckets. In this set of experiments, the maximum number of gaps for the bursty model is 84, and for the perturbation model it is 23.

We also ran scalability for experiments for both AM and FM algorithms. The results for these are very similar to the GM algorithm, though individual per item processing times are slightly slower for both AM and FM as compared to GM.

3.8 Conclusions and Future Work

We presented a generic framework for online approximation of time-series data that yields a unified set of algorithms for several popular models: data streams, amnesic approximation, and out-of-order stream approximation. Our framework essentially develops a popular greedy method of bucket-merging into a more generic form, for which we can prove space-quality approximation bounds. When specialized to piecewise linear bucket approximations and commonly used error metrics, such as L_2 or L_∞, our framework leads to provable error bounds where none were known before, offers new results, or yields simpler and unified algorithms. The conceptual simplicity of our scheme translates into highly practical implementations, as borne out in our simulation studies: the algorithms produce near-optimal approximations, require very small memory footprints, and run extremely fast.

Acknowledgments: We thank Sudipto Guha for helping us understand the details of his algorithm during our implementation, Eamonn Keogh for giving us access to time series data, and Themis Palpanas for clarifying one of the proofs in [104].

Chapter 4

Measuring Micro Bursts*

4.1 Introduction

How can a manager of a computing resource detect bursts in resource usage that cause performance degradation without keeping a complete log? The problem is one of extracting a needle from a haystack; the problem gets worse as the needle gets smaller (as finer-grain bursts cause drops in performance) and the haystack gets bigger (as the consumption rate increases). While this chapter addresses this general problem, we focus on detecting bursts of bandwidth usage, a problem that has received much attention [7, 107, 120] in modern data centers.

The simplest definition of a *microburst* is the transmission of more than B bytes of data in a time interval t on a single link, where t is in the order of 100's of microseconds.

Parts of this chapter appeared in the following publication: [119]

For input and output links of the same speed, bursts must occur on several links at the same time to overrun a switch buffer, as in the Incast problem [25, 107]. Thus, a more useful definition is the sending of more than B bytes in time t over *several* input links that are destined to the same output switch port. This general definition requires detecting bursts that are correlated in time across several input links.

Microbursts cause problems because data center link speeds have moved to 10 Gbps while commodity switch buffers use comparatively small amounts of memory (Mbytes). Since high-speed buffer memory contributes significantly to switch cost, commodity switches continue to provision shallow buffers, which are vulnerable to overflowing and dropping packets. Dropped packets lead to TCP retransmissions which can cause millisecond latency increases that are unacceptable in data centers.

Administrators of financial trading data centers, for instance, are concerned with the microburst phenomena [4] because even a latency advantage of 1 millisecond over the competition may translate to profit differentials of \$100 million per year [92]. While financial networks are a niche application, high-performance computing is not. Expensive, special-purpose switching equipment used in high-performance computing (e.g. Infiniband and FiberChannel) is being replaced by commodity Ethernet switches. In order for Ethernet networks to compete, managers need to identify and address the fine-grained variations in latencies and losses caused by microbursts. At the core of this problem is the need to identify the bandwidth patterns and corresponding applications causing these latency spikes so that corrective action can be taken.

Efficient and effective monitoring becomes increasingly difficult as faster links allow very short-lived phenomenon to overwhelm buffers. For a commodity 24-port 10 Gbps switch with 4 MB of shared buffer, the buffer can be filled (assuming no draining) in 3.2 msec by a single link. However, given that bursts are often correlated across several links and buffers must be shared, the time scales at which interesting bursts occur can be ten times smaller, down to 100's of μs. Instead of 3.2 msec, the buffer can overflow in 320 μs if 10 input ports each receive 0.4 MB in parallel. Assume that the strategy to identify correlated bursts across links is to first identify bursts on single links and then to observe that they are correlated in time. The single link problem is then to efficiently identify periods of length t where more than B bytes of data occur. Currently, t can vary from hundreds of microseconds to milliseconds and B can vary from 100's of Kbytes to a few Mbytes. Solving this problem *efficiently* using minimal CPU processing and logging bandwidth is one of the main concerns of this chapter.

Although identifying "bursts" on a single link for a range of possible time scales and byte thresholds is challenging, the ideal solution should do two more things. First, the solution should efficiently extract flows responsible for such bursts so that a manger can reschedule or rate limit them. Second, the tool should allow a manager to detect bursts correlated in time across links. While the first problem can be solved using heavy-hitter techniques [96], we briefly describe some new ideas for this problem in our context. The second problem can be solved by archiving bandwidth measurement records indexed by link and time to a relational database which can then be queried for persistent patterns. This requires an efficient summarization technique so that the archival storage required by the database is manageable.

Generalizing to Bandwidth Queries: Beyond identifying microbursts, we believe that modeling traffic at fine time scales is of fundamental importance. Such modeling could form the basis for provisioning NIC and switch buffers, and for load balancing and traffic engineering at fine time scales. While powerful, coarse-grain tools are available, the ability to flexibly and efficiently measure traffic at different, and especially fine-grain, resolutions is limited or non-existent.

For instance, we are unable to answer basic questions such as: what is the distribution of traffic bursts? At which time-scale did the traffic exhibit burstiness? With the identification of long-range dependence (LRD) in network traffic [42], the research community has undergone a mental shift from Poisson and memory-less processes to LRD and bursty processes. Despite its widespread use, however, LRD analysis is hindered by our inability to estimate its parameters unambiguously. Thus, our larger goal is to use fine-grain measurement techniques for fine-grain traffic modeling.

While it is not difficult to choose a small number of preset resolutions and perform measurements for those, the more difficult and useful problem is to support traffic measurements for *all time scales*. Not only do measurement resolutions of interest vary with time (as in burst detection), but in many applications they only become critical *after the fact*, that is, after the measurements have already been performed. This chapter describes an end-host bandwidth measurement tool that succinctly summarizes bandwidth information and yet answers general queries at arbitrary resolutions without maintaining state for all time scales.

Some representative queries (among many) that we wish such a tool to support are the following:

1. What is the *maximum* bandwidth used at time scale t?
2. What is the *standard deviation* and 95th percentile of the bandwidth at time scale t?
3. What is the *coarsest* time scale at which bandwidth exceeds threshold L?

In these queries, the query parameters t or L are chosen *a posteriori* — after all the measurements have been performed, and thus require supporting all possible resolutions and bandwidths.

Existing techniques: All the above queries above can be easily answered by keeping the entire packet trace. However, our data structures take an order of magnitude less storage than a packet trace (even a sampled packet trace) and yet can answer flexible queries with good accuracy. Note that standard summarization techniques (including simple ones like SNMP packet counters [1]) and more complex ones (e.g., heavy-hitter determination [89]) are very efficient in storage but must be targeted towards a particular purpose and at a fixed time scale. Hence, they cannot answer flexible queries for arbitrary time scales.

Note that sampling 1 in N packets, as in Cisco NetFlow [2], does not provide a good solution for bandwidth measurement queries. Consider a 10 Gbps link with an average packet size of 1000 bytes. This link can produce 10 million packets per second. Suppose the scheme does 1 in 1000 packet sampling. It can still produce 10,000 samples per second with say 6 bytes per sample for time-stamp and packet size. To identify bursts of 1000 packets of 1500 bytes each (1.5 MB), any algorithm would look for intervals containing 1 packet and scale up by the down sampling factor of 1000. The major problem is that this causes false positives. If the trace is well-behaved and has no bursts in any specified period (say 10 msec), the scaling scheme will still falsely identify 1 in 1000 packets as being part of bursts because of the large scaling factor needed for data reduction. Packet sampling, fundamentally, takes no account of the passage of time.

From an information-theoretic sense, packet traces, are inefficient representations for bandwidth queries. Viewing a trace as a time series of point masses (bytes in each packet), it is more memory-efficient to represent the trace as a series of *time intervals* with bytes sent per interval. But this introduces the new problem of choosing the intervals for representation so that bandwidth queries on any interval (chosen after the trace has been summarized) can be answered with minimal error.

Our first scheme builds on the simple idea that for any fixed sampling interval, say 100 microseconds, one can easily compute traffic statistics such as max or *Standard Deviation* by a few counters each. By exponentially increasing the sampling interval, we can span an aggregation period of length T, and still compute statistics at all time scales from microseconds to milliseconds, using only $O(\log T)$ counters. We call this approach Exponential Bucketing (EXPB). The challenge in EXPB is to avoid updating all $\log T$ counters on each packet arrival and to prove error bounds.

Our second idea, dubbed Dynamic Bucket Merge (DBM), constructs an approximate streaming histogram of the traffic so that bursts stand out as peaks in this histogram. Specifically, we adaptively partition the traffic into k intervals/buckets, in such a way that the periods of heavy traffic map to more refined buckets than those of low traffic. The time-scales of these buckets provide a "visual history" of the burstiness of the traffic—the narrower the bucket in time, the burstier the traffic. In particular, DBM is well-suited for identifying not only whether a burst occurred, but *how many bursts*, and *when*.

System Deployment: Exponential Bucketing and Dynamic Bucket Merge have low computational and storage overheads, and can be implemented at multi-gigabit speeds in software or hardware. As shown in Figure 4.22, we envision a deployment scenario where both end hosts and network devices record fine-grain bandwidth summaries to a centralized log server. We argue that even archiving to a single commodity hard disk, administrators could pinpoint, to the second, the time at which correlated bursts occurred on given links, even up to a year after the fact.

This data can be indexed using a relational database, allowing administrators to query bandwidth statistics across links and time. For example, administrators could issue queries to “Find all bursts that occurred between 10 and 11 AM on all links in Set S”. Set S could be the set of input links to a single switch (which can reveal Incast problems) or the path between two machines. Bandwidth for particular links can then be visualized to further delineate burst behavior. The foundation for answering such queries is the ability to efficiently and succinctly summarize the bandwidth usage of a trace in real-time, the topic of this chapter.

We break down the remainder of our work as follows. We begin with a discussion of related algorithms and systems in §4.2. §4.3 illustrates the Dynamic Bucket Merge and Exponential Bucketing algorithms, both formally and with examples. We follow with our evaluations in §4.4, describe the implications for a system like Figure 4.22 in §4.5, and conclude in §4.6.

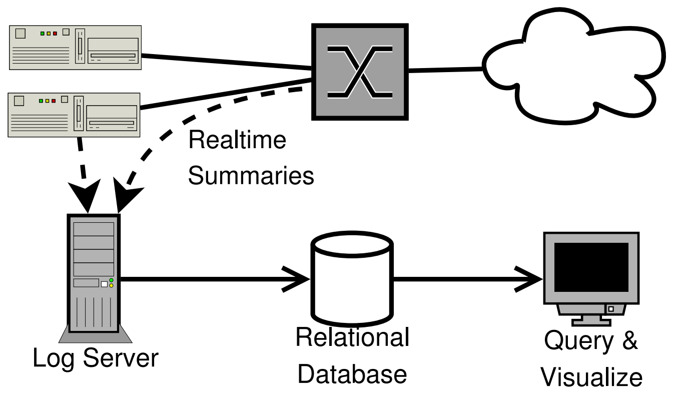


Figure 4.22: Example Deployment. End hosts and network devices implementing `EXPB` and `DBM` push output data over the network to a log server. Data at the server can be monitored and visualized by administrators then collapsed and archived to long-term, persistent storage.

4.2 Related Work

Tcpdump [5] is a mature tool that captures a full log of packets at the endhost, which can be used for a wide variety of statistics, including bandwidth at any time scale. While flexible, tcpdump consumes too much memory for continuous monitoring at high speeds across every link and for periods of days. Netflow [2] can capture packet headers in routers but has the same issues. While sampled Netflow reduces storage, configurations with substantial memory savings cannot detect bursts without resulting in serious false positives. SNMP counters [1], on the other hand, provide packet and byte counts but can only return values at coarse and fixed time scales.

There are a wide variety of summarization data structures for traffic streams, many of which are surveyed in [96]. None of these can directly be adapted to solve the bandwidth problem at all time scales, though solutions to quantile detection do solve some aspects of the problem [96]. For example, classical heavy-hitters [89] measures the heaviest traffic flows during an interval. By contrast, we wish to measure "heavy-hitting sub-intervals across time", so to speak. However, heavy-hitter solutions are complementary in order to identify flows that cause the problem. The LDA data structure [81] is for a related problem – that of measuring average *latency*. LDA is useful for directly measuring latency violations. Our algorithms are complementary in that they help analyze the bandwidth patterns that *cause* latency violations.

`DBM` is inspired by the adaptive space partitioning scheme of [67], but is greatly simplified, and also considerably more efficient, due to the time-series nature of packet arrivals.

4.3 Algorithms

Suppose we wish to perform bandwidth measurements during a time window $[0, T]$, assuming, without loss of generality, that the window begins at time zero. We assume that during this period N packets are sent, with p_i being the byte size of the ith packet and t_i being the time at which this packet is logged by our monitoring system, for

$i = 1, 2, \ldots, N$. These packets are received and processed by our system as a *stream*, meaning that the ith packet arrives before the jth packet, for any $i < j$.

The *bandwidth* is a rate, and so converting our observed sequence of N packets into a quantifiable bandwidth usage requires a *time scale*. Since we wish to measure bandwidth at different time scales, let us first make precise what we mean by this. Given a time scale (or *granularity*) Δ, where $0 < \Delta < T$, we divide the measurement window $[0, T]$ into sub-intervals of length Δ, and aggregate all those packets that are sent within the same interval. In this way, we arrive at a sequence $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals.[6]

Therefore, every choice of Δ *leads to a corresponding sequence* S_Δ*, which we interpret as the bandwidth use at the temporal granularity* Δ. All statistical measurements of bandwidth usage at time scale Δ correspond to statistics over this sequence S_Δ. For instance, we can quantify the statistical behavior of the bandwidth at time scale Δ by measuring the *mean, standard deviation, maximum, median, quantiles*, etc. of S_Δ.

In the following, we describe two schemes that can estimate these statistics for *every a posteriori* choice of the time scale Δ. That is, after the packet stream has been

[6]To deal with the boundary problem properly, we assume that each sub-interval includes its right boundary, but not the left boundary. If we assume assume that no packet arrives at time 0, we can form a proper non-overlapping partition this way.

processed by our algorithms, the users can *query* for an arbitrary granularity Δ and receive provable quality approximations of the statistics for the sequence S_Δ.

Our first scheme, `DBM`, is time scale agnostic, and essentially maintains a streaming histogram of the values $s_1, s_2, \ldots, s_k$, by adaptively partitioning the period $[0, T]$. Our second scheme `EXPB` explicitly computes statistics for *a priori* settings of Δ, and then uses them to approximate the statistics for the queried value of Δ.

Since the two schemes are quite orthogonal to each other, it is also possible to use them both in conjunction. We give worst-case error guarantees for both of the schemes. Both schemes are able to compute the mean with perfect accuracy and estimate the other statistics, such as the maximum or standard deviation, with a bounded error. The approximation error for the `DBM` scheme is expressed as an additive error, while the `EXPB` scheme offers a multiplicative relative error. In particular, for the `DBM` scheme, the estimation of the maximum or standard deviation is bounded by an error term of the form $O(\varepsilon B)$, where $0 < \varepsilon < 1$ is a user-specified parameter dependent on the memory used by the data structure, and $B = \sum_{i=1}^{N} p_i$ is the total packet mass over the measurement window. In the following, we describe and analyze the `DBM` scheme, followed by a description and analysis of the `EXPB` scheme.

4.3.1 Dynamic Bucket Merge

`DBM` maintains a partition of the measurement window $[0, T]$ into what we call *buckets*. In particular, a m-bucket partition $\{b_1, b_2, \ldots, b_m\}$, is specified by a sequence of time instants $t(b_i)$, with $0 < t(b_i) \leq T$, with the interpretation that the bucket b_i spans the interval $(t(b_{i-1}),\ t(b_i)]$. That is, $t(b_i)$ marks the time when the ith bucket ends, with the convention that $t(b_0) = 0$, and $t(b_m) = T$. The number of buckets m is controlled by the memory available to the algorithm and, as we will show, the approximation quality of the algorithm improves linearly with m. In the following, our description and analysis of the scheme is expressed in terms of m. Each bucket maintains $O(1)$ information, typically the statistics we are interested in maintaining, such as the total number of bytes sent during the bucket. In particular, in the following, we use the notation $p(b)$ to denote the total number of data bytes sent during the interval spanned by a bucket b.

The algorithm processes the packet stream p_1, p_2, …, p_N in arrival time order, always maintaining a partition of $[0, T]$ into at most m buckets. (In fact, after the first m packets have been processed, the number of buckets will be exactly m, and the most recently processed packet lies in the last bucket, namely, b_m.) The basic algorithm is quite straightforward. When the next packet p_j is processed, we place it into a new bucket b_{m+1}, with time interval (t_{j-1}, T)—recall that t_{j-1} is the time stamp associated with the preceding packet p_{j-1}. We also note that the right boundary of the predecessor bucket b_m now becomes t_{j-1} due to the addition of the bucket b_{m+1}. Since we now have $m + 1$ buckets, we merge two *adjacent* buckets to reduce the bucket count down to m. Several different criteria can be used for deciding which buckets to merge, and we consider some alternatives later, but in our basic scheme we merge the buckets based on their *packet mass*. That is, we merge two adjacent buckets whose sum of the packet mass is the smallest over all such adjacent pairs. A pseudo-code description of DBM is presented in Algorithm 3.

Algorithm 3: DBM

```
1 foreach $p_j \in S$ do
2     Allocate a new bucket $b_i$ and set $p(b_i) = p_j$
3     if $i == m + 1$ then
5         Merge the two adjacent $b_w$, $b_{w+1}$ for which $p(b_w) + p(b_{w+1})$ is minimum;
6     end
7 end
```

DBM Example

To clarify the operation of DBM we give the following example, illustrated in Figure 4.23.

Suppose that we run DBM with 4 buckets ($m = 4$), each of which stores a *count* of the number of buckets that have been merged into it, the *sum* of all bytes belonging to it, and the *max* number of bytes of any bucket merged into it. Now suppose that 4 packets

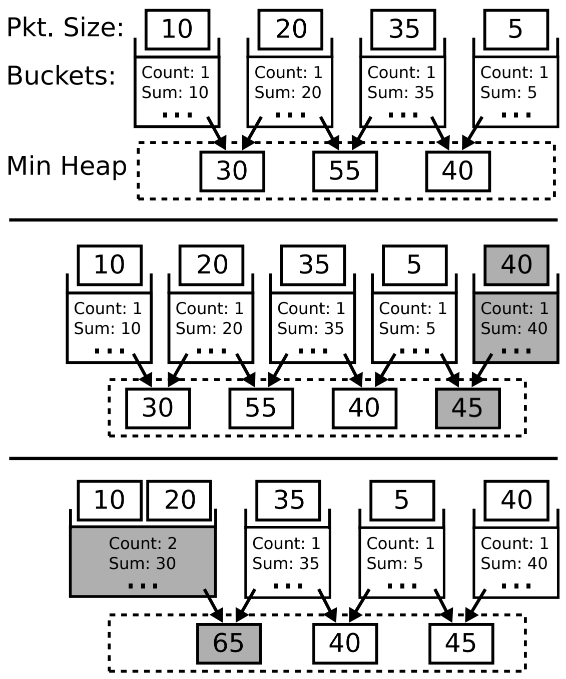


Figure 4.23: Dynamic Bucket Merge with 4 buckets. Initially each bucket contains a single packet and the min heap holds the sums of adjacent bucket pairs. When a new packet (value = 40) arrives, a 5th bucket is allocated and a new entry added to the heap. In the merge step, the smallest value (30) is popped from the heap and the two associated buckets are merged. Last, we update the heap values that depended on either of the merged buckets.

have arrived with masses 10, 20, 35, and 5, respectively. The state of `DBM` at this point is shown at the top of Figure 4.23. Note that Algorithm 3 required that we merge the buckets with the minimum combined sum. Hence, we maintain a min heap which stores the sums of adjacent buckets.

When a fifth packet with a mass of 40 arrives, `DBM` allocates a new bucket for it and updates the heap with the sum of the new bucket and its neighbor.

In the final step, the minimum sum is pulled from the heap and the buckets contributing to that sum are merged. In this example, the bucket containing mass 10 and 20 are merged into a single bucket with a new mass of 30 and a max bucket value of 20. Note that we also update the values in the heap which included the mass of either of the merge buckets.

DBM Analysis

The key property of `DBM` is that it can estimate the total number of bytes sent during *any* time interval. In particular, let $[t, t']$ be an arbitrary interval, where $0 \leq t, t' \leq T$, and let $p(t, t')$ be the total number of bytes sent during it, meaning $p(t, t') = \sum_{i=1}^{N} \{p_i \mid t \leq t_i \leq t'\}$. Then we have the following result.

Lemma 4.3.1. The data structure `DBM` estimates $p(t, t')$ within an additive error $O(B/m)$, for any interval $[t, t']$, where m is the number of buckets used by `DBM` and $B = \sum_{i=1}^{N} p_i$ is the total packet mass over the measurement window $[0, T]$.

Proof: We first note that in `DBM` each bucket's packet mass is at most $2B/(m-1)$, unless the bucket contains a single packet whose mass is strictly larger than $2B/(m-1)$. In particular, we argue that whenever two buckets need to be merged, there always exists an adjacent pair with total packet mass less than $2B/(m-1)$. Suppose not. Then, summing the sizes of all $(m-1)$ pairs of adjacent buckets must produce a total mass strictly larger than $2(m-1)B/(m-1) = 2B$, which is impossible since in this sum each bucket is counted at most twice, so the total mass must be less than $2B$.

With this fact established, the rest of the lemma follows easily. In order to estimate $p(t,t')$, we simply add up the buckets whose time spans intersect the interval $[t,t']$. Any bucket whose interval lies entirely inside $[t,t']$ is accurately counted, and so the only error of estimation comes from the two buckets whose intervals only partially intersect $[t,t']$—these are the buckets containing the endpoints t and t'. If these buckets have mass less than $2B/(m-1)$ each, then the total error in estimation is less than $4B/m$, which is $O(\frac{B}{m})$. If, on the other hand, either of the end buckets contains a single packet with large mass, then that packet is correctly included or excluded from the estimation, depending on its time stamp, and so there is no estimation error. This completes the proof. $\square$

Theorem 4.3.2. With `DBM` we can estimate the maximum or the standard deviation of S_Δ within an additive error εB, using memory $O(1/\varepsilon)$.

Proof: The proof for the maximum follows easily from the preceding lemma. We simply query `DBM` for time windows of length Δ, namely, $(i\Delta, (i+1)\Delta]$, for $i = 0, 1, \ldots, \lceil T/\Delta \rceil$, and output the maximum packet mass estimated in any of those intervals. In order to achieve the target error bound, we use $m = \frac{4}{\varepsilon} + 1$ buckets.

We now analyze the approximation of the standard deviation. Recall that the sequence under consideration is $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, for some time scale Δ, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals. Let $Var(S_\Delta)$, $E(S_\Delta)$, and $E(S_\Delta^2)$, respectively, denote the variance, mean, and mean of the squares for S_Δ. Then, by definition, we have

$$Var(S_\Delta) \;=\; E(S_\Delta^2) - E(S_\Delta)^2 = \frac{\sum_{i=1}^{k} {s_i}^2}{k} - E(S_\Delta)^2$$

Since `DBM` estimates each s_i within an additive error of εB, our estimated variance respects the following bound:

$$\leq \frac{\sum (s_i + \varepsilon B)^2}{k} - E(S_\Delta)^2$$

However, we can compute $E(S_\Delta)^2$ exactly, because it is just the square of the mean. In order to derive a bound on the error of the variance, we assume that $k > m$, that is, the size of the sequence S_Δ is at least as large as the number of buckets in `DBM`. (Naturally, statistical measurements are meaningless when the sample size becomes too small.) With this assumption, we have $2/k < 2/m$, and since $\varepsilon = 4/(m-1)$, we get that $2\frac{\sum s_i}{k} \leq \varepsilon B$, which, considering $k \geq 1$, yields the following upper bound for the estimated variance:

$$\leq \frac{\sum s_i^2}{k} - E(S_\Delta)^2 + \frac{k+1}{k}\varepsilon^2 B^2 \;\leq\; Var(S_\Delta) + 2\varepsilon^2 B^2$$

which implies the claim. □

Similarly, we can show the following result for approximating quantiles of the sequence S_Δ.

Theorem 4.3.3. With DBM we can estimate any quantile of S_Δ within an additive error εB, using memory $O(1/\varepsilon)$.

Proof: Let $s_1, s_2, \ldots, s_k$ be the sequence of data in the intervals $(i\Delta, (i+1)\Delta]$, for $i = 1, 2, \ldots, k = \lceil T/\Delta \rceil$, sorted in increasing order, and let $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ be the sorted estimated sequence for the same intervals. We now compute the desired quantile, for instance the 95th percentile, in this sequence. Supposing the index of the quantile is q, we return $\hat{s}_q$. We argue that the error of this approximation is $O(\varepsilon B)$. We do this by estimating bounds on the s_i values that are erroneously (due to approximation) misclassified, meaning reported below or equal the quantile when they are actually larger or vice versa. If no s_i have been misclassified then $\hat{s}_q$ and s_q correspond to the same sample, and by Lemma 4.3.1 the estimated value $\hat{s}_q - s_q \leq \varepsilon B$, hence the claim follows. On the other hand, if a misclassification occurred, then the sample s_q is reported at an index different than q in the estimated sequence. Assume without loss of generality that the sample s_q has been reported as $\hat{s}_u$ where $u > q$. Then, by the pigeonhole principle, there is at least a sample s_h ($h > q$) that is reported as $\hat{s}_d$, $d \leq q$. By Lemma 4.3.1, $\hat{s}_d - s_h \leq \varepsilon B$. Since s_q and s_h switched ranks in the estimated sequence

$\hat{s}$, by Lemma 4.3.1 it holds that $s_h - s_q \leq \varepsilon B$ and $\hat{s}_u - \hat{s}_d \leq \varepsilon B$. By assumption $u > q \geq d$, then it follows that $\hat{s}_u \geq \hat{s}_q \geq \hat{s}_d$ in the sorted sequence $\hat{s}$, which implies that $\hat{s}_q - \hat{s}_d \leq \varepsilon B$. The chain of inequalities implies that $\hat{s}_q - s_q \leq 3\varepsilon B$, which completes the proof. □

Algorithm 3 can be implemented at the worst-case cost of $O(\log m)$ per packet, with the heap operation being the dominant step. The memory usage of DBM is $\Theta(m)$ as each bucket maintains $O(1)$ information.

Extensions to DBM for better burst detection

Generic DBM is a useful oracle for estimating bandwidth in any interval (chosen after the fact) with bounded additive error. However, one can tune the merge rule of DBM if the goal is to pick out the bursts only. Intuitively, if we have an aggregation period with k bursts for small k (say 10) spread out in a large interval, then ideally we would like to compress the large trace to k high-density intervals. Of course, we would like to also represent the comparatively low traffic adjacent intervals as well, so an ideal algorithm would partition the trace into $2k + 1$ intervals where the bursts and ideal periods are clearly and even visually identified. We refer to the generic scheme discussed earlier that uses *merge-by-mass* as DBM-mm, and describe two new variants as follows.

- merge-by-variance (DBM-mv): merges the two adjacent buckets that have the minimum aggregated packet mass variance

- merge-by-range (`DBM-mr`): merges the two adjacent buckets that have the minimum aggregated packet mass range (defined as the difference between maximum and minimum packet masses within the bucket)

These merge variants can also be implemented in logarithmic time, and require storing $O(1)$ additional information for each bucket (in addition to $p(b_i)$).

One minor detail is that `DBM-mv` and `DBM-mr` are sensitive to null packet mass in an interval while `DBM-mm` is not. For these reasons, we make the `DBM-mr` and `DBM-mv` algorithms work on the sequence defined by S_Δ, where Δ is the minimum time scale at which bandwidth measurements can be queried. Then `DBM-mr` and `DBM-mv` represents S_Δ as a histogram on m buckets, where each bucket has a discrete value for the signal. The goal of a good approximation is to minimize its predicted value versus the true under some error metric. We consider both the L_2 norm and the L_∞ norm for the approximation error.

$$E_2 = (\sum_{i=1}^{n} |s_i - \hat{s}_i|^2)^{\frac{1}{2}} \tag{4.27}$$

where $\hat{s}_i$ is the approximation for value s_i.

$$E_\infty = \max_{i=1}^{n} |s_i - \hat{s}_i| \tag{4.28}$$

We compare the performance of `DBM-mr` and `DBM-mv` algorithms with the optimal offline algorithms, that is, a bucketing scheme that would find the optimal partition of S_Δ

to minimize the E_2 or the E_∞ metric. Then, the analysis of [17, 56] can be adapted to yield the following results that formally state our intuitive goal of picking out m bursts with $2m + 1$ pieces of memory.

Theorem 4.3.4. The L_∞ approximation error of the m-bucket `DBM-mr` is never worse than the corresponding error of an optimal $m/2$-bucket partition.

Theorem 4.3.5. The L_2 approximation error of the m-bucket `DBM-mv` is at most $\sqrt{2}$ times the corresponding error of an optimal $m/4$-bucket partition.

4.3.2 Exponential Bucketing

Our second scheme, which we call Exponential Bucketing (`EXPB`), explicitly computes statistics for *a priori* settings of $\Delta_1, \ldots, \Delta_m$, and then uses them to approximate the statistics for the queried value for *any* Δ, for $\Delta_1 \leq \Delta \leq \Delta_m$. We assume that the time scales grow in powers of two, meaning that $\Delta_i = 2^{i-1}\Delta_1$. Therefore, we can assume that the scheme processes data at the time scale Δ_1, namely, the sequence $S_{\Delta_1} = (s_1, s_2, \ldots, s_k)$.

Conceptually, `EXPB` maintains bandwidth statistics for all m time scales $\Delta_1, \ldots, \Delta_m$. A naïve implementation would require updating $O(m)$ counters per (aggregated) packet s_i. However, by carefully orchestrating the accumulator update when a new s_i is available it is possible to avoid spending m updates per measurement as shown in Algorithm 4.

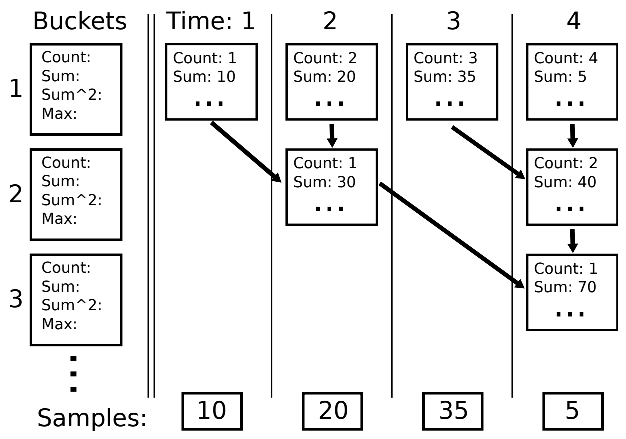


Figure 4.24: Exponential Bucketing Example. Each of the m buckets collects statistics at 2^{i-1} times the finest time scale. At the end of each time scale, Δ_i, buckets 1 to i must be updated. Before storing the new sum in a bucket j, we first add the old sum into bucket $j + 1$, if it exists.

The intuition is as follows. Suppose one is maintaining statistics at 100 μs and 200 μs intervals. When a packet arrives, we update the 100 μs counter but not the 200 μs counter. Instead, the 200 μs counter is updated only when the 100 μs counter is zeroed. In other words, only the lowest granularity counter is updated on every packet, and coarser granularity counters are only updated when all the finer granularity counters are zeroed.

EXPB Example

To better understand the `EXPB` algorithm we now present the example illustrated in Figure 4.24.

Algorithm 4: `EXPB`

```
sum=$< 0, \dots, 0 >$ ($m$ times) ;
**foreach** $s_i$ **do**
    sum[0]=$s_i$;
    j=0;
    **repeat**
        updatestat(j,sum[j]);
        **if** $j < m$ **then**
            sum[j+1]+=sum[j];
        **end**
        sum[j]=0;
        j++;
    **until** $i$ *mod* $2^j \neq 0$ *or* $j \geq m$;
**end**
```

In this example, we maintain 3 buckets ($m = 3$) each of which stores statistics at time scales of 1, 2 and 4 time units. Each bucket stores the *count* of the intervals elapsed, the *sum* of the bytes seen in the current interval, and fields to compute max and standard deviation. We label the time units along the top and the number of bytes accumulated during each interval along the bottom.

In the first time interval 10 bytes are recorded in the first bucket and 10 is pushed to the sum of the second bucket. We repeat this operation when 20 is recorded in the second interval. Since 2 time units have elapsed, we also update the statistics for the Δ_2 time scale, and add bucket two's sum to bucket 3. In the third interval we update bucket 1 as before. Finally, at time 4 we update bucket 2 with the current sum from bucket 1, update bucket two's statistics, and push bucket two's sum to bucket 3. Finally, we update the statistics for Δ_3 with bucket three's sum.

EXPB Analysis

Algorithm 4 uses $O(m)$ memory and runs in $O(k)$ worst-case time, where $k = \lceil T/\Delta_1 \rceil$ is the number of intervals at the lowest time scale of the algorithm. The per-interval processing time is amortized constant, since the repeat loop starting at Line 6 simply counts the number of trailing zeros in the binary representation of i, for all $0 < i < k = T/\Delta$. The procedure $updatestat()$ called at Line 8 updates in constant time the $O(1)$ information necessary to maintain the statistics for each Δ_i, for $1 \leq i \leq m$.

We now describe and analyze the bandwidth estimation using `EXPB`. Given any query time scale Δ, we output the maximum of the bandwidth corresponding to the smallest index j for which $\Delta_j \geq \Delta$, and use the sum of squared packet masses stored for granularity Δ_j to compute the standard deviation. The following lemma bounds the error of such an approximation.

Lemma 4.3.6. With `EXPB` we can return an estimation of the maximum or standard deviation of S_Δ that is between factor $1/2$ and 3 from the true value. The bound on the standard deviation holds in the limit when the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ is large.

Proof: We first prove the result for the statistic maximum, and then address the standard deviation. Let I be the interval $((i-1)\Delta, i\Delta]$ corresponding to the time scale Δ in which the maximum value is achieved, and let $p(I)$ be this value. Since $\Delta_j \geq \Delta$, there are at two most consecutive intervals I_i^j, I_{i+1}^j at time scale Δ_j that together cover I. By the pigeonhole principle, either I_i^j or I_{i+1}^j must contain at least half the mass of I, and therefore the maximum value at time scale Δ_j is at least 1/2 of the maximum value at Δ. This proves the lower bound side of the approximation. In order to obtain a corresponding upper bound, we simply observe that if I_i^j is the interval at time scale Δ_j with the maximum value, then I_i^j overlaps with at most 3 intervals of time scale Δ. Thus, the maximum value at time scale Δ_j cannot be more than 3 times the maximum at Δ proving an upper bound on the approximation.

The analysis for the standard deviation follows along the same lines, using the observation that $stddev_\Delta = \sqrt{E(S_\Delta^2) - E(S_\Delta)^2}$. An argument similar to the one used for the maximum value holds for the approximation of $E(S_\Delta^2)$. Then assuming the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ to be a constant sufficiently greater than 1 implies the claim. We omit the simple algebra from this extended abstract. $\square$

We note that there is a non-trivial extension of `EXPB` which allows it to work with a set of exponentially increasing time granularities whose common ratio can be any $\alpha > 1$. This can reduce average error. For a general $\alpha > 1$, Algorithm 4 cannot be easily adapted, so we need a generalization of it that uses an event queue while processing measurements to schedule when in the future a new measurement of length Δ_j must be sent to $updatestat()$.

Extensions to `EXPB`

We now describe how to extend the approach of `EXPB` for a set of exponentially increasing time granularities whose common ratio can be any $\alpha > 1$. We define $\Delta_i = \lfloor \alpha^{i-1} \Delta_1 \rfloor$ (removing duplicates) since α^i might not be an integer.

For a general $\alpha > 1$ Algorithm 4 cannot be easily adapted, so we describe a generalization of it. Algorithm 5 makes use of an event queue while processing samples to schedule when in the future a new sample of length Δ_j must be sent to $updatestat()$.

The $updatestat()$ procedure and the estimation algorithm remain the same as for `EXPB`.

Theorem 4.3.7. Algorithm 5 runs in $O(\log m)$ amortized time per sample. Space complexity is worst case $O(m)$.

Algorithm 5: Extension to `EXPB` for any α

```
1  cs=0;
2  foreach $\Delta_j \in \Delta_1, \ldots, \Delta_m$ do
3      EventPush($< \Delta_j$,$\Delta_j, 0 >$);
4  end
5  foreach $s_i$ do
6      cs+=$s_i$;
8      while EventPeek().key $\leq i$ do
9          <key,$\Delta_j$,p_cs>=EventPop(); updatestat(j,p_cs-cs);
10         EventPush($i + \Delta_j$,$\Delta_j$, cs);
11     end
12 end
```

Proof: We first prove the space bound. The loop at Line 8 maintains the invariant that at any time no two elements with the same Δ_j are in the queue. Since there are m possible Δ_j this yields the memory bound.

For the time bound, observe that the total number of push (pop) operations to the event queue for a fixed α is amortized $O(k)$ for k samples (since the Δ_j are in geometric progression of common ratio α). Each operation on the event queue requires $O(\log m)$ (implementing the queue, for instance, as a min-heap), hence the amortized bound. □

Although Algorithm 5 does not give improved error guarantees over `EXPB`, one expects that using $1 < \alpha < 2$ as common ratio would improve accuracy in practice.

4.3.3 Culprit Identification

As mentioned earlier, we do not want to simply identify bursts but also to *identify the flow* (e.g., TCP connection, or source IP address, protocol) that caused the burst so that the network manager can reschedule or move the offending station or application. The naive approach would be to add a heavy-hitters [89] data structure to each DBM bucket, which seems expensive in storage. Instead, we modify `DBM` to include two extra variables per bucket: a flowID and a flow count for the flowID.

The simple heuristic we suggest is as follows. Initially, each packet is placed in a bucket, and the bucket's flowID is set to the flowID of its packet. When merging two buckets, if the buckets have the same flowID, then that flowID becomes the flowID of the merged bucket and the flow counts are summed. If not, then one of the two flowIDs is picked with probability proportional to their flow counts. Intuitively, the higher count flows are more likely to be picked as the main contributor in each bucket as they are more likely to survive merges.

For `EXPB`, a simple idea is to use a standard heavy-hitters structure [89] corresponding to each of the logarithmic time scales. When each counter is reset, we update the flowID if the maximum value has changed and reinitialize the heavy-hitters structure for the next interval. This requires only a logarithmic number of heavy-hitters structures. Since there appears to be redundancy across the structures at each time scale, more compression appears feasible but we leave this for future work.

4.4 Evaluation

We now evaluate the performance and accuracy of `DBM` and `EXPB` to show that they fulfill our goal of a tool that efficiently utilizes memory and processing resources to faithfully capture and display key bandwidth measures. We will show that `DBM` and `EXPB` use significantly fewer resources than packet tracing and are suitable for network-wide measurement and visualization.

4.4.1 Measurement Accuracy

We implemented `EXPB` and the three variants of `DBM` as user-space programs and evaluated them with real traffic traces. Our traces consisted of a packets captured from the 1 Gigabit switch that connects several infrastructure servers used by the Systems and Networking group at U.C. San Diego, and socket-level send data produced by the record-breaking TritonSort sorting cluster [110].

Our “rsync” trace captured individual packets from an 11-hour period during which our NFS server ran its monthly backup to a remote machine using rsync. This trace recorded the transfer of 76.2 GB of data in 60.6 million packets, of which 66.6 GB was due to the backup operation. The average throughput was 15.4 Mbps with a maximum of 782 Mbps for a single second.

The "tritonsort" trace contains time-stamped byte counts from successful `send` system calls on a single host during the sorting of 500 GB of data using 23 nodes connected by a 10 Gbps network. This trace contains an average of 92,488 send events per second, with a peak of 123,322 events recorded in a single 1-second interval. In total, 20.8 GB were transferred over 34.24 seconds for an average throughput of 4.9 Gbps.

Ideally, our evaluation would include traffic from a mix of production applications running over a 10 Gbps network. While we do not have access to such a deployment, our traces provide insight into how `DBM` and `EXPB` might perform given the high bandwidth and network utilization of the "tritonsort" trace and the large variance in bandwidth from second to second in the "rsync" trace.

For our accuracy evaluation, we used an aggregation period of 2 seconds. To avoid problems with incomplete sampling periods in `EXPB`, we must choose our time scales such that they all evenly divide our aggregation period. Since the prime factors of 2 seconds in nsec are 2^{11} and 5^{10} nsec, `EXPB` can use up to 11 buckets. Thus for `EXPB`, we choose the finest time scale to be $\Delta = 78.125$ μs (5^7 nsec) and the coarsest to be $\Delta = 80$ msec ($2^{11}5^7$ nsec), which is consistent with the time scales for interesting bursts in data centers. For consistency, we also configure `DBM` to use a base sampling interval of 78.125 μs, but note that it can answer queries up to $\Delta = 2$ seconds.

To provide a baseline measurement, we computed bandwidth statistics for all of our traces at various time scales where $\Delta \geq 78.125\ \mu$s. To ensure that all measurements in S_Δ are equal, we only evaluated time scales that evenly divided 2 seconds. In total, this provided us with ground-truth statistics at 52 different time scales ranging from 78.125 μs to 2 seconds. In the following sections we report accuracy in terms of error relative to these ground-truth measurements. While any number of values could be used for Δ and T in practice, we used these values across our experiments for the sake of a consistent and representative evaluation between algorithms.

Accuracy vs. Memory

We begin by investigating the tradeoff between memory and accuracy. At one extreme, SNMP can calculate average bandwidth using only a single counter. In contrast, packet tracing with tcpdump can calculate a wide range of statistics with perfect accuracy, but with storage cost scaling linearly with the number of packets. Both `DBM` and `EXPB` provide a tradeoff between these two extremes by supporting complex queries with bounded error, but with orders of magnitude less memory.

For comparison, consider the simplest event trace which captures a 64-bit timestamp and a 16-bit byte length for each packet sent or received. Using this data, one could calculate bandwidth statistics for the trace with perfect accuracy at a memory cost of 6 bytes *per event*. In contrast, `DBM` and `EXPB` require 8 and 16 bytes of storage per bucket used, respectively, along with a few bytes of meta data for each aggregation period.

		Max of Avg. Relative Error					
		$\leq$ 2 msec			$>$ 2msec		
	Output Rate	Max	S.Dev.	95th	Max	S.Dev.	95th
packet trace (avg)	555 KBps						
(peak)	740 KBps	0%	0%	0%	0%	0%	0%
`DBM-mm`, 1000 buckets	4 KBps	25.9%	43.3%	18.3%	2.2%	5.7%	1.1%
`DBM-mv`, 1000 buckets	4 KBps	16.7%	58.9%	26.7%	7.2%	39.0%	10.4%
`DBM-mr`, 1000 buckets	4 KBps	14.0%	35.0%	16.1%	2.0%	4.1%	0.9%
`EXPB`, 11 buckets	96 Bps	2.7%	2.5%	N/A	2.8%	8.1%	N/A

Table 4.1: Memory vs. Accuracy. We evaluate the "tritonsort" trace with a base time scale of Δ =78.125 μs and a 2 second aggregation period. Data output rate is reported for a simple packet trace compared with the `DBM` and `EXPB` algorithms. For each statistic, we compute the max of the average relative error of measurements for each of our reference time scales.

To quantify these differences, we queried our traces for max, standard deviation, and 95th percentile (`DBM` only). For each statistic, we compute the average relative error of the measurements at each of our reference time scales and report the worst-case. To avoid spurious errors due to low sample counts, we omit accuracy data for standard deviations with fewer than 10 samples per aggregation period and 95th percentiles with fewer than 20 samples per aggregation period. We show the tradeoff between storage and accuracy in Table 4.1.

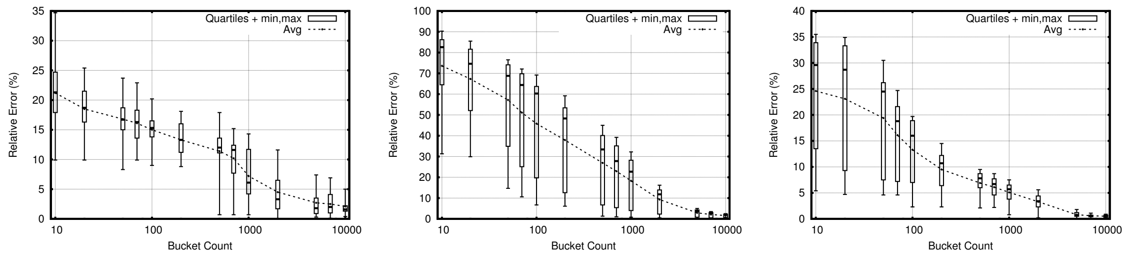
While the simple packet trace gives perfectly accurate statistics, both `DBM` and `EXPB` consume memory at a fixed rate which can be configured by specifying the number of buckets and the aggregation period. In the presented configuration, both `DBM` and `EXPB`

		Max of Avg. Rel. Error		
	Output	Max	S.Dev.	95th
trace (avg)	9.2 KBps			
(peak)	396 KBps	0%	0%	0%
`DBM-mr`	4 KBps	7.6%	14.7%	14.9%
`EXPB`	96 Bps	14.2%	5.9%	N/A

Table 4.2: We repeated our evaluation with the "rsync" trace and report accuracy results for our two best performing algorithms — `DBM-mr` and `EXPB`. We calculated the average relative error for each of our reference time scale and show the worst case.

generate 4 KBps and 96 Bps, respectively — orders of magnitude less memory than the simple trace.

The cost of reduced storage overhead in `DBM` and `EXPB` is the error introduced in our measurements. However, we see that the range of average relative error rates is reasonable for max, standard deviation, and 95th percentile measurements. Further, of the `DBM` algorithms, `DBM-mr` gives the lowest errors throughout. While not shown, `DBM`'s errors are largely due to under-estimation, but its accuracy improves as the query interval grows. `EXPB` gives consistent estimation errors for max across all of our reference points, but gradually degrades for standard deviation estimates as query intervals increase. Thus, for this trace, `EXPB` achieves the lowest error for query intervals less than 2msec. We have divided Table 4.1 to show the worst-case errors in these regions.



(a) Max measurements (b) Std. Dev. measurements (c) 95th Percentile measurements

Figure 4.25: Relative error for DBM-mr algorithm shown for the 400 μs time scale with a varying number of buckets. The box plots show the range of relative errors from the 25th to 75th percentiles, with the median indicated in between. The box whiskers indicate the min and max errors.

In Table 4.2 , we show the accuracy of DBM-mr and EXPB when run on the "rsync" trace with the same parameters as before. We note that again DBM-mr gives the most accurate results for larger query intervals, but now out-performs EXPB for query intervals greater than 160μs for max and 1msec for standard deviation.

To see the effect of scaling the number of buckets, we picked a representative query interval of 400 μs and investigated the accuracy of DBM-mr as the number of buckets were varied. The results of measuring the max, standard deviation and 95th percentile on the "tritonsort" trace are shown in Figure 4.25. We see that the relative error for all measurements decreases as the number of buckets is increased. However, at 4,000 buckets the curves flatten significantly and additional buckets beyond this do not produce any significant improvement in accuracy. While one might expect the error to drop to zero when the number of buckets is equal to the number of samples at S_Δ (5000 samples for 400μs), we do not see this since the trace is sampled at a finer granularity (78.125 μs) and the buckets are merged online. There is no guarantee that DBM will merge the buckets such that each spans exactly 400μs of the trace.

With approximations of the max and standard deviation with this degree of accuracy, we see both DBM and EXPB as an excellent, low-overhead alternative to packet tracing.

DBM Visualization

One unique property of the DBM algorithms is that they can be visualized to show users the shape of the bandwidth curves. Note that we proved earlier that DBM-mr is optimal in some sense in picking out bursts. We now investigate experimentally how all DBM variants do in burst detection.

In Figures 4.26 we show the output for a single, 2 second aggregation period from the "rsync" trace using DBM-mr. For visual clarity, we configured DBM-mr to aggregate measurements at a 4 msec base time scale (250 data points) using 9 buckets. Figure 4.26 shows the raw data points (bandwidth use in each 4 msec interval of the 2 second trace) with the DBM-mr output superimposed. Notice that DBM-mr picks out four bursts (the vertical lines). The fourth burst looks smaller than the 3.1 Mbps burst observable in the raw trace. This is because there were two adjacent measurement intervals in the raw trace with bandwidths of 3.1 and 2.2 Mbps, respectively. DBM-mr merged these measurements into a single bucket of with an average bandwidth of 2.65 Mbps for 8 msec.

We show the output for all `DBM` algorithms in a more clean visual form in Figures 4.27a, 4.27b and 4.27c. We have normalized the width of the buckets and list their start and end times on the x-axis. Additionally, we label each bucket with its mass (byte count). This representation compresses periods of low traffic and highlights short-lived, high-bandwidth events. From the visualization of `DBM-mr` in Figure 4.27c, we can quickly see that there were four periods of time, each lasting between 4 and 8 msec where the bandwidth exceeded 2.3 Mbps. Note that in Figure 4.27a, `DBM-mm` picks out only two bursts. The remaining bursts have been merged into the three buckets spanning the period from 1440 to 1636 msec, thereby reducing the bandwidth (the y-axis) because the total time of the combined bucket increases.

In practice, a network administrator might want to quickly scan such a visualization and look for microburst events. To simulate such a scenario, we randomly inserted three bursts, each lasting 4 msec and transmitting between 4.0 and 4.4 MB of data. We show the `DBM` visualization for this augmented trace in bottom of Figure 4.27. `DBM-mr` and `DBM-mm` both allocate their memory resources to capture all three of these important events, even though they only represent 12 msec of a 2 second aggregation period. Again, `DBM-mr` cleanly picks out the three bursts.

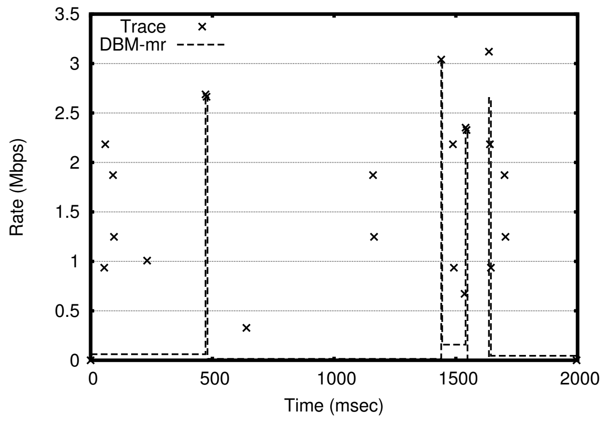
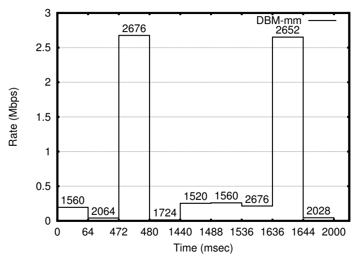
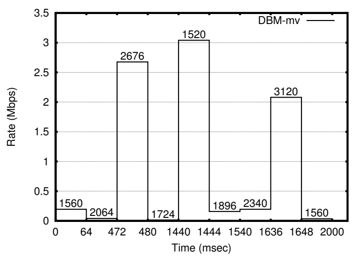


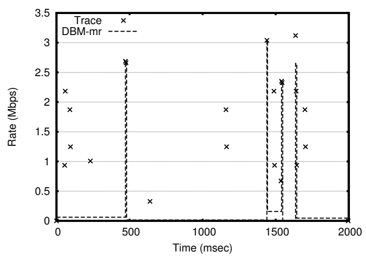
Figure 4.26: Visualization of events from a 2 second aggregation period overlaid with the output of `DBM-mr` using 9 buckets and a 4 msec measurement time scale.

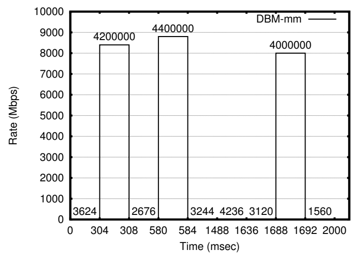
Accuracy at High Load

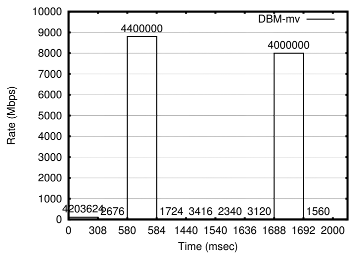
As mentioned in Lemma 4.3.1, the error associated with the `DBM` algorithms increases with the ratio of total packet mass (total bytes) to number of buckets within an aggregation period. We now investigate to what extent increasing the mass within an aggregation period affects the measurement accuracy of `DBM`. To evaluate this, we first configured `DBM` to use a base time scale of $\Delta = 78.125$ μs and 1000 buckets, as before, but vary the mass stored in `DBM` by changing the aggregation period. Figures 4.28a & 4.28b show the change in average relative error for both max and standard deviation statistics in our high-bandwidth "tritonsort" trace at a representative query time scale (400 μs) as the aggregation period is varied between 1 and 16 seconds.


(a) DBM-mm visualization


(b) DBM-mv visualization


(c) DBM-mr visualization


(d) DBM-mm visualization of bursty traffic


(e) DBM-mv visualization of bursty traffic

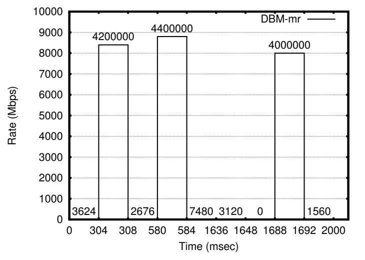
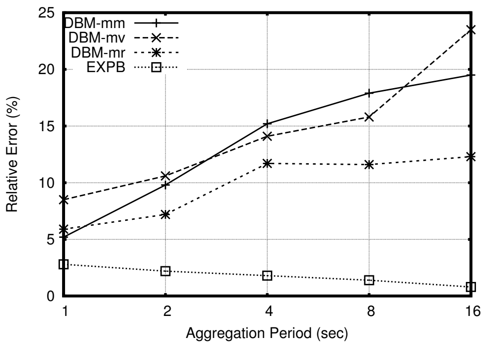
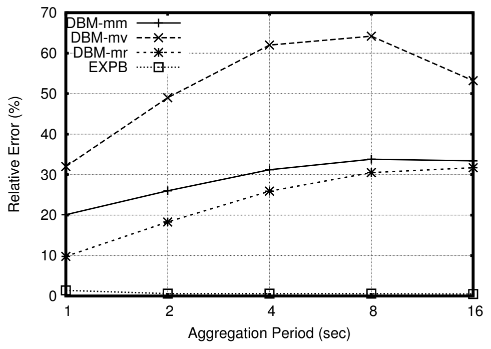

(f) DBM-mr visualization of bursty traffic

Figure 4.27: Visualization of DBM with 9 buckets over a single 2 second aggregation period. The start and end times for each bucket are shown on the x-axis, and each bucket is labeled with its mass (byte count). The top figures show the various DBM approximations of a single aggregation period, while the lower graphs show the same period with three short-lived, high bandwidth bursts randomly inserted.

For DBM-mm and DBM-mv with 1000 buckets the relative error diverges significantly as the aggregation period is increased. In contrast, DBM-mr shows only a subtle degradation for max from 5.9% to 12.3%. For standard deviation, DBM-mv show



(a) Max measurements with 1000 buckets



(b) Std. Dev. measurements with 1000 buckets

Figure 4.28: Average relative error for the DBM with 1000 buckets and EXPB with 11 buckets shown on the “tritonsort” trace for a 400 μs query interval and various aggregation periods.

consistently poor performance with average relative errors increasing from 32% to 64%, while both DBM−mm and DBM−mr trend together with DBM−mr’s errors ranging from 9.8 to 31.7%.

We contrast DBM−mr’s performance for these experiments with that of EXPB. We see that EXPB’s average relative error in the max measurement gradually falls from 2.8% to 1.9% as the aggregation period increases. Further, the error in standard deviation falls from 1.4% at a 1 second aggregation period to 0.5% at 16 seconds.

These results indicate that degradation in accuracy does occur as the ratio of the total packet mass to bucket count increases, as predicted by Lemma 4.3.1. While DBM must be configured correctly to bound the ratio of packet mass to bucket count, EXPB’s accuracy is largely unaffected by the packet mass or aggregation period.

4.4.2 Performance Overhead

As previously stated, we seek to provide an efficient alternative to packet capture tools. Hence we compare the performance overhead of `DBM` and `EXPB` to that of an unmodified vanilla kernel, and to the well-established tcpdump[5].

We implemented our algorithms in the Linux 2.6.34 kernel along with a userspace program to read the captured statistics and write them to disk. To provide greater computational efficiency we constrained the base time scale and the aggregation period to be powers of 2. The following experiments were run on 2.27 GHz, quad-core Intel Xeon servers with 24 GB of memory. Each server is connected to a top-of-rack switch via 10 Gbps ethernet and has a round trip latency of approximately 100 μs.

To quantify the impact of our monitoring on performance, we first ran iperf [3] to send TCP traffic between two machines on our 10 Gbps network for 10 seconds. In addition, we instrumented our code to report the time spent in our routines during the test. We first ran the vanilla kernel source, then added different versions of our monitoring to aggregate 64 μs intervals over 1 second periods. We report both the bandwidth achieved by iperf and the average latency added to each packet at the sending server in Table 4.3. For comparison, we also report performance numbers for tcpdump when run with the default settings and writing the TCP and IP headers (52 bytes) of each packet directly to

Version	Buckets	Avg. BW	Overhead/Pkt
vanilla	N/A	9.053 Gbps	0.0 nsec
`DBM-mm`	10	9.057 Gbps	256.5 nsec
	100	9.010 Gbps	335.7 nsec
	1000	9.104 Gbps	237.5 nsec
	10000	8.970 Gbps	560.4 nsec
`DBM-mv`	10	9.043 Gbps	205.7 nsec
	100	8.986 Gbps	327.9 nsec
	1000	9.067 Gbps	432.2 nsec
	10000	9.067 Gbps	457.2 nsec
`EXPB`	14	9.109 Gbps	169.4 nsec
tcpdump	N/A	8.732 Gbps	N/A

Table 4.3: Average TCP bandwidth reported by iperf over 60 10-second runs. We also show the average time spent in the kernel-level monitoring functions for each packet sent. `DBM` and `EXPB` were run with a base time scale of $\Delta = 64\ \mu$s and $T =$ 1 second aggregation period.

local disk. As `DBM-mr` is nearly identical to `DBM-mm` with respect to implementation, we omit `DBM-mr`'s results.

As discussed in section 4.3.1, we see that the latency overhead per packet increases roughly as the log of the number of buckets. However, iperf's maximum throughput is not degraded by the latency added to each packet. Since the added latency per packet is several orders of magnitude less than the RTT, the overhead of `DBM` should not affect TCP's ability to quickly grow its congestion window. In contrast to `DBM`, tcpdump achieves 3.5% less throughput.

To observe the overhead of our monitoring on an application, we transferred a 1GB file using `scp`. We measured the wall-clock time necessary to complete the transfer by running scp within the Linux's `time` utility. To quantify the affects of our measurement on the total completion time, we measured the total overhead imposed on packets as they moved up and down the network stack. We report this overhead as a percentage of each experiment's average completion time (monitoring time divided by scp completion time). Each experiment was replicated 60 times and results are reported in Table 4.4. We see that although the cumulative overhead added by `DBM` grows logarithmically with the number of buckets, the time for scp to complete increases by at most 4.5%.

We see that our implementations of `DBM` and `EXPB` have a negligible impact on application performance, even while monitoring traffic at 10 Gbps.

4.4.3 Evaluation Summary

Our experiments indicate `DBM-mr` consistently provides better burst detection and has reasonable average case and worst case error for various statistics. When measuring at arbitrary time scales, `EXPB` has comparable or better average and worst-case error than `DBM` while using less memory. In addition, `EXPB` is unaffected by high mass in a given aggregation period. On other hand, `DBM` can approximate time series, which is useful for seeing how burst are distributed in time and for calculating more advanced statistics

Version	Buckets	Time	Overhead
vanilla	N/A	14.133 sec	N/A
`DBM-mm`	10	14.334 sec	1.3%
	100	14.765 sec	1.9%
	1000	14.483 sec	2.7%
	10000	14.527 sec	2.5%
`DBM-mv`	10	14.320 sec	1.7%
	100	14.344 sec	2.3%
	1000	14.645 sec	2.9%
	10000	14.482 sec	3.1%
`EXPB`	14	14.230 sec	0.4%
tcpdump	N/A	15.253 sec	7.9%

Table 4.4: The time needed to transfer a 1GB file over `scp`. We measured the cumulative overhead incurred by our monitoring routines for all send and receive events. We report this overhead as a percentage of each experiment's total running time.

(i.e. percentiles). We recommend a parallel implementation where `EXPB` is used for Max and Standard Deviation and `DBM-mr` is used for all other queries.

4.5 System Implications

So far, we have described `DBM` and `EXPB` as part of an end host monitoring tool that can aggregate and visualize bandwidth data with good accuracy. However, we see these algorithms a part of a larger infrastructure monitoring system.

Long-term Archival and Database Support It is useful for administrators to retrospectively troubleshoot problems that are reported by customers days after the fact. At slightly more than 4 KBps, the data produced by both `DBM` and `EXPB` for a week (2.4 GB per link) could easily be stored to a commodity disk. With this data, an administrator can pinpoint traffic abnormalities at microsecond timescales and look for patterns across links. The data can be compacted for larger time scales by reducing granularity for older data. For example, one hour of `EXPB` data could be collapsed into one set of buckets containing max and standard deviation information at the original resolutions but aggregated across the hour.

With such techniques, fine-grain network statistics for hundreds of links over an entire year could be stored to a single server. The data could be keyed by link and time and stored in a relational database to allow queries across time (is the traffic on a single link becoming more bursty with time?) or across links (did a number of bursts correlate on multiple switch input ports?).

Hardware Implementation Both `DBM` and `EXPB` algorithms can be implemented in hardware for use in switches and routers. `EXPB` has an amortized cost of two bucket updates per measurement interval. Since bucket updates are only needed at the frequency of the measurement time scale, these operations could be put on a work queue and serviced asynchronously from the main packet pipeline. The key complication for implementing `DBM` in hardware is maintaining a binary heap. However, a 1000 bucket heap can be maintained in hardware using a 2-level radix-32 heap that uses 32-way comparators at 10 Gbps. Higher bucket sizes and speeds will require pipelining the heap. The extra hardware overhead for these algorithms in gates is minimal. Finally, the logging overhead is very small, especially when compared to NetFlow.

4.6 Conclusions

Picking out bursts in a large amount of resource usage data is a fundamental problem and applies to all resources, whether power, cooling, bandwidth, memory, CPU, or even financial markets. However, in the domain of data center networks, the increase of network speeds beyond 1 Gigabit per second and the decrease of in-network buffering has made the problem one of great interest.

Managers today have little information about how microbursts are caused. In some cases they have identified paradigms such as InCast, but managers need better visibility into bandwidth usage and the perpetrators of microbursts. They would also like better understanding of the temporal dynamics of such bursts. For instance, do they happen occasionally or often? Do bursts linger below a tipping point for a long period or do they arise suddenly like tsunamis? Further, correlated bursts across links lead to packet drops. A database of bandwidth information from across an administrative domain would be valuable in identifying such patterns. Of course, this could be done by logging a record for every packet, but this is too expensive to contemplate today.

This chapter provides the first step to realizing such a vision for a cheap network-wide bandwidth usage database by showing efficient summarization techniques at links

($\sim$4 KB per second, for example, for running `DBM` and `EXPB` on 10 Gbps links) that can feed a database backend as shown in Figure 4.22. Ideally, this can be supplemented by algorithms that also identify the flows responsible for bursts and techniques to join information across multiple links to detect offending applications and their timing. Of the two algorithms we introduce, Exponential Bucketing offers accurate measurement of the average, max and standard deviation of bandwidths at arbitrary sampling resolutions with very low memory. In contrast, Dynamic Bucket Merge approximates a time-series of bandwidth measurements that can visualized or used to compute advanced statistics, such as quantiles.

While we have shown the application of `DBM` and `EXPB` to bandwidth measurements in endhosts, these algorithms could be easily ported to in-network monitoring devices or switches. Further, these algorithms can be generally applied to any time-series data, and will be particularly useful in environments where resource spikes must be detected at fine time scales but logging throughput and archival memory is constrained.

4.7 Acknowledgements

This research was supported by grants from the NSF (CNS-0964395) and Cisco. We would also like to thank our shepherd, Dave Maltz, and our anonymous reviewers for their insightful comments and feedback.

Conclusion

In this dissertation we analyzed several problems stemming from diverse domains of computer science. The unifying theme of these problems is that an exact solution is impractical to compute in the general case. In particular, the dissertation focuses on problems on weighted graphs (Chapters 1 and 2) and streams of data (Chapters 3 and 4).

Chapter 1 considers the problem of finding a time-dependent shortest path in a n-node graph where delays on the edges are functions of time. We showed that if one wants to plot the arrival time to destination as a function of the departure time, the plotted function might change slope a super-polynomial number of times. To overcome this issue, we presented an approximation algorithm to plot such function using much fewer breakpoints but retaining a relative-error type guarantee on the arrival time at destination. It remains open to study the complexity of the arrival time function for restricted graph classes, planar graphs in particular.

In Chapter 2 we studied the problem of balanced partitions of graphs, where one wants to partition a vertex set of a graph into k disjoint subsets of approximately equal size such that a cut metric is minimized. We showed that the problem is hard to approximate even when the graph class is drastically restricted and described a bicriteria PTAS with logarithmic approximation guarantees on the cut metric. A few questions remain unanswered. Most notably, it is unknown whether it is possible to devise algorithms with a better tradeoff between approximation and runtime than the ones presented here.

In the second part of the dissertation, we turned our attention to problems defined on streams of data. In Chapter 3 we developed an algorithmic framework that yields provably error guarantees for approximating time series under various metrics in the data stream model and other variations. The algorithms developed under the framework's umbrella require little memory footprint and run extremely fast. The framework is proven to adapt to many error metrics (e.g., L_p norm). However, whether the framework can provide algorithms for other metrics of practical interests (e.g., L_1 norm) remains an interesting open question.

Chapter 4 develops the ideas presented in Chapter 3 to provide methods of detecting traffic bursts at various and especially fine-grain, time scales in computer networks. The simplicity of our algorithms allowed implementing an efficient and flexible end-host bandwidth measurement tool to identify traffic bursts at small time scales. The results are encouraging in the fact that our algorithms could actually detect traffic anomalies at wire speed on a Giga-bit link requiring only a small amount of resources. Future development will include complementing our framework with algorithms to join information across multiple links in order to detect offending applications.

Bibliography

[1] A Simple Network Management Protocol (SNMP). http://www.ietf.org/rfc/rfc1157.txt.

[2] Cisco Netflow. www.cisco.com/web/go/netflow.

[3] iperf. http://http://iperf.sourceforge.net/.

[4] Performance Management for Latency-Intolerant Financial Trading Networks. *Financial Service Technology*, 9.

[5] tcpdump. http://www.tcpdump.org/.

[6] M. Abam, M. Berg, P. Hachenberger, and A. Zarei. Streaming algorithms for line simplification. In *SOCG*, 2007.

[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *SIGCOMM*, 2010.

[8] K. Andreev and H. Räcke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[9] P. Arbenz, G. van Lenthe, U. Mennel, R. Müller, and M. Sala. Multi-level μ-finite element analysis for human bone structures. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *LNCS*, pages 240–250. Springer, 2007.

[10] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proceedings of the 26th annual ACM symposium on Theory of computing (STOC)*, pages 222–231, 2004.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[12] N. Bansal, D. Coppersmith, and B. Schieber. Minimizing setup and beam-on times in radiation therapy. In J. Díaz, K. Jansen, J. Rolim, and U. Zwick, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, volume 4110 of *LNCS*, pages 27–38. Springer, 2006.

[13] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *J. Alg.*, 31(1):1–28, 1999.

[14] D. P. Bertsekas and J. N. Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991.

[15] S. Bhatt and F. T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984.

[16] H. Bodlaender, P. Schuurman, and G. Woeginger. Scheduling of pipelined operator graphs. *Journal of Scheduling*, 15:323–332, 2012.

[17] C. Buragohain, N. Shrivastava, and S. Suri. Space Efficient Streaming Algorithms for the Maximum Error Histogram. In *ICDE*, 2007.

[18] C. Busch and S. Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *STACS*, 2007.

[19] X. Cai, T. Kloks, and C. K. Wong. Time-varying shortest path problems with constraints. *Networks*, 29(3):141–150, 1997.

[20] P. J. Carstensen. Parametric cost shortest path problems. Unpublished Bellcore memo, 1984.

[21] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Record: Journal of the Transportation Research Board*, 1645:170–175, 1998.

[22] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3), 2001.

[23] K. Chan and A. Fu. Efficient time series matching by wavelets. In *ICDE*, 1999.

[24] Y. Chen, G. Dong, J. Han, B. Wah, and J. Wang. Multidimensional regression analysis of time-series data streams. In *ICDE*, 2002.

[25] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN 2009*.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[27] G. Cormode, F. Korn, and S. Tirthapura. Time-decaying aggregates in out-of-order streams. In *PODS*, 2008.

[28] B. C. Dean. Continuous-time dynamic shortest path algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[29] B. C. Dean. Shortest paths in FIFO time-dependent networks: Theory and algorithms. Technical report, 2004.

[30] F. Dehne, M. Omran, and J. Sack. Shortest paths in time-dependent fifo networks. *Algorithmica*, pages 1–20, 2010.

[31] F. Dehne, M. T. Omran, and J.-R. Sack. Shortest paths in time-dependent FIFO networks using edge load forecasts. In *IWCTS '09: Proceedings 2nd International Workshop on Computational Transportation Science*, pages 1–6. ACM, 2009.

[32] D. Delling, A. Goldberg, T. Pajor, and R. Werneck. Customizable route planning. *Experimental Algorithms*, pages 376–387, 2011.

[33] D. Delling and D. Wagner. Time-dependent route planning. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[34] C. Demetrescu and G. F. Italiano. Dynamic shortest paths and transitive closure: An annotated bibliography (draft). `www.diku.dk/PATH05/biblio-dynpaths.pdf`, 2005.

[35] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[36] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of 20th International Parallel and Distributed Processing Symposium*, 2006.

[37] J. Díaz, J. Petit, and M. J. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

[38] J. Díaz, M. J. Serna, and J. Torán. Parallel approximation schemes for problems on planar graphs. *Acta Informatica*, 33(4):387–408, 1996.

[39] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, pages 205–216. ACM, 2008.

[40] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

[41] J. Erickson. Maximum flows and parametric shortest paths in planar graphs, 2010. Submitted to the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (2010).

[42] A. Erramilli, O. Narayan, and W. Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Trans. Netw.*, 4(2):209–223, 1996.

[43] G. Even, J. Naor, S. Rao, and B. Schieber. Fast approximate graph partitioning algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.

[44] A. E. Feldmann. Fast balanced partitioning of grid graphs is hard. *ArXiv e-prints*, (arXiv:1111.6745v1), November 2011.

[45] A. E. Feldmann. *Balanced Partitioning of Grids and Related Graphs: A Theoretical Study of Data Distribution in Parallel FEM Simulations*. PhD thesis, ETH Zurich, 2012.

[46] A. E. Feldmann, S. Das, and P. Widmayer. Restricted cuts for bisections in solid grids: A proof via polygons. In *Proceedings of the 37th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 143–154, 2011.

[47] A. E. Feldmann and L. Foschini. Balanced partitions of trees and applications. In *29th International Symposium on Theoretical Aspects of Computer Science (STACS)*, 2012.

[48] A. E. Feldmann and L. Foschini. Balanced partitions of trees and applications. *Algorithmica*, 2012. Submitted.

[49] A. E. Feldmann and P. Widmayer. An $O(n^4)$ time algorithm to compute the bisection width of solid grid graphs. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA)*, pages 143–154, 2011.

[50] T. Feo, O. Goldschmidt, and M. Khellaf. One-half approximation algorithms for the k-partition problem. *Operations Research*, 40:170–173, 1992.

[51] T. Feo and M. Khellaf. A class of bounded approximation algorithms for graph partitioning. *Networks*, 20(2):181–195, 1990.

[52] D. Fernández-Baca and G. Slutzki. Parametric problems on graphs of bounded tree-width. *J. Algorithms*, 16(3):408–430, 1994.

[53] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[54] L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. SODA, 2011.

[55] L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, 2012. Submitted.

[56] S. Gandhi, L. Foschini, and S. Suri. Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order. In *ICDE*, 2010.

[57] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[58] A. Gilbert, Y. Kotidis, S. Guha, S. Muthukrishnan, et al. Fast small-space algorithms for approximate histogram maintenance. In *STOC*, 2002.

[59] S. Guha. On the space—time of optimal, approximate and streaming algorithms for synopsis construction problems. *The VLDB Journal*, 17(6):1509–1535, 2008.

[60] S. Guha. Tight results for clustering and summarizing data streams. In *Departmental Papers, Department of Computer and Information Science, Univeristy of Pennsylvania*, 2009.

[61] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *STOC*, 2001.

[62] S. Guha, N. Koudas, and K. Shim. Approximation and streaming algorithms for histogram construction problems. *ACM TODS*, 31(1):396–438, 2006.

[63] L. J. Guibas. Kinetic data structures — a state of the art report. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209. A. K. Peters, Wellesley, MA, 1998.

[64] D. M. Gusfield. *Sensitivity analysis for combinatorial optimization*. PhD thesis, University of California, Berkeley, 1980.

[65] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[66] B. Hendrickson and R. Leland. The chaco users guide version 2.0. Technical report, Technical Report SAND95-2344, Sandia National Laboratories, 1995.

[67] J. Hershberger, N. Shrivastava, S. Suri, and C. Toth. Adaptive Spatial Partitioning for Multidimensional Data Streams. *Algorithmica*, 2006.

[68] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988.

[69] H. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, et al. Optimal histograms with quality guarantees. In *VLDB*, 1998.

[70] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *Proceedings of the 22st International Conference on Data Engineering (ICDE)*, 2006.

[71] R. M. Karp and J. B. Orlin. Parameter shortest path algorithms with an application to cyclic staffing. Technical report, Massachusetts Institute of Technology, Operations Research Center, 1980.

[72] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359, 1999.

[73] J. Kelner and D. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, page 60. ACM, 2006.

[74] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD*, 2001.

[75] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In *In ICDM*, 2001.

[76] E. Keogh and M. Pazzani. A simple dimensionality reduction technique for fast similarity search in large time series databases. In *PADKK*, 2000.

[77] E. Keogh, X. Xi, L. Wei, and C. Ratanamahatana. The ucr time series classification/clustering homepage. 2006.

[78] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 81–90, 1999.

[79] D. G. Kirkpatrick and P. Hell. On the completeness of a generalized matching problem. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 240–245, 1978.

[80] P. Klein, S. Plotkin, and S. Rao. Excluded minors, network decomposition, and multicommodity flow. In *Proceedings of the 25th annual ACM symposium on Theory of computing*, pages 682–690, 1993.

[81] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *SIGCOMM 2009*.

[82] F. Korn, H. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD*, 1997.

[83] R. Krauthgamer, J. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 942–949, 2009.

[84] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics*, 22(3):277–286, 2003.

[85] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

[86] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1), 2008.

[87] M. Li, M. Liu, L. Ding, E. Rundensteiner, and M. Mani. Event stream processing with out-of-order data arrival. In *ICDCS Workshop*, 2007.

[88] R. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM journal on computing*, 9:615–627, 1980.

[89] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. ElephantTrap: A low cost device for identifying large flows. In *HOTI*, 2007.

[90] R. M. MacGregor. *On partitioning a graph: a theoretical and empirical study.* PhD thesis, University of California, Berkeley, 1978.

[91] A. Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 245 –254, 2010.

[92] R. Martin. Wall Street's Quest To Process Data At The Speed Of Light. *Information Week*, April 23 2007.

[93] C. S. Mata and J. S. B. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions (extended abstract). In *Symposium on Computational Geometry*, pages 264–273, 1997.

[94] J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *J. ACM*, 38(1):18–73, 1991.

[95] K. Mulmuley and P. Shah. A lower bound for the shortest path problem. In *Proceedings 15th Annual IEEE Conference on Computational Complexity*, pages 14–21, 2000.

[96] S. Muthukrishnan. *Data streams: Algorithms and applications.* now Publishers Inc., 2005.

[97] S. Muthukrishnan and S. Strauss. Approximate histogram and wavelet summaries of streaming data. In *DIMACS TR 52*, 2003.

[98] K. Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

[99] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A^* search for time-dependent fast paths. In *Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 334–346. Springer, 2008.

[100] S. Nath. Energy efficient sensor data logging with amnesic flash storage. In *IPSN*, 2009.

[101] E. Nikolova, M. Brand, and D. R. Karger. Optimal route planning under uncertainty. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2006.

[102] E. Nikolova, J. A. Kelner, M. Brand, and M. Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *14th European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 552–563. Springer, 2006.

[103] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[104] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, 2004.

[105] J. K. Park and C. A. Phillips. Finding minimum-quotient cuts in planar graphs. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 766–775, 1993.

[106] E. Petrank. The hardness of approximation: Gap location. *Computational Complexity*, 4(2):133–157, 1994.

[107] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*.

[108] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, 2008.

[109] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, 1997.

[110] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI 2011*.

[111] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *12th Annual European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 580–591. Springer, 2004.

[112] H. D. Sherali, K. Ozbay, and S. Subramanian. The time-dependent shortest pair of disjoint paths problem: Complexity, models, and algorithms. *Networks*, 31(4):259–272, 1998.

[113] J. Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, 2000.

[114] D. Shmoys. Cut problems and their application to divide-and-conquer. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 192–235. PWS Publishing Co., 1996.

[115] H. D. Simon and S. H. Teng. How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[116] K. Soumyanath and J. S. Deogun. On the bisection width of partial k-trees. In *Proceedings of the 20th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, volume 74 of *Congressus Numerantium*, pages 25–37, 1990.

[117] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 112–119. ACM, 2005.

[118] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3), 2003.

[119] F. Uyeda, L. Foschini, F. Baker, S. Suri, and G. Varghese. Efficiently measuring bandwidth at all time scales. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 6–6. USENIX Association, 2011.

[120] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *SIGCOMM 2009*.

[121] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[122] C. Willmott and K. Matsuura. Global air temperature and precipitation: Regridded monthly and annual climatologies. In *http://climate.geog.udel.edu/*.

[123] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(11):1101–1113, 1993.

[124] N. Young, R. Tarjan, and J. Orlin. Faster parametric shortest path and minimum balance algorithms. *Networks*, 21(2):205–221, 2006.