universität wien

Dissertation / Doctoral Thesis

Titel der Dissertation / Title of the Doctoral Thesis

Provably Finding and Exploiting Patterns in Data

verfasst von / submitted by

Stefan Neumann

angestrebter akademischer Grad / partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2020 / Vienna, 2020

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt: field of study as it appears on the student record sheet:	Informatik
Betreuerin / Supervisor:	Univ.-Prof. Dr. Monika Henzinger

Abstract

In the last decade, there has been immense progress and growth in the fields of artificial intelligence, data mining and machine learning. This development was enabled by specialized new hardware, the ever-larger availability of data and breakthroughs in the development of algorithms that *find and exploit patterns in the data.* While these methods are known to be extremely successful in practice, our theoretical understanding of them is still limited. However, formal guarantees for these algorithms are highly desirable, because they yield important insights into the strengths and the limitations of these algorithms.

In this thesis, we make an effort to narrow the gap between theory and practice. To this end, we develop algorithms for *provably* finding and exploiting patterns in data. The results are summarized as follows:

- *Provably Finding Patterns:* We provide algorithms that provably extract a set of planted clusters from random bipartite graphs. This problem has applications, for example, in the analysis of online shopping data, where one wishes to identify groups of products that are frequently bought together and groups of customers who purchase similar products. We present the first algorithm which can provably extract *tiny* planted clusters. This result provides a theoretical justification for the success of existing heuristic methods that are used in practice. We further present the first streaming algorithm which, after two passes over a bipartite graph, returns a set of planted clusters and which scales to much larger datasets than existing methods.
- *Understanding the Complexity of Finding Patterns:* We study the computational complexity of frequently-used subroutines of data mining algorithms and provide new hardness results. We show that for approximately computing the number of triangles in a graph and for approximating the support of itemsets in transactional databases, existing random sampling algorithms cannot be significantly improved unless popular conjectures in computational complexity are false. Furthermore, we present a hierarchy of the enumeration complexity of several maximal frequent pattern mining problems and we provide a condition under which this hierarchy collapses.
- *Provably Exploiting Patterns:* We formulate a theoretical model that captures the structure of the patterns in real-world communication networks. We provide algorithms that reduce the network traffic by exploiting these patterns and we show that the competitive ratios obtained by our algorithms are asymptotically optimal.

To obtain our results, we use *beyond worst-case analysis*, i.e., instead of considering worst-case inputs for our algorithms, we consider inputs that satisfy the properties of real-world datasets.

Zusammenfassung

Im letzten Jahrzehnt gab es immensen Fortschritt und Wachstum in den Bereichen der Künstlichen Intelligenz, des Data Mining und des Maschinellen Lernens. Diese Entwicklung wurde durch spezialisierte neue Hardware, die immer größere Verfügbarkeit von Daten und Durchbrüche bei der Entwicklung von Algorithmen, die *Muster in Daten finden und ausnutzen*, ermöglicht. Obwohl wir uns in der Praxis täglich vom großen Erfolg dieser Algorithmen überzeugen können, ist unser theoretisches Verständnis von ihnen weiterhin eingeschränkt. Allerdings wären formale Garantien für diese Algorithmen höchst wünschenswert, weil sie wichtige Einblicke in die Stärken und die Grenzen dieser Algorithmen erlauben.

Das Ziel dieser Dissertation ist es, die Kluft zwischen Theorie und Praxis zu verkleinern. Dafür entwickeln wir Algorithmen, die *beweisbar* Muster in Daten finden und ausnutzen. Die Zusammenfassung der Ergebnisse ist wie folgt:

- *Beweisbar Muster finden:* Wir entwickeln Algorithmen, die beweisbar eine Menge von versteckten Clusterns aus bipartiten Zufallsgraphen extrahieren. Eine Anwendung von diesem Problem ist die Analyse von Onlineshopping-Daten, in denen Gruppen von Produkten gesucht werden, die häufig gemeinsam gekauft werden, und Gruppen von Kund·innen, die ähnliche Produkte kaufen. Wir präsentieren den ersten Algorithmus, der beweisbar *winzige* versteckte Cluster finden kann. Dieses Resultat liefert eine theoretische Begründung für den Erfolg bereits existierender heuristischer Methoden, die in der Praxis verwendet werden. Außerdem präsentieren wir den ersten Datenstromalgorithmus, der nur zwei Iterationen über die Knoten eines bipartiten Graphen vornimmt und anschließend eine Menge an versteckten Clustern ausgibt. Der Algorithmus skaliert auf viel größere Daten als bekannte Methoden.
- *Verstehen der Komplexität des Musterfindens:* Wir betrachten die Komplexität häufig verwendeten Subroutinen von Data Mining-Algorithmen und erhalten neue Härteresultate. Für die approximative Berechnung der Anzahl der Dreiecke in einem Graphen und für die Approximation der Häufigkeit von Itemsets in transaktionalen Datenbanken zeigen wir, dass existierende Algorithmen, die auf zufälligen Stichprobenverfahren beruhen, nicht signifikant verbessert werden können, außer gängige Hypothesen der Komplexitätstheorie sind falsch. Außerdem präsentieren wir eine Hierarchie der Enumerationskomplexität von mehreren Maximal Frequent Pattern Mining-Problemen und wir bestimmen eine Bedingung, unter der diese Hierarchie zusammenbricht.

- *Beweisbar Muster ausnutzen:* Wir formulieren ein theoretisches Modell, das die Struktur von Mustern in realen Kommunikationsnetzwerken abbildet. Wir entwickeln Algorithmen, die die Muster in den Daten ausnutzen und damit den Datenverkehr im Netzwerk reduzieren. Wir beweisen, dass der kompetitive Faktor unserer Algorithmen asymptotisch optimal ist.

Um unsere Resultate zu erhalten, verwenden wir die *beyond worst-case* Analyse: Anstatt von worst-case Eingaben für unsere Algorithmen auszugehen, nehmen wir an, dass die Eingabedaten die Eigenschaften von praktischen Datensätzen erfüllen.

Acknowledgments

This thesis would not exist without the support of many people.

First, I am deeply grateful to my generous supervisor Monika Henzinger. Thank you for giving me a great amount of freedom to work on projects of my own and for having faith in me, even when I doubted myself. During this Ph.D., I have learned a lot and a lot of it I have learned from you. Thank you!

Next, I am happy Aris Gionis and Rasmus Pagh agreed to be the reviewers of this thesis. I could not have hoped for a better committee.

Two major influences for the results in this thesis have been Pauli Miettinen and Jilles Vreeken. What I learned in your courses and our discussions has had a lasting impact on my research. Pauli, thank you for our collaborations and for hosting me in Kuopio! Jilles, thank you for giving me advice and some cheering up whenever I needed it!

I am highly grateful to Eli Upfal for hosting me at Brown University and for inviting me to the CaStleD workshop in Bertinoro. I have thoroughly enjoyed this time. The stay at Brown would not have been the same without the great discussions with Vincent Cohen-Addad, Phil Klein and his group, everyone in Eli's group and my superb office mate Leonardo Pellegrina. Thank you all!

I am indebted to all my other coauthors from whom I have learned a lot. I am grateful to Andy Wiese for writing my first paper with me and for our more recent collaborations, to Rainer Gemulla for helping me connect matrix rounding tricks and communication complexity, to Sayan Bhattacharya for studying cell-probe lower bounds with me, to Andrea Lincoln and Virginia Vassilevska Williams for our joint project on conditional lower bounds, to Kailash Budhathoki and Julian Ritter for turning our Twitter discussions into a workshop paper, to Harald Räcke and Stefan Schmid for getting me interested in online algorithms and, most recently, to Holger Dell and Wolfgang Dvořák for bearing with me when I wanted to connect approximate counting and frequency estimation in databases.

Being a part of the TAA group for the last few years has been great and I could not have asked for better colleagues than the ones I have had. Thanks to Veronika Loitzenbauer for the warm welcome and a lot of advice during the start of the Ph.D., to Gramoz Goranci for all the discussions about life (academic and non-academic), to Wolfgang Dvořák for being a true "Hawara", to Alexander Noe and Wolfgang Ost for being the most tolerant officemates I could have hoped for, to Alexander Svozil for resolving the mystery of the beeping noise, to Yun Kuen Cheung for our long discussions about all aspects of computer science research and to Marcelo Fonseca Faraj, Sebastian Forster, Kathrin Hanauer, Rudolf Hürner, Wanchote Jiamjitrak, Sagar Kale, Shahbaz

Khan, Dariusz Leniowski, Richard Paul, Ami Paz, Pan Peng, Christian Schulz, Bernhard Schuster and Xiaowei Wu for being such bright and nice colleagues. Furthermore, I am grateful to everyone who helped me battle the bureaucracy of University of Vienna: Birgit Aubrunner, Ulrike Frolik-Steffan, Iris Gundacker, Christina Licayan, and Werner Schröttner. Especially the chats with Ulli always cheered me up after frustrating administrative tasks.

Also, I am grateful to the fellow Ph.D. students at the Vienna Graduate School on Computational Optimization for all the drinks we shared after our lectures.

Next, thanks to everyone who supported my career by writing recommendation letters and all reviewers of my papers for their helpful comments and suggestions.

Finally, I am indebted to my friends and my family. Without their support, I would not be where I am today.

Funding Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement No. 340506, and the Doctoral Programme "Vienna Graduate School on Computational Optimization" which is funded by the Austrian Science Fund (FWF, project no. W1260-N35).

Bibliographic Note

The chapters of this thesis are based on the following publications and manuscripts, which are related to my work on finding and exploiting patterns in data:

- **Chapter 2:** Stefan Neumann. "Bipartite Stochastic Block Models with Tiny Clusters". In: *NeurIPS*. 2018, pp. 3871–3881.
  Stefan Neumann. "Finding Tiny Clusters in Bipartite Graphs". In: *INFORMATIK*. Session *Best of Data Science Made in Germany, Austria and Switzerland*. 2019.
- **Chapter 3:** Stefan Neumann and Pauli Miettinen. "Biclustering and Boolean Matrix Factorization in Data Streams". In: *PVLDB*. 2020. To appear.
- **Chapter 4:** Holger Dell, Wolfgang Dvořák, and Stefan Neumann. *Conditional Hardness for Approximate Counting Problems*. Manuscript. 2020. Authors ordered alphabetically.
- **Chapter 5:** Stefan Neumann and Pauli Miettinen. "Reductions for Frequency-Based Data Mining Problems". In: *ICDM*. 2017, pp. 997–1002.
- **Chapter 6:** Monika Henzinger, Stefan Neumann, and Stefan Schmid. "Efficient Distributed Workload (Re-)Embedding". In: *POMACS* 3.1 (2019), 13:1–13:38. Authors ordered alphabetically. Conference version in *SIGMETRICS'19*.
- **Chapter 7:** Monika Henzinger, Stefan Neumann, Harald Räcke, and Stefan Schmid. *Tight Bounds for Online Graph Partitioning*. Manuscript. 2020. Authors ordered alphabetically.

The chapters are written in such a way that each chapter can be read and understood on its own.

During my Ph.D., I have also published about other topics in data mining [143, 152, 155] and algorithms [32, 98, 99, 102, 103, 156], but these results are not included in this thesis.

Contents

1 Introduction **1**
- 1.1 Provably Finding Patterns . . . 3
- 1.2 Understanding the Complexity of Finding Patterns . . . 5
- 1.3 Provably Exploiting Patterns . . . 7

2 Bipartite Stochastic Block Models with Tiny Clusters **11**
- 2.1 Introduction . . . 11
- 2.2 Related Work . . . 14
- 2.3 Recovering the Left Clusters . . . 15
- 2.4 Recovering the Right Clusters . . . 17
- 2.5 Implementation . . . 21
- 2.6 Experiments . . . 22
- 2.7 Conclusion . . . 30

3 Biclustering and Boolean Matrix Factorization in Data Streams **31**
- 3.1 Introduction . . . 31
- 3.2 Preliminaries . . . 33
- 3.3 First Pass: Recover Right Clusters . . . 37
- 3.4 Second Pass: Recover Left Clusters . . . 41
- 3.5 Implementation . . . 43
- 3.6 Experiments . . . 46
- 3.7 Theoretical Guarantees . . . 52
- 3.8 Related Work . . . 57
- 3.9 Conclusion . . . 57

4 Conditional Hardness of Approximate Counting **59**
- 4.1 Introduction . . . 60
- 4.2 Our Results . . . 61
- 4.3 Related Work . . . 66
- 4.4 Preliminaries . . . 67
- 4.5 Approximating the Support of Itemsets . . . 69
- 4.6 Approximate Triangle Counting . . . 74
- 4.7 Approximate #SAT . . . 81
- 4.8 Conclusion . . . 83

5 Reductions for Frequency-Based Data Mining Problems 85
5.1 Introduction . . . 85
5.2 Preliminaries . . . 88
5.3 Related Work . . . 91
5.4 Maximality-Preserving Reductions . . . 92
5.5 Constraining the Set of Patterns . . . 96
5.6 Algorithms and Experiments . . . 101
5.7 Conclusion . . . 104

6 Efficient Distributed Workload (Re-)Embedding 105
6.1 Introduction . . . 105
6.2 Model . . . 108
6.3 Online Partition for Two Servers . . . 111
6.4 Generalization to Many Servers . . . 122
6.5 Distributed and Fast Algorithms . . . 135
6.6 Lower Bounds . . . 142
6.7 Applications: Union–Find and Online k-Way Partitioning . . . 147
6.8 Related Work . . . 149
6.9 Conclusion . . . 151

7 Tight Bounds for Online Graph Partitioning 153
7.1 Introduction . . . 154
7.2 Preliminaries . . . 158
7.3 Algorithmic Framework . . . 160
7.4 Adjusting Schedules . . . 162
7.5 Analysis . . . 173
7.6 Randomized Algorithm . . . 182
7.7 Lower Bounds . . . 188
7.8 Omitted Proofs . . . 195
7.9 Conclusion . . . 198

Bibliography 199

CHAPTER 1

Introduction

In the last decade, the world has witnessed unprecedented progress in the fields of artificial intelligence, data mining and machine learning. This has lead to the immense growth of these areas, both in the industry as well as in academia. This development was enabled by specialized new hardware, the ever-larger availability of data and breakthroughs in the development of algorithms that *find and exploit patterns in the data*.

While these algorithms have been highly successful in practice, many of them are *heuristics*, i.e., they do not come with provable guarantees. As a result, even though we can see their success in real-world applications on an everyday basis, our theoretical understanding of many of these algorithms and the underlying computational problems remains limited. Indeed, one may say that the progress in the applied algorithm development has happened at such a rapid pace that the gap between what is practically possible and our theoretical understanding has widened.

Nevertheless, a solid theoretical understanding of practical methods is highly desirable: The rigorous mathematical analysis of an algorithm yields important insights into the algorithm's strengths and it also uncovers its limitations. Thus, formal guarantees can help us assess an algorithm's quality and they give us a better understanding of how much confidence we can have in its output. The latter is particularly important when the data contains sensitive attributes, such as race or gender, or when the algorithm's output is used to predict whether a certain patient has a disease or not.

The fact that only few practical algorithms have theoretical performance guarantees, is, however, not a coincidence. Many of the computational problems related to finding and exploiting patterns in data are NP-hard and often they are even NP-hard to approximate. Thus, it seems that the classic worst-case analysis approach is too pessimistic for these types of problems: While it is possible (and often relatively easy) to come up with artificial worst-case instances such that a given algorithm fails spectacularly, real-world data usually does not exhibit this malign behavior and in-

stead is much more good-natured. Therefore, to obtain algorithms with theoretical guarantees, we will have to go *beyond worst-case analysis*.

In this thesis, we follow the beyond worst-case analysis approach and present algorithms that *provably* find and exploit patterns in the data. We thereby narrow the gap between what is practically possible and our theoretical understanding.

The contributions of this thesis can be classified into the following categories:

1. *Provably Finding Patterns:* We provide algorithms that provably extract a set of planted clusters from random bipartite graphs. This problem has applications, for example, in the analysis of online shopping data, where one wishes to identify groups of products that are frequently bought together and groups of customers who purchase similar products. We present the first algorithm which can provably extract *tiny* planted clusters. This result provides a theoretical justification for the success of existing heuristic methods that are used in practice. We further present the first streaming algorithm which, after two passes over a bipartite graph, returns a set of planted clusters and which scales to much larger datasets than existing methods.
2. *Understanding the Complexity of Finding Patterns:* We study the computational complexity of frequently-used subroutines of data mining algorithms and provide new hardness results. We show that for approximately computing the number of triangles in a graph and for approximating the support of itemsets in transactional databases, existing random sampling algorithms cannot be significantly improved unless popular conjectures in computational complexity are false. For instance, our results imply that for approximating how many customers of an online shop bought a given set of products, a simple random sampling algorithm is almost optimal (unless a conjecture computational complexity is false). Furthermore, we present a hierarchy of the enumeration complexity of several maximal frequent pattern mining problems and we provide a condition under which this hierarchy collapses.
3. *Provably Exploiting Patterns:* We formulate a theoretical model that captures the structure of the patterns in real-world communication networks. We provide algorithms that reduce the network traffic by exploiting these patterns and we show that the competitive ratios obtained by our algorithms are asymptotically optimal. We further show that our model captures applications such as implementing distributed union–find data structures. Our algorithms rely on a new technique that combines efficient integer linear programming (ILP) with the manual maintenance of optimal ILP solutions.

To obtain our results, we make heavy use of beyond worst-case analysis. For instance, the algorithms of Point 1 are analyzed for random graphs instead of arbitrary graphs and when deriving the theoretical guarantees, we take into account the properties one would expect from real-world data, such as the sparsity of the graph and the size of the patterns. For the results of Point 3, we introduce a formal model for the patterns in a communication network and we provably obtain better efficiency than what is possible when no assumption on the patterns is made. The hardness results from Point 2 are derived using classical worst-case analysis, but

they may motivate to study these problems with beyond worst-case analysis in the future: Indeed, the established lower bounds are *tight*, i.e., the lower bounds show that the running times of the existing algorithms cannot be significantly improved under worst-case analysis. Hence, to obtain further progress, one has to take into account the properties of the data and further research in this direction may be worthwhile.

We will now describe each of the obtained results in more detail.

1.1 Provably Finding Patterns

A popular problem in knowledge discovery and unsupervised learning is to find patterns in bipartite graphs. This problem is also known as *biclustering*, *co-clustering* or *Boolean matrix factorization* and its study dates back to the 1970s [96]. The problem has received attention in diverse areas of computer science such as data mining [63, 142, 218], machine learning [131, 207, 219, 220], and bioinformatics [134].

In this problem, the input is a bipartite graph $G = (U \cup V, E)$ and the patterns are clusters $U_1, \dots, U_k \subseteq U$ and $V_1, \dots, V_k \subseteq V$ such that each *bicluster* (U_i, V_i) satisfies certain interestingness criteria. Popular criteria for interestingness include that the subgraphs induced by the biclusters (U_i, V_i) either form a biclique [82, 161] or contain "many" edges relative to the global density of the graph [172, 219] (see below for the concrete problem we study).

In applications, the two sides of the bipartite graph represent objects of different types and an edge indicates the interaction of the corresponding objects. For example, in online shopping data, the vertices on the left side U of G correspond to customers and the vertices on the right side V of G correspond to products. An edge (u, v) indicates that customer u purchased product v. Each bicluster (U_i, V_i) corresponds to a group of products V_i which are purchased by similar customers U_i.

Real-world datasets often exhibit two crucial properties: First, the degrees on one side of the graph are bounded and, second, the right-side clusters V_j are *tiny* compared to the size of V. Sticking with the above online shopping example, note that almost all customers purchase at most a few hundred different products and, thus, the degree of the vertices in U is bounded. Next, in experiments, it is often observed that the product clusters V_j usually consist of at most a few dozen products. For instance, typical product clusters are the seven Harry Potter books or the 23 films from the Marvel Cinematic Universe. Therefore, the clusters V_j are much smaller than V, which often consists of millions of products.

In this line of work, two of the major challenges were as follows: (1) While practical algorithms have been known to find tiny clusters V_j highly successfully, the best theoretical results could only guarantee to recover medium-sized clusters (see below for details). Can we bridge this gap? (2) Existing practical algorithms deliver excellent results for graphs with thousands of vertices, but they do not scale to graphs with hundreds of thousands or even millions of vertices. Can we obtain more efficient algorithms by exploiting the sparsity of the data?

We answer both questions affirmatively.

The Case for Random Graph Models. The fact that previous algorithms were mostly heuristics without provable guarantees is not coincidental. Indeed, under *worst-case inputs* most problems for finding patterns in bipartite graphs are NP-hard [161] and there are strong lower bounds even for approximation algorithms [52].

To bypass these hardness results, we consider a standard random graph model (see below for the formal definition). This model has been popular in different communities, ranging from machine learning over theoretical computer science to mathematics and physics [1]. The model has the nice property that we can obtain polynomial-time algorithms which guarantee to return "the right answer", i.e., they guarantee to find the set of planted ground-truth clusters $U_1, \dots, U_k$ and $V_1, \dots, V_k$. Furthermore, several practical algorithms (without theoretical guarantees) were derived under similar random graph models, e.g., [172, 180], and these methods provide excellent results on real-world data. This suggests that the assumptions of the random graph model are quite realistic.

More concretely, the random graph model is formally defined as follows. Let $G = (U \cup V, E)$ be a bipartite graph with k *planted* clusters $U_1, \dots, U_k \subseteq U$ and $V_1, \dots, V_k \subseteq V$ on each side of the graph. Further, let $p, q \in [0, 1]$ be probabilities with $p > q$. We assume that edges (u, v) between vertices $u \in U_i$ and $v \in V_i$ are inserted with probability p and that edges (u, v) with $u \in U_i$ and $v \in V_j$ with $i \neq j$ are inserted with probability q. Thus, there are "relatively many" edges between the vertices of each bicluster (U_i, V_i) compared to the global density of the graph, where there are "relatively few" edges. Now the computational problem is as follows: Given a random graph generated from the above distribution, recover the planted clusters $U_1, \dots, U_k$ and $V_1, \dots, V_k$.

Finding Tiny Clusters. As we argued before, in real-world data it is natural to assume that the right-side clusters V_j are tiny. While practical algorithms have been highly successful at identifying these clusters, existing algorithms with theoretical guarantees [131, 138, 207] were only able to recover right-side clusters of size $|V_j| = \Omega(\sqrt{n})$, where $n = |V|$. This is unrealistic in the practical scenario described above (for example, if $|V| \geq 10^6$ then $\sqrt{|V|} \geq 10^3$). Hence, there is a stark contrast between the practical results and their theoretical justification.

In Chapter 2, we close this gap between theory and practice by providing a practical algorithm which provably recovers tiny clusters. More concretely, we show that for any $\varepsilon > 0$, one can recover clusters V_j of size $|V_j| = O(n^\varepsilon)$ if some conditions on p, q and the sizes of the left-side clusters U_i hold (see Theorem 2.1 in Section 2.1 for details); previous algorithms only allowed $\varepsilon \geq \frac{1}{2}$. The algorithm's guarantees hold even when the graph is extremely sparse and the degree of each vertex is only polylogarithmic in the total number of vertices.

Furthermore, the experiments show that on synthetic datasets the algorithm outperforms practical heuristic methods and that on real-world datasets it finds tiny clusters of high quality. Thus, the algorithm combines theoretical guarantees with state-of-the-art practical performance.

Efficient Streaming Algorithms. A second important question in the study of algorithms for finding patterns in bipartite graphs is to make existing methods more scalable.

In Chapter 3, we study the above problem in the streaming setting, where the input is a stream of left-side vertices $u \in U$ together with all of their incident edges. The goal is to provide an algorithm which only performs a few passes over the stream and uses little memory.

We present the first algorithm which after a *single* pass over the stream returns the right-side clusters V_i. The algorithm is extremely memory-efficient and its space usage is only a $O(\log m)$ factor higher than simply storing a single vertex for each cluster. Specifically, its space usage is $O(ks \log m)$, where $m = |U|$ is the number of points in the stream, k is the number of clusters and s is an upper bound on the degrees of the left-side vertices (recall that we argued above that s is small in real-world datasets). For a version of the algorithm, we prove that it finds the planted clusters in the random graph model using information-theoretically optimal space.

We further show that after a second pass over the stream, the algorithm can recover the left-side clusters. We also provide an extension of the algorithm to compute Boolean matrix factorizations (BMF), which is a popular problem in data mining [104, 133, 142] and machine learning [172, 180]. This provides the first streaming algorithm for BMF in the literature.

In experiments on real-world datasets, the algorithm is orders of magnitude faster and more memory-efficient than a static baseline algorithm. Specifically, the algorithm never uses more than 500 MB RAM on any dataset, even when the graphs contain millions of vertices and edges, and it has the desirable property that its running time scales linearly in the number of edges of the graph. The quality of the clusterings returned by the algorithm is within a factor 2 of the static baseline.

1.2 Understanding the Complexity of Finding Patterns

Next, we develop a better understanding of the computational complexity of fundamental problems related to the task of finding patterns in data.

Complexity of Approximating Pattern Frequencies. Some of the most heavily used subroutines in data mining algorithms are for computing the supports of patterns in the data. For example, when the data is a graph G and the pattern is a triangle, the goal would be to count the number of triangles $\#\mathsf{Triangle}(G)$ in G. In transactional databases, the goal would be to compute the support of an itemset. That is, let $[d] = \{1, \dots, d\}$ be a set of items, then an *itemset* is a subset of $[d]$. An $m \times d$ *transactional database* $\mathcal{D}$ is a (multi-)set $\mathcal{D} = \{T_1, \dots, T_m\}$, where each T_i is an itemset over $[d]$. Now the support $\#\mathsf{supp}(T)$ of an itemset T is the number of *transactions* $T_i \in \mathcal{D}$ such that $T \subseteq T_i$, i.e., $\#\mathsf{supp}(T) = |\{i : T \subseteq T_i\}|$. As an application, suppose the items $[d]$ are the products of an online shop, each transaction corresponds to the products bought by a customer and $\#\mathsf{supp}(T)$ counts the number of customers who bought the products in T.

Despite the prevalent tasks of computing $\#\mathsf{supp}(T)$ and $\#\mathsf{Triangle}(G)$, subroutines for computing these quantities are still considered slow in practice and, indeed, $\#\mathsf{supp}(T)$ is usually implemented using exhaustive search over the database. Thus, if we could speed up the subroutines for computing $\#\mathsf{supp}(T)$ or $\#\mathsf{Triangle}(G)$, then many practical algorithms could be made faster.

Unfortunately, existing complexity conjectures imply that practical algorithms cannot compute the *exact* values of $\#\mathsf{supp}(T)$ or $\#\mathsf{Triangle}(G)$ faster than using exhaustive search. For example, Williams [202] showed that under a popular hypothesis in complexity theory, the exact computation of $\#\mathsf{supp}(T)$ cannot be done much faster than simply reading the whole database $\mathcal{D}$.

Since computing $\#\mathsf{supp}(T)$ and $\#\mathsf{Triangle}(G)$ exactly is slow, some algorithms resort to computing these quantities *approximately* to obtain faster running times [130, 136, 174, 175, 194]. Now it is a natural question to ask how fast this approximate computation can be done.

In Chapter 4, we study the fine-grained complexity of *approximating* $\#\mathsf{supp}(T)$ and $\#\mathsf{Triangle}(G)$. Fine-grained complexity [204] is a relatively new area in theoretical computer science that aims at providing tight running time lower bounds for computational problems. These lower bounds are obtained under hypotheses which are stronger than the standard $\mathsf{P} \neq \mathsf{NP}$ assumption.

In particular, we consider *gap versions* of $\#\mathsf{supp}(T)$ and $\#\mathsf{Triangle}(G)$. That is, for a given threshold K, the task is to decide whether $\#\mathsf{supp}(T) \geq K$ or $\#\mathsf{supp}(T) \leq K/3$. If $K/3 < \#\mathsf{supp}(T) < K$, the algorithm may return any answer. This problem has already been studied in several previous works [130, 136, 174, 175, 194] from an upper bound perspective because efficient subroutines for this question can be used to speed up existing algorithms.

One of the results in Chapter 4 is as follows. Suppose that we want to solve the gap version of computing $\#\mathsf{supp}(T)$ for a transactional database with m transactions and suppose K is parameterized as $K = m^\gamma$ for $\gamma > 0$. The problem can be solved by a simple randomized algorithm which samples $O(m^{1-\gamma} \log m)$ transactions and then computes the support of the itemset T on this sample. We show that this algorithm cannot be significantly improved. More concretely, we show that under a popular complexity conjecture, the gap version of computing $\#\mathsf{supp}(T)$ cannot be solved faster than in time[1] $m^{1-\gamma-o(1)}$ (see Theorem 4.4 for details). Thus, we settle the complexity of solving the gap version of $\#\mathsf{supp}(T)$.

We further provide matching upper and lower bounds for solving the gap version of $\#\mathsf{Triangle}(G)$. More concretely, we show that if a popular computational complexity hypothesis holds, then any algorithm, that decides whether an n-vertex graph satisfies $\#\mathsf{Triangle}(G) \geq n^\gamma$ or $\#\mathsf{Triangle}(G) \leq n^\gamma/3$ and does not use fast matrix multiplication, requires time at least $n^{3-\gamma-o(1)}$. This lower bound is again matched by a simple random sampling algorithm. See Theorem 4.8 for the formal

[1] When a problem "cannot be solved faster than time $T^{\alpha-o(1)}$", this is equivalent to the following statement: "For all $\varepsilon > 0$, there is no algorithm with running time $T^{\alpha-\varepsilon}$." Hence, there is no algorithm with run-time polynomially faster than $O(T^\alpha)$.

statement. Moreover, we show that our lower bounds for approximate triangle counting imply lower bounds for approximately computing important graph measures in social network analysis, such as approximating the clustering coefficient or the transitivity of the graph.

We also present matching upper and lower bounds for deciding whether a SAT formula has at least $2^{\gamma n}$ satisyfing assignments or none at all. In particular, we show that if a popular complexity conjecture is true, then this problem cannot be solved faster than in time $2^{(1-\gamma-o(1))n}$.

Enumerating Patterns. Another important subroutine in data mining algorithms is to find all maximal frequent patterns. That is, for a threshold K and a transactional database $\mathcal{D}$, an itemset T is *frequent* for $\mathcal{D}$ if $\#\mathsf{supp}(T) \geq K$ and, otherwise, T is *infrequent*. Furthermore, T is *maximal frequent* if for all itemsets T' with $T \subsetneq T'$ it holds that T' is infrequent. This problem has also been studied for other types of data, where the databases consist of graphs or sequences.

In maximal frequent pattern mining, the task is to compute and output all maximal frequent patterns. Since there can potentially be exponentially many maximal frequent patterns, it is suitable to measure the complexity of this problem not only in the size of its input but in the size of the input *and output* [117]. This then leads to enumeration and extendability problems. For example, a typical problem would be to decide whether a set of patterns can be extended: given a (set of) maximal frequent patterns, compute another maximal frequent pattern which is not contained in the given set or return that no such pattern exists. Such problems are studied under the term *enumeration complexity* [113].

In Chapter 5, we develop a *hierarchy* of the enumeration complexities of maximal frequent pattern mining problems over different types of data. The data types we consider include databases of transactions, sequences and different graph classes (ranging from trees to general graphs). We also show that if we slightly generalize the problem and allow to constrain the set of feasible patterns, then the hierarchy collapses and all problems obey the same enumeration complexities.

1.3 Provably Exploiting Patterns

The final two chapters of this thesis are devoted to *exploiting* patterns in data and we study algorithms for serving communication requests in distributed systems. Indeed, it can be empirically shown that the communication requests in such systems often include patterns and that datacenters that adapt to these patterns have a lower cost than demand-oblivious datacenters with the same performance [83, 91].

In Chapter 6, we introduce the following formal model which captures the structure of the patterns in real-world datacenters. Suppose the datacenter contains ℓ servers and n workloads. Initially, each workload is assigned to one of the servers and each server has enough capacity to store $(1+\varepsilon)k$ workloads, where $k = n/\ell$ and $\varepsilon > 0$ is a constant. Now the workloads start communicating and at each time step,

two workloads communicate with one another. This sequence of communication requests is presented to an algorithm in an online manner.

Our assumption for the communication requests is that they *induce small patterns*. More formally, consider the workloads as vertices of a graph and the communication requests as edges. Then our model assumes that after observing all communication requests, all connected components in the resulting graph contain at most k vertices, i.e., each of the patterns can be fit on a single server. These connected components are the *patterns* of the workloads. Furthermore, we assume that when the communication requests finish, all workloads from the same pattern must be assigned to the same server.

The cost paid for communication requests and relocating workloads are as follows. If the communicating workloads are assigned to the same server, this request has cost 0, and, otherwise, the cost of the request is 1. After each time step, an algorithm can decide to *relocate* workloads between the servers while maintaining the capacity constraints of the servers; for each relocation, the algorithm has to pay $\alpha > 1$. We analyze the algorithm using *competitive analysis*, i.e., we divide the cost paid by an online algorithm ONL by the cost of the optimal offline algorithm OPT which knows all communication requests in advance.

In Chapter 6, we provide a *deterministic* algorithm with a competitive ratio of $O((\ell \log \ell \log k)/\varepsilon)$. Note that for $\ell = O(1)$ servers, the algorithm is $O((\log k)/\varepsilon)$-competitive ($\ell = O(1)$ could be the case, e.g., when the servers correspond to a few large datacenters which are distributed around the globe). Without the above patterns model, the best possible algorithm would have a competitive ratio of $\Omega(k)$ for $\ell = O(1)$ [20, 23]. Therefore, by exploiting the above pattern model, we have obtained an exponential improvement over the worst-case analysis approach.

Furthermore, for the algorithm in Chapter 6 we show how it can be implemented in a distributed setting with little communication overhead and we believe that (with some adjustments) it can be implemented in practice.

As an application of our pattern model, we show that it can be used to implement distributed union–find data structures. Thus, as a corollary, we obtain competitive algorithms for running union–find data structures with almost optimal network communication.

In Chapter 7, we provide asymptotic improvements upon the algorithm from Chapter 6. More concretely, the main result of this chapter is a *randomized* algorithm with competitive ratio $O_\varepsilon(\log \ell + \log k)$. Here, the $O_\varepsilon(\cdot)$-notation hides terms which only depend on ε. We also provide a matching lower bound of $\Omega(\log \ell + \log k)$ for randomized algorithms. Therefore, our results are optimal when ignoring constant factors and terms which only depend on ε.

Furthermore, we also provide a *deterministic* algorithm with competitive ratio $O_\varepsilon(\ell \log k)$ and a lower bound of $\Omega(\ell \log k)$. The results are again optimal up to constant factors and terms only depending on ε.

To obtain the results of Chapter 7, we introduce a novel technique which combines efficient integer linear programming (ILP) with the manual maintenance of optimal ILP solutions. More precisely, when new communication requests arrive,

we use an ILP to assign the workload patterns to the servers; this is similar to existing approximation schemes for scheduling algorithms. However, we cannot obtain our competitive ratios if we run the ILP after each communication request. Instead, we identify certain types of communication requests, after which we can manually obtain an optimal ILP solution at zero cost without resolving the ILP.

Let us briefly compare the results of Chapters 6 and 7. The results of Chapter 7 are asymptotically tight w.r.t. the parameters ℓ and k, but they come with a super-polynomial dependency on ε. The algorithm of Chapter 6 has asymptotically worse guarantees w.r.t. ℓ and k, but its competitive ratio only polynomially depends on ε. Thus, the results of the chapters are not fully comparable. From a pratical point of view, the algorithm of Chapter 6 is much simpler than the one of Chapter 7 and appears to be more likely to deliver good results in applications.

CHAPTER 2

Bipartite Stochastic Block Models with Tiny Clusters

We study the problem of finding clusters in random bipartite graphs. We present a simple two-step algorithm which provably finds even tiny clusters of size $O(n^{\varepsilon})$, where n is the number of vertices in the graph and $\varepsilon > 0$. Previous algorithms were only able to identify clusters of size $\Omega(\sqrt{n})$. We evaluate the algorithm on synthetic and on real-world data; the experiments show that the algorithm can find extremely small clusters even in presence of high destructive noise.

2.1 Introduction

Finding clusters in bipartite graphs is a fundamental problem and has many applications. In practice, the two parts of the bipartite graph usually correspond to objects from different domains and an edge corresponds to an interaction between the objects. For example, paleontologists use biclustering to find co-occurrences of localities (left side of the graph) and mammals (right side of the graph) [75]; bioinformaticians want to relate biological samples and gene expression levels [67]; in an online shop setting, one wants to find clusters of customers and products.

Discovering clusters in bipartite graphs has been researched in many different settings. However, most of these algorithms were heuristics and do not provide theoretical guarantees for the quality of their results. This was recently addressed by Xu et al. [207] and Lim et al. [131] who initiated the study of biclustering algorithms with formal guarantees. They considered random bipartite graphs and proved under which conditions their algorithms can recover the ground-truth clusters.

In this chapter, we consider a standard random graph model and propose a simple two-step algorithm which provably discovers the ground-truth clusters in bipartite graphs: (1) Cluster the vertices on the left side of the graph based on the

similarity of their neighborhoods (Section 2.3). (2) Infer the right side clusters based on the previously discovered left clusters using degree-thresholding (Section 2.4).

Our algorithm allows to recover even tiny clusters of size $O(n^\varepsilon)$, where n is the number of vertices on the right side of the graph and $\varepsilon > 0$. Previously, existing algorithms could only discover clusters of size $\Omega(\sqrt{n})$. Note that finding tiny clusters is of high practical importance. For example, in an online shop with millions of products ($n \geq 10^6$), finding only clusters of at least a thousand products ($\sqrt{n} \geq 10^3$) is not very interesting. One would want the product clusters to be much smaller.

The formal guarantees of our algorithm are provided at the end of this section. From a high-level point of view, the algorithm can be seen as a way to leverage formal guarantees for mixture models and clustering algorithms into biclustering algorithms with formal guarantees. This partially explains why heuristics such as "apply k-means to both sides of the graph" are very successful in practice.

Finally, we implement a version of the proposed algorithm (Section 2.5) and we evaluate it on synthetic and on real-world data. The experiments show that in practice the algorithm can find extremely small clusters and it outperforms all algorithms we compare with (Section 2.6).

Bipartite Stochastic Block Models. We now introduce *bipartite stochastic block models (SBMs)* which we will be using throughout the chapter. Let $G = (U \cup V, E)$ be a bipartite graph with m vertices in U and n vertices in V; we call U the *left side* of G and V the *right side* of G.

The left side U is partitioned into clusters $U_1, \dots, U_k$, i.e., $U_i \cap U_j = \emptyset$ for $i \neq j$ and $\bigcup_i U_i = U$. For V there are clusters $V_1, \dots, V_k$ with $V_i \subseteq V$; we do not assume that the V_j are disjoint or that their union equals V. The U_i are the *left clusters* of G and the V_j are the *right clusters* of G.

Fix two probabilities $p > q \geq 0$. For any two vertices $u \in U_i$ and $v \in V_i$, insert an edge with probability p; for $u \in U_i$ and $v \notin V_i$, insert an edge with probability q.

The algorithmic task for bipartite SBMs is as follows. Given parameters k, p, q and a graph G generated in the previously described way, recover all clusters U_i and V_j.

Main Theoretical Results. We propose the following simple algorithm:

1. Recover the clusters U_i by clustering the vertices in U according to the similarity of their neighborhoods (see Section 2.3).
2. For each recovered U_i, set V_i to all vertices with "many" neighbors in U_i (see Section 2.4).

To state the formal guarantees of the proposed algorithm we require two parameters. We let ℓ be the size of the smallest cluster on the left side, i.e., $\ell = \min_i |U_i|$. Furthermore, let δ denote the smallest difference between any two clusters on the

right side; more formally, $\delta = \min_{i \neq j} |V_i \triangle V_j|$, where $V_i \triangle V_j = (V_i \backslash V_j) \cup (V_j \backslash V_i)$ is the symmetric difference of V_i and V_j.

We now state the main result of this chapter. In the theorem, $D(p \mid\mid q)$ denotes the Kullback–Leibler divergence of Bernoulli random variables with parameters $p, q \in [0, 1]$, i.e., $D(p \mid\mid q) = p \log(\frac{p}{q}) + (1-p) \log(\frac{1-p}{1-q})$.

Theorem 2.1. *Suppose $\sigma^2 = \max\{p(1-p), q(1-q)\} \geq (\log^6 n)/n$. There exist constants C_1, C_2 such that if $\ell \geq (C_1 \log n)/D(p \mid\mid q)$ and*

$$\frac{(p-q)^2}{\sigma^2} > C_2 k \frac{n + m \log m}{\ell \delta}, \tag{2.1}$$

then there exists an algorithm which on input G, k, p and q returns all clusters U_i and V_i. The algorithm succeeds with high probability.

To give a better interpretability of the theorem, consider its two main assumptions: (1) The condition $\ell \geq C_1 \log n / D(p \mid\mid q)$ is necessary so that the vertices in V_i have sufficiently many neighbors in U_i. (2) To get a better understanding of Equation (2.1), consider the case where $m = \Theta(n)$, $k = O(1)$, and p, q are constants. Also, ignore logarithmic factors. We obtain a smooth tradeoff between δ and ℓ: The inequality in Equation (2.1) is satisfied if $\delta = \Theta(n^{\varepsilon})$ and $\ell = \Theta(n^{1-\varepsilon})$. That is, if the right clusters are very small or similar (δ is small), the algorithm requires larger clusters on the left side (ℓ must be large). On the other hand, if the right clusters are very large and dissimilar (large δ), the algorithm requires only very small left clusters (small ℓ suffices). More generally, if $p - q = \Theta(n^{-C})$ and $p \gg q$, the algorithm requires $\ell\delta = \Theta(n^{1+C})$.

The fact that the algorithm can recover clusters of size $O(n^{\varepsilon})$ is interesting since previous algorithms required $\min\{\ell, \delta\} = \Omega(\sqrt{n})$ (see Section 2.2). Furthermore, the lower bounds of Hajek, Wu and Xu [90] show that breaking the $\Omega(\sqrt{n})$ barrier is *impossible* in general graphs. Hajek et al. also provide lower bounds in the bipartite setting which show that one cannot find biclusters of size $k \times k$ for $k = o(\sqrt{n})$. We bypass this lower bound through the previously discussed smooth tradeoff between ℓ and δ. We conjecture that the tradeoff we obtain is asymptotically optimal.

We also study the setting in which the algorithm only obtains an *approximate* clustering of the left side of the graph (Section 2.4.2). In this setting, we show that if the approximation of the left clusters is of good enough quality, then the right clusters can still be recovered *exactly*. We also observe this behavior in our experiments in Section 2.6.

Experimental Evaluation. We implemented a version of the algorithm from Theorem 2.1 and present the practical details in Section 2.5. The experimental results are reported in Section 2.6. In the experiments, our main focus will be to verify whether, in practice, the algorithm can find the small clusters that the theoretical analysis promised.

On synthetic data, the experiments show that, indeed, the algorithm finds tiny clusters even in the presence of high destructive noise and it outperforms all methods that we compare against.

The algorithm is also qualitatively evaluated on real-world datasets. On these datasets it finds clusters which are interesting and which have natural interpretations.

2.2 Related Work

Stochastic Block Models (SBMs). During the last years, many papers on SBMs have been published. We only discuss bipartite SBMs here and refer to the survey by Abbe [1] for other settings.

Lim, Chen and Xu [131] study the biclustering of observations with general labels. When constrained to only two labels, their results provide a bipartite SBM. However, in the bipartite SBM case, [131] has two drawbacks compared to the results presented here: (1) The data-generating process in [131] rules out certain nested structures of the sets V_i. E.g., [131] does not allow to have clusters V_1, V_2, V_3 such that $V_3 = V_1 \cup V_2$. (2) The main result of [131] relies on a notion of *coherence*, which measures how difficult the structure of the clusters is to infer. Due to this dependency on coherence, the results of this chapter and [131] are only partially comparable. In case of a constant number of clusters or "worst-case coherence", though, the algorithm of [131] only works if both ℓ and δ are $\Omega(\sqrt{n})$.

Zhou and Amini [219] study spectral methods for bipartite SBMs. [219] considers a more general connectivity structure and obtains sharper bounds for the recovery rates than in this chapter. However, in [219] the clusters V_i cannot overlap and, hence, the results of this chapter and [219] are incomparable.

Abbe and Sandon [2, 3] and Gao et al. [79] study optimal recovery for SBMs in general graphs. Their results apply to bipartite graphs with a constant number of overlapping communities of linear size. Zhou and Amini [220] improve these results for bipartite SBMs under a broader range of parameters.

One can use the result of McSherry [138] to recover the clusters of a bipartite graph but this has two caveats: (1) It does not allow the V_i to overlap. (2) Both ℓ and δ must be of size $\Omega(\sqrt{n})$.

Florescu and Perkins [73] provided an SBM for bipartite graphs with two linear-size communities on each side of the graph. Xu et al. [207] consider a biclustering setting with clusters of size $\Omega(n)$.

Boolean Matrix Factorization (BMF). Another way to find clusters in bipartite graphs is BMF. BMF takes the biadjacency matrix $D \in \{0,1\}^{m \times n}$ of a bipartite graph and finds factor matrices $L \in \{0,1\}^{m \times k}$ and $R \in \{0,1\}^{k \times n}$ such that $D \approx L \circ R$, where $\circ$ is the Boolean matrix-matrix-product. In other words, BMF tries to approximate D with a Boolean-rank k matrix. The interpretation is that the columns of L contain the left clusters and the rows of R contain the right clusters.

This setting is more general than the one presented in this chapter as it allows the clusters U_i to overlap.

BMF was studied from applied [142, 144, 172, 180, 181] and also from theoretical [28, 52, 74] perspectives. Section 2.6 provides an experimental comparison of BMF algorithms and the algorithm from this chapter.

2.3 Recovering the Left Clusters

We describe how the clusters U_i can be recovered. Our approach is to cluster the vertices $u \in U$ according to the similarity of their neighborhoods in V. The intuition is that if two vertices u and u' are in the same cluster U_i, they should have relatively many neighbors in common (those in V_i). On the other hand, if u and u' are from different clusters U_i and U_j, their neighbors should be relatively different (as $V_i \triangle V_j$ is supposed to be large).

Technically, we will apply mixture models. We use the result by Mitra [147] since it is simple to state. We could as well use other mixture models such as the one by Dasgupta et al. [60] or clustering algorithms such as Kumar and Kannan [121], Bilu and Linial [34] or Cohen-Addad and Schwiegelshohn [56]. The different methods come with different assumptions on the data.

2.3.1 Mixture Models and Mitra's Algorithm

Mixture Models on the Hypercube. From a high-level point of view, the question of mixture models is as follows: Given samples from different distributions, cluster the samples according to which distributions they were sampled from. We will now present the formal details behind this.

Let there be k probability distributions $D_1, \dots, D_k$ in $\{0,1\}^n$ and denote the mean of D_r as $\mu_r \in [0,1]^n$. Let σ^2 be an entry-wise upper bound on all μ_r, i.e., $\mu_r(i) \leq \sigma^2$ for all $r = 1, \dots, k$ and $i = 1, \dots, n$. For each distribution D_r define a weight $w_r > 0$ such that $\sum_r w_r = 1$.

From each distribution D_r, create $w_r m$ samples and denote the set of these samples as T_r. In total we obtain m samples and denote the set containing all samples as T, i.e., $T = \bigcup_r T_r$.

The algorithmic problem in mixture models is as follows. Given T and k, find a partition $P_1, \dots, P_k$ of the samples in T such that $\{T_1, \dots, T_k\} = \{P_1, \dots, P_k\}$.

Mitra's Algorithm. Mitra [147] provided an algorithm for solving the mixture models problem. To state its guarantees, we define a matrix $A \in \{0,1\}^{m \times n}$ which has the samples from T in its rows. Thus, by clustering the rows of A, we obtain a clustering of T. The following lemma gives a condition under which Mitra's algorithm returns the correct clustering. In the lemma, we write $||v||_2 = (\sum_i v_i^2)^{1/2}$.

Lemma 2.2 (Mitra [147]). *Suppose $\sigma^2 \geq \log^6 n/n$. Let $\zeta = \min\{||\mu_r - \mu_s||_2^2 : r \neq s\}$ and $w_{\min} = \min_r w_r$. Then there exists a constant c such that if*

$$\zeta > ck\sigma^2 \frac{1}{w_{\min}} \left(\frac{m+n}{m} + \log m \right),$$

then on input A and k, the output $\{P_1, \ldots, P_k\}$ of Mitra's algorithm satisfies that $\{P_1, \ldots, P_k\} = \{T_1, \ldots, T_k\}$ with high probability. That is, the algorithm recovers the clusters T_r.

2.3.2 Proposition and Analysis

Let us come back to our original problem of recovering the left clusters of G. To find the left clusters U_i, we apply Mitra's algorithm to the rows of the biadjacency matrix D of G. Formally, the biadjacency matrix $D \in \{0,1\}^{m \times n}$ of G is the matrix with $D_{uv} = 1$ iff there exists an edge $(u, v) \in G$.

Proposition 2.3 states under which conditions this approach succeeds.

Proposition 2.3. *Let all variables be as in Section 2.1. Let $\delta = \min_{i \neq j} |V_i \triangle V_j|$ and $\ell = \min_i |U_i|$. Suppose $\sigma^2 = \max\{p(1-p), q(1-q)\} \geq \log^6 n/n$. There exists a constant C such that if*

$$\frac{(p-q)^2}{\sigma^2} > Ck \frac{n + m \log m}{\ell \delta}, \tag{2.2}$$

then applying Mitra's algorithm on D returns a partition $\{\tilde{U}_1, \ldots, \tilde{U}_r\}$ of D's rows such that $\{\tilde{U}_1, \ldots, \tilde{U}_r\} = \{U_1, \ldots, U_r\}$ with high probability. That is, the algorithm recovers the left clusters U_i of G.

Proof. Observe that D is a matrix arising from a mixture model as discussed earlier: Consider a vertex $u \in U_i$ and its corresponding row D_u in D. Then the probability that entry $D_{uv} = 1$ is p if $v \in V_i$ and q if $v \notin V_i$. Furthermore, for two vertices $u, u' \in U_i$ these distributions are exactly the same.

Hence, we view the rows of D as samples from k distributions D_i with distribution D_i corresponding to cluster U_i. For each cluster U_i, we have $|U_i|$ samples from D_i. For the mean μ_i of D_i, we have component-wise $\mu_i(v) = p$, if $v \in V_i$, and $\mu_i(v) = q$, if $v \notin V_i$. Thus, partitioning the rows of D with a mixture model is exactly the same as recovering the clusters U_i of G.

It is left to check that the conditions of Lemma 2.2 are satisfied. By assumption on the V_j, $\left|\left|\mu_i - \mu_j\right|\right|_2^2 \geq (p-q)^2\delta$ for $i \neq j$. Since we have $|U_i|$ samples from distribution D_i, the mixing weights are $w_i = |U_i|/m$ and $w_{\min} = \ell/m$. To apply Lemma 2.2, we must satisfy the inequality

$$(p-q)^2\delta > ck\sigma^2 \frac{m}{\ell} \left(\frac{m+n+m\log m}{m} \right) = ck\sigma^2 \left(\frac{m+n+m\log m}{\ell} \right).$$

By rearranging terms and noticing that $Cm \log m \geq c(m + m \log m)$ for large enough C, this is the inequality we required in the proposition (Equation (2.2)). □

2.4 Recovering the Right Clusters

This section present three algorithms to recover the right clusters V_j given the left clusters U_i. The first two algorithms are very simple and have provable guarantees, but they require knowledge about the parameters p and q. The third algorithm is a heuristic, which tries to estimate the correct values for p and q.

2.4.1 Exact Left and Right Clusters

First, we present an algorithm for which we prove that it recovers the right-side clusters V_i exactly when it is given the exact left-side clusters U_i. The algorithm is very simple: For each given cluster U_i, $\tilde{V}_i$ consists of all vertices from V which have "many" neighbors in U_i. We will show that the algorithm succeeds with high probability. We also prove Theorem 2.1 at the end of the subsection.

High-Degree Thresholding Algorithm. The input for the algorithm are p, q and the clusters $U_1, \dots, U_k$. For each cluster U_i, the algorithm constructs $\tilde{V}_i$ by adding all vertices $v \in V$ which have at least $\theta|U_i|$ neighbors in U_i, where we set

$$\theta = \log\left(\frac{1-q}{1-p}\right)\left(\log\left(\frac{p(1-q)}{q(1-p)}\right)\right)^{-1}. \tag{2.3}$$

Proposition and Analysis. In Proposition 2.4, we show that for a fixed cluster U_i of sufficiently large size, $V_i = \tilde{V}_i$ with probability $1 - O(n^{-2})$. A union bound implies that $\tilde{V}_i = V_i$ for all $i = 1, \dots, k$ with high probability. In the proposition, we use the notation $D(p \mid\mid q)$ to denote the Kullback–Leibler divergence of Bernoulli random variables with parameters $p, q \in [0, 1]$, i.e., $D(p \mid\mid q) = p\log(\frac{p}{q}) + (1 - p)\log(\frac{1-p}{1-q})$.

Proposition 2.4. *There exists a constant C such that if $|U_i| \geq C \log n / D(p \mid\mid q)$, then $\tilde{V}_i$ returned by the high-degree thresholding algorithm satisfies $V_i = \tilde{V}_i$ with probability at least $1 - O(1/n^2)$. The algorithm runs in time $O(|U_i|n)$.*

Proof. Consider a vertex $v \in V$. The vertex v has an edge to $u \in U_i$ with probability p, if $v \in V_i$, and with probability q, if $v \notin V_i$. Let Z_v be the random variable denoting the number of edges from v to vertices in U_i; Z_v is binomially distributed with $|U_i|$ trials and success probability p (if $v \in V_i$) or q (if $v \notin V_i$). To find out whether $v \in V_i$, we must decide whether Z_v is distributed with parameter p or q. If we decide for the correct parameter then the decision to include v into $\tilde{V}_i$ is correct.

We make the decision for the parameter based on the likelihood of observing Z_v edges incident upon v. Parameter p is more likely if:

$$\frac{\binom{|U_i|}{Z_v} p^{Z_v}(1-p)^{|U_i|-Z_v}}{\binom{|U_i|}{Z_v} q^{Z_v}(1-q)^{|U_i|-Z_v}} = \left(\frac{p}{q}\right)^{Z_v}\left(\frac{1-p}{1-q}\right)^{|U_i|-Z_v} \geq 1.$$

Solving this inequality for Z_v gives that one should decide for parameter p if $Z_v \geq \theta|U_i|$, where θ is as in Equation (2.3).

The maximum likelihood approach above succeeds with probability at least $1 - O(1/n^3)$; this follows from [27, Theorem 6] if $|U_i| \geq C \log n/D(p \mid\mid q)$, where C is a sufficiently large constant. The probability for obtaining a correct result for *all* vertices $v \in V$ is at least $1 - O(1/n^2)$; this follows from a union bound. Conditioning on this event we obtain $V_i = \tilde{V}_i$. □

Given Proposition 2.4, we can now prove Theorem 2.1.

Proof of Theorem 2.1. By Proposition 3, the clusters U_i can be recovered with high probability. By Proposition 4 and the statement before the proposition, all V_i can be recovered with high probability given the correct U_i. Now a union bound implies that both events happen simultaneously with high probability. Furthermore, the conditions of the propositions are satisfied because they are the same as those of Theorem 1. □

2.4.2 Approximate Left and Exact Right Clusters

Next, we show that given a good enough *approximate* clustering of the left side of the graph, the clusters on the right side of the graph can still be recovered *exactly*. We obtain this result using the same high-degree thresholding algorithm as in Section 2.4.1, but we change its analysis; the new proof does not require the exact left clusters as input, but this comes with slightly stronger assumptions on the sizes of the clusters U_i.

The results from of subsection explain a phenomenon from the experiments on synthetic data (Section 2.6.2): Even though the recovery of the left-side clusters was not perfect, the algorithm from Section 2.4.1 still returned the exact right-side clusters in many cases.

To simplify our analysis, we assume that each cluster U_i contains exactly ℓ vertices. Thus, the left side of the graph contains $m = k\ell$ vertices. Furthermore, we assume that $q = Cp$ for $C = 1 - \Omega(1)$, i.e., p and q differ by at least a constant factor.

We will be working with the following definition of an approximate clustering. Let $U_1, \dots, U_k$ and $\tilde{U}_1, \dots, \tilde{U}_k$ be partitions of a set U of size $|U| = m$. Suppose that $U_1, \dots, U_k$ are the ground-truth clusters. We say that $\tilde{U}_1, \dots, \tilde{U}_k$ is an ε-*approximate clustering* of $U_1, \dots, U_k$ if there are at most εm misclassified elements. More formally, we assume that

$$\frac{1}{2} \sum_{i=1}^{k} |U_i \triangle \tilde{U}_i| \leq \varepsilon m = \varepsilon k \ell,$$

where $\triangle$ denotes the symmetric difference of two sets.

We now state our main result which shows that given an ε-approximate clustering of the left side of the graph, the clusters V_i can still be recovered exactly if ε is small enough.

Proposition 2.5. *Suppose $\tilde{U}_1, \dots, \tilde{U}_k$ is an ε-approximate clustering of $U_1, \dots, U_k$ and let $q = Cp$ for $C = 1 - \Omega(1)$. Then there exist constants D_1, D_2 such that if (1) $\ell \geq (D_1 \log n)/p$ and (2) $\varepsilon \leq D_2/k$, then there exists an algorithm which returns the clusters $V_1, \dots, V_k$ with high probability.*

For the proof, we need the following Chernoff bound (see, e.g., Theorem 1.1 in Dubhashi and Panconesi [64]).

Lemma 2.6. *Let $X_1, \dots, X_n$ be independent random variables in $[0, 1]$ and set $X = \sum_i X_i$. Then for $\varepsilon > 0$,*

$$\mathbf{Pr}\left(X > (1+\varepsilon)\mathbf{E}\left[X\right]\right) \leq \exp(-\varepsilon^2 \mathbf{E}\left[X\right]/3),$$
$$\mathbf{Pr}\left(X < (1-\varepsilon)\mathbf{E}\left[X\right]\right) \leq \exp(-\varepsilon^2 \mathbf{E}\left[X\right]/2).$$

Proof of Proposition 2.5. Let $\alpha < 1$, $\beta > 1$ be constants such that $\alpha/\beta > C$. We are going to reuse a version of the high-degree threshold algorithm from Section 2.4.1: For each i and $v \in V$, include v in V_i if v has at least $\alpha(1-\varepsilon k)p \cdot \ell$ neighbors in $\tilde{U}_i$. Otherwise, do not insert v into V_i.

Fix any $i \in \{1, \dots, k\}$. We show that given $\tilde{U}_i$, V_i can be recovered with probability at least $1 - O(n^{-2})$. Using a union bound over the k clusters, all V_i are recovered with probability at least $1 - O(n^{-1})$.

To show that V_i can be recovered from $\tilde{U}_i$ with the desired probability, we show that with probability at least $1 - O(n^{-3})$, it can be decided whether $v \in V_i$ or $v \notin V_i$ for each $v \in V$. A union bound implies that V_i is recovered exactly with probability at least $1 - O(n^{-2})$.

Observe that by the definition of an ε-approximate clustering,

$$\ell - \varepsilon k \ell \leq |\tilde{U}_i| \leq \ell + \varepsilon k \ell.$$

This implies that if $v \in V_i$, then there are at least $\ell - \epsilon k \ell$ vertices in $\tilde{U}_i$ to which v has an edge with probability p. Thus, the expected number of neighbors[1] of v in $\tilde{U}_i$ is at least

$$\mu_1 = (1 - \varepsilon k)\ell p.$$

On the other hand, if $v \notin V_i$, then the expected number of neighbors of v in $\tilde{U}_i$ is maximized when $\tilde{U}_i$ contains ℓ vertices to which v has an edge with probability q and $\varepsilon m = \varepsilon \ell k$ vertices to which v has an edge with probability p. Thus, if $v \notin V_i$, its expected number of neighbors in $\tilde{U}_i$ is at most

$$\mu_2 = \ell q + \varepsilon \ell k p = (C + \varepsilon k)\ell p.$$

[1] Note that if $v \in V_i$, $\tilde{U}_i$ might contain vertices to which v has edges with probability q. However, we can safely ignore these when computing the the lower bound on the expected number of neighbors of v in $\tilde{U}_i$.

Now setting D_1 to a large enough constant and applying Lemma 2.6, we obtain that the following event occurs with probability at least $1 - \exp(-3 \log n) = 1 - O(n^{-3})$: (1) If $v \in V_i$, then v has at least $\alpha\mu_1$ neighbors in $\tilde{U}_i$. (2) If $v \notin V_i$, then v has less than $\alpha\mu_1$ neighbors in $\tilde{U}_i$.

To see that (2) holds, it is left to show that the gap between μ_1 and μ_2 is sufficiently large. Note that the only interesting case is when μ_1 and μ_2 differ only by a constant factor. To show that (2) holds, we prove that $\alpha\mu_1 > \beta\mu_2$ for constants α and β as set at the beginning of the proof. We obtain that:

$$\begin{aligned} & \alpha\mu_1 > \beta\mu_2 \\ \iff \quad & \alpha(1-\varepsilon k)\ell p > \beta(C+\varepsilon k)\ell p \\ \iff \quad & \alpha(1-\varepsilon k) > \beta(C+\varepsilon k). \end{aligned}$$

Note that by condition on ε we obtain

$$1 - \varepsilon k \geq 1 - D_2$$

and

$$C + \varepsilon k \leq C + D_2.$$

Hence, the inequality $\alpha\mu_1 > \beta\mu_2$ holds if

$$\alpha(1 - D_2) > \beta(C + D_2) \qquad \iff \qquad D_2 \leq \frac{\alpha - \beta C}{\alpha + \beta}.$$

Observe that $\alpha - \beta C$ is always positive since we assumed that $\alpha/\beta > C$ (and, hence, $\alpha > \beta C$). We conclude that the inequality $\alpha\mu_1 > \beta\mu_2$ can be satisfied by setting D_2 to a sufficiently small constant. □

2.4.3 Heuristic for Estimating p and q

Note that both previously discussed algorithms for recovering the right-side clusters require knowledge of the parameters p and q, which is usually not available in practice. Hence, we now discuss a heuristic for estimating the parameters p and q.

Consider the high-degree thresholding algorithm from Section 2.4.1. Recall that for a given left-side cluster U_i, the algorithm sets V_i to the set of vertices containing all $v \in V$ with at least $\theta|U_i|$ neighbors in U_i. Here, computing $\theta = \theta(p, q)$ requires knowledge about p and q.

Now suppose we obtain a set V_i that was computed in the previously specified way for parameter $\theta = \theta(p, q)$.

By assumption of the SBM model, we know that in a case without noise (i.e., $p = 1, q = 0$), all vertices in V_i would have $|U_i|$ edges to vertices in U_i and 0 edges to vertices outside U_i (here, we assume that the V_i do not overlap). In the noiseless setting, there would be a total of $|V_i| \cdot |U_i|$ edges from vertices in V_i to vertices in U_i.

Input: G a bipartite $m \times n$ graph, k, p, q
1: **procedure** pcv(G, k, p, q)
2: $D \leftarrow$ the $m \times n$ biadjacency matrix of G
3: $A \leftarrow$ rank k SVD of D ▷ Step (1)
4: $\tilde{U}_1, \dots, \tilde{U}_k \leftarrow$ the clusters obtained by running k-means on the rows of A
5: **for** $i = 1, \dots, k$ ▷ Step (2)
6: $\theta \leftarrow$ compute θ as in Equation (2.3)
7: $\tilde{V}_i \leftarrow$ all vertices in V with at least $\theta|U_i|$ neighbors in U_i

Algorithm 1: The pcv algorithm

Similarly, we can compute the number of edges which are present in the random graph. To this end, let $|E(U_i, V_i)|$ denote the number of edges between vertices in U_i and V_i and let $|E(U \setminus U_i, V_i)|$ denote the number of edges between vertices in $U \setminus U_i$ and V_i.

Now we can *estimate* the parameters $\hat{p}$ and $\hat{q}$ from the cluster V_i that was computed from $\theta(p, q)$. For this purpose, we set

$$\hat{p} = \frac{|E(U_i, V_i)|}{|V_i| \cdot |U_i|} \quad \text{and} \quad \hat{q} = \frac{|E(U \setminus U_i, V_i)|}{|V_i| \cdot |U \setminus U_i|}.$$

Note that here $\hat{p}$ ($\hat{q}$) is simply the fraction of edges which should (not) have been there in the noiseless setting and which were observed in the random graph.

Obviously, if p and q were the correct parameters for generating V_i, then we should have $p \approx \hat{p}$ and $q \approx \hat{q}$. In particular, $|p - \hat{p}| + |q - \hat{q}|$ should be small.

This gives raise to the following heuristic algorithm for estimating good parameters p and q: Let $\mathcal{P}$ be a set of candidates for p and let $\mathcal{Q}$ be a set of candidates for q. Now iterate over all tuples $(p, q) \in \mathcal{P} \times \mathcal{Q}$ such that $p > q$. For each such tuple, generate the set V_i with parameter $\theta(p, q)$ using the high-degree thresholding algorithm. Given the set V_i, estimate $\hat{p}$ and $\hat{q}$ as described above. Of all the tuples, pick the one which minimizes the objective function $|p - \hat{p}| + |q - \hat{q}|$.

We experimentally evaluate the heuristic in Section 2.6.2.

2.5 Implementation

While so far we have been concerned with theory, we will now consider practice. The pseudocode of the algorithm we implemented is presented in Algorithm 1. As stated in Section 2.1, the algorithm performs two steps: (1) Recover the clusters $\tilde{U}_1, \dots, \tilde{U}_k$ in U. (2) Recover the clusters $\tilde{V}_1, \dots, \tilde{V}_k$ in V based on the $\tilde{U}_i$. We call the algorithm pcv, which is short for *project, cluster, vote*.

While for Step (2) we use exactly the algorithm discussed in Section 2.4, we made some changes for Step (1). The main reason is that Mitra's algorithm discussed in Section 2.3 was developed in a way to give theoretical guarantees and not necessarily to give the best results in practice.

Instead, for Step (1) we use a simpler algorithm for recovering the clusters $\tilde{U}_i$: Project the biadjacency of G on its first k left singular vectors and then run k-means. This delivers better results in practice and is conjectured to give the same theoretical guarantees as Mitra's algorithm (see [138, 147]).

We implemented Algorithm 1 in Python. To compute the truncated SVD we used scikit-learn [166]. The source code is available in the supplementary material of the conference pupblication [150].

When developing the algorithm, we also tried using other clustering methods than k-means. However, none of them delivered consistently better results than k-means and the differences in the outputs were mostly minor. Hence, we do not study this further here.

We note that due to k-means, pcv is a randomized algorithm. On the synthetic graphs we will consider, this had almost no influence on the quality of the results. On real-world graphs, this randomness resulted in different clusterings in each run of the algorithm. However, some "prominent clusters" were always there and the computed clusters always had an interpretable structure.

Parameters. The parameters p and q are only used to compute the parameter θ from Section 2.4 and, in settings in which they are unknown, they can be picked via the heuristic from Section 2.4.3. We note that in practice it might be reasonable to pick a different threshold θ for each cluster depending on its sparsity; we leave this to future work.

It suffices if k is a sufficiently tight upper bound on its true value. pcv will not necessarily output exactly k clusters; if k-means outputs less than k clusters, then pcv will do the same. In practice it is sometimes handy to use different values for k in the SVD and in k-means.

We further added a parameter $L \in \mathbb{N}$. In practice, often some of the $\tilde{U}_i$ returned by pcv are tiny (e.g., containing less than five vertices). To avoid creating too much output, we use the parameter L to ignore all clusters $\tilde{U}_i$ of size less than L. In the experiments we always set $L = 10$.

2.6 Experiments

In this section, we practically evaluate the performance of pcv. Throughout the experiments our main objective will be to understand how well pcv can recover small clusters on the right side of the graph. In the synthetic experiments, we will be most interested in how small p can be so that pcv can still recover clusters of size less than 10 on the right side of the graph. We picked real-world datasets from which we expect that they contain only very small clusters on the right side.

The experiments were done on a MacBook Air with a 1.6 GHz Intel Core i5 and 8 GB RAM. The source code and the synthetic data are provided in the supplementary materials of the conference publication [150].

2.6.1 Algorithms

pcv was compared with the lim algorithm by Lim, Wu and Xu [131], message by Ravanbakhsh, Póczos and Greiner [172], and the lfm algorithm by Rukat, Holmes and Yau [181]. For each of the algorithms, implementations provided by the authors were used. message and lfm are BMF algorithms (see Section 2.2).

When we report the running times of the algorithms, note that the quality of the implementations is incomparable. For example, lim is implemented in Matlab, message and pcv are purely implemented in Python and lfm is programmed in Python with certain subroutines precompiled using *Numba*.

During the experiments on synthetic data, all algorithms were given the correct parameters for the parameters k, p and q whenever the algorithms allowed this. For lim and lfm we optimized their parameters as follows.

lim takes as input a weight matrix W which is a weighted version of the biadjacency matrix B of the graph. After a correspondence with the authors of [131], we set

$$W = \log\left(\frac{p}{q}\right) B + \log\left(\frac{1-p}{1-q}\right)(1-B).$$

As output, lim returns a denoised version of the data. To obtain the left and right clusters of the graph, we applied k-means first to the rows of the output and then 2-means to the columns of the submatrices of the output; this is similar to what was reported in [131]. Further, lim has a parameter λ which [131] set to $\sqrt{2n}$ (this is 44.7 in our setting); we have run the algorithm with parameter $\lambda = 20, 25, 30, 35, 40, 45$ as sometimes this gave better results.

For the lfm algorithm we inverted the data (i.e., we ran the algorithm on the complement graph) to improve its performance and we fixed the value for λ to 0.5 for the first 100 iterations. Furthermore, we set the number of latent dimensions to k. This procedure was suggested in a correspondence with one of the authors of [181]. Since the results of the lfm algorithm depended heavily on the randomness of the algorithm, we ran the algorithm 10 times on each dataset; all other algorithms were run once.

Some of the algorithms returned fractional values in the interval $(0, 1)$ and we rounded them to 0/1 with threshold 0.5.

2.6.2 Synthetic Data

Let us start by considering the performance of the algorithms on synthetically generated graphs. The graphs were generated as described in Section 2.1.

The ground-truth clusters U_i and V_i were picked in the following way. For each U_i, ℓ vertices were added to the (initially empty) left side of the graph. On the right side of the graph, we inserted n vertices. Each of the V_j consists of r vertices which were picked uniformly at random from the n vertices. Due to the randomness in the graph generation, some of the V_j will overlap and most of them will not. By *size of a cluster* we mean the number of vertices contained in the cluster.

When not mentioned otherwise, the parameters were set to $n = 1000$, $k = 8$, $\ell = 70$, and $m = \ell \cdot k$ (i.e., 1000 vertices on the right, 8 ground-truth clusters on both sides and left-side clusters of size 70). The size of the right-side clusters was set to $r = 8$. The parameters p and q were set depending on the dataset.

For each of the reported parameter settings, five random graphs were generated. The results that are reported in the following are averages over these datasets. When an algorithm was run multiple times on the same dataset, we report the best result on the right clusters of the graph.

Quality Measure. Consider the k ground-truth clusters $U_1, \dots, U_k$ and the s clusters $\tilde{U}_1, \dots, \tilde{U}_s$ returned by an algorithm. The *quality* Q of the solution $\tilde{U}_j$ is computed as follows. For each ground-truth cluster U_i, find the cluster $\tilde{U}_j$ which maximizes the Jaccard coefficient of U_i and $\tilde{U}_j$. Then sum over the Jaccard coefficients for all ground-truth clusters U_i and normalize by k. Formally,

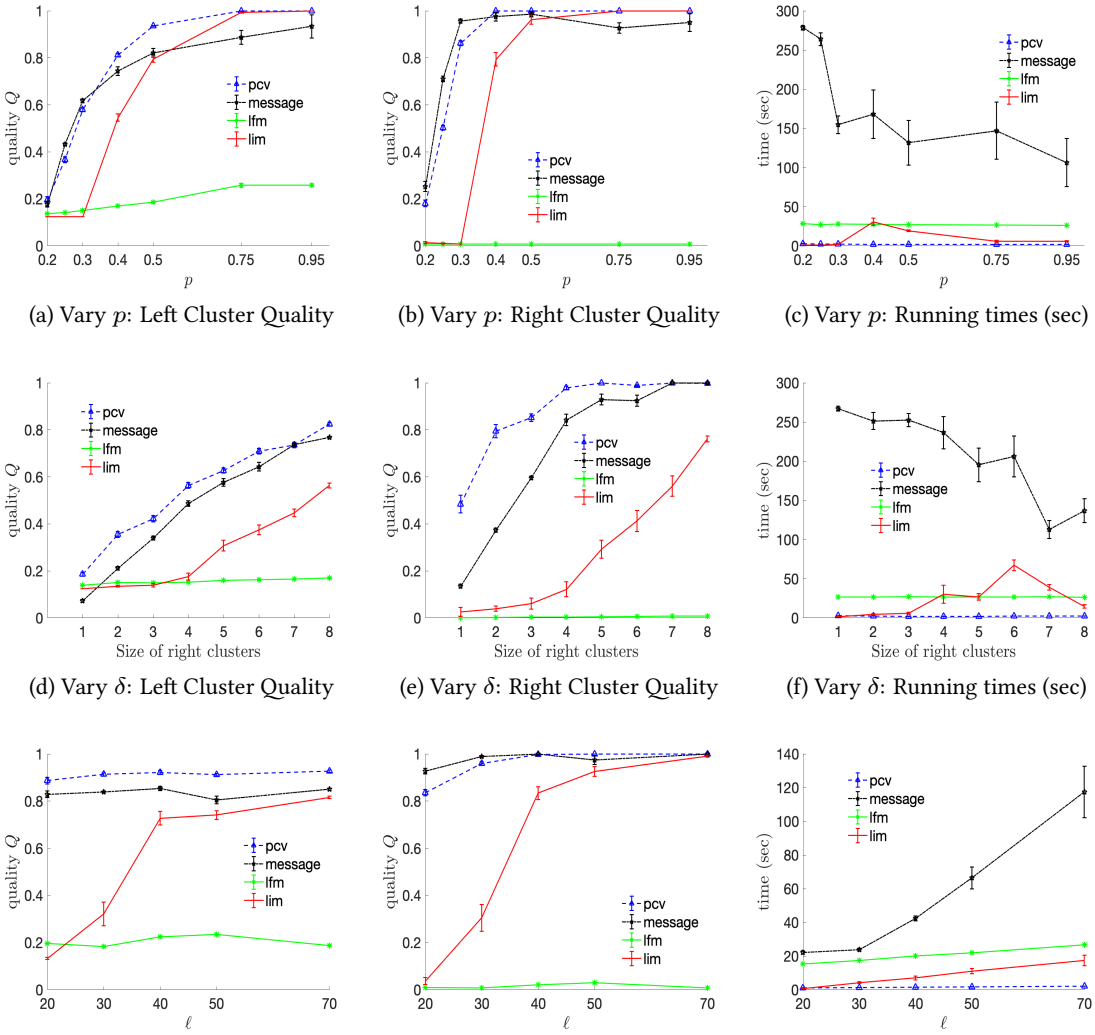
$$Q = \frac{1}{k} \sum_{i=1}^{k} \max_{j=1,\dots,s} J(U_i, \tilde{U}_j) \in [0, 1],$$

where $J(A, B) = |A \cap B|/|A \cup B|$ is the Jaccard coefficient. Higher values for Q imply a better quality of the solution. E.g., if $Q = 1$ then the clusters $\tilde{U}_j$ match *exactly* the ground-truth clusters U_i. We used the same quality measure for the clusters V_i.

Remark. Let us briefly motivate why we used the quality measure Q instead of using the reconstruction error. Let us first recall the definition of the reconstruction error. Let B be the biadjacency matrix of the graph G. Then for the outputs $\tilde{U}_i$, $\tilde{V}_i$ of an algorithm, define an $m \times n$ matrix A by setting $A_{uv} = 1$ iff there exists an i such that $u \in \tilde{U}_i$ and $v \in \tilde{V}_i$. Now the reconstruction error is defined as $||A - B||_2^2$. The main advantage of the reconstruction error is that it does not require knowledge about the ground-truth clustering. Thus, it can be easily computed also on real-world datasets. However, it has two major drawbacks for our purposes. First, the reconstruction error does not allow us to understand how well the algorithms perform on each side of the graph. With the quality measure Q, this is possible. Second, in the experiments we consider scenarios with very high destructive noise. For example, in random graphs with parameter $p = 0.4$ and ground-truth clusters U_i, V_i. Then it is more likely that an edge (u, v) from $u \in U_i$ to $v \in V_i$ is *not* present than that is this present in the graph. Thus, *an empty graph has a lower reconstruction error than the ground-truth clustering*, which is a highly undesirable property. The quality measure Q does not have this drawback.

Varying p. We start by studying how much the results of the algorithms are affected by destructive noise and vary the values for $p = 0.2, 0.25, 0.3, 0.5, 0.75, 0.95$, while is fixed to$q = 0.03$. The results are presented in Figures 2.1a–2.1c.

We see that on both sides of the graph, `pcv` and `message` outperform `lfm` and `lim` for $p \leq 0.3$; for $p \geq 0.5$, `lim` picks up and delivers very good results.



(a) Vary p: Left Cluster Quality (b) Vary p: Right Cluster Quality (c) Vary p: Running times (sec)

(d) Vary δ: Left Cluster Quality (e) Vary δ: Right Cluster Quality (f) Vary δ: Running times (sec)

(g) Vary ℓ: Left Cluster Quality (h) Vary ℓ: Right Cluster Quality (i) Vary ℓ: Running times (sec)

Figure 2.1: Results on synthetic data. Figures 2.1a–2.1c have varying p, Figures 2.1d–2.1f have varying sizes of the right clusters, Figures 2.1g–2.1i have varying ℓ. Markers are mean values over five different datasets; error bars are one third of the standard deviation over the five datasets.

In Figure 2.1a we see that on the left clusters, `pcv` and `message` deliver similar performances with `pcv` picking up the signal better for $p \geq 0.5$; the results of `lim` improve as p increases and they are perfect for $p = 0.75, 0.95$; `lfm` always delivers relatively poor results.

For the right clusters the situation is similar with `message` having slight advantages over `pcv` for $p \leq 0.3$; `pcv` and `lim` deliver better results than `message` in settings with less noise ($p \geq 0.75$). It is interesting to observe that `pcv` already re-

covers the ground-truth clusters on the right side for $p \geq 0.4$ and even for $p = 0.3$ the results are of good quality.

The running times of the algorithms are reported in Figure 2.1c. pcv is the fastest method with lim and lfm being somewhat slower. message is by far the slowest method and we see that when p is small, message takes a long time until it converges.

Varying sizes of the right clusters. We now study how small the right clusters V_i can get such that they can still be recovered by the algorithms. To this end, we vary the size of the right clusters and note that this corresponds to varying δ (for example, when all clusters are disjoint, δ is exactly twice the size of the right clusters).

Previously, we saw that pcv, message and lim did well at the recovery of right clusters of size 8 even for $p = 0.4$. We study this further by fixing $p = 0.4, q = 0.03$ and varying the size of the right clusters from 1 to 8. The results are reported in Figures 2.1d–2.1f.

The results for clustering the left side of the graph are presented in Figure 2.1d. We observe a clear ranking with pcv being the best algorithm before message; lim is the third-best algorithm and lfm is the worst.

For the right side of the graph (Figure 2.1e) we observe that pcv outperforms message for ground-truth clusters sizes less than 7; even for clusters of sizes 2 and 3, pcv finds good solutions. The performance of lim improves as the cluster sizes grow.

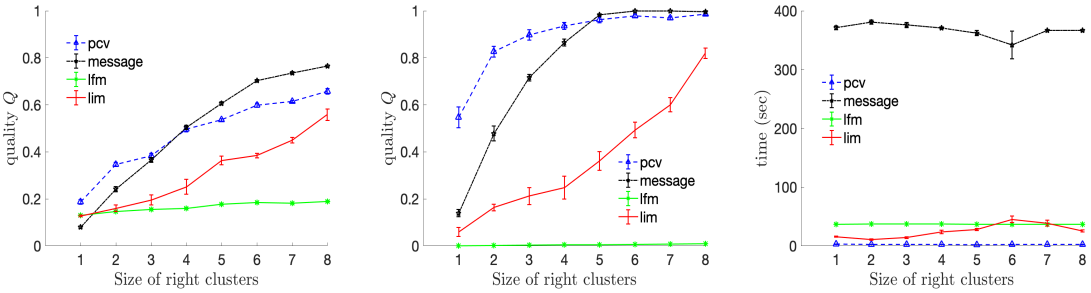
The running times (Figure 2.1f) are similar to what we have seen before for varying p.

Varying ℓ. We study how ℓ, the size of the left clusters U_i, influences the results of the algorithms. We used values $\ell = 20, 30, 40, 50, 70$. The other parameters were fixed to $p = 0.5$, $q = 0.03$, $k = 8$ and the size of the right clusters was set to 8. The results are reported in Figures 2.1g–2.1i.

On the left clusters, pcv is the best algorithm with message also delivering good results; the results of lim are of good quality for $\ell \geq 40$. On the right clusters, message is initially ($\ell \leq 30$) slightly better than pcv and for $\ell \geq 40$, pcv and message deliver essentially perfect results; lim finds good right clusters for $\ell \geq 40$. The running times are similar to what we have seen in previous experiments.

It is interesting and maybe even a bit surprising that even for $\ell = 20$, pcv and message can find very good clusters on the right side of the graph which only consist of 8 out of a 1000 vertices.

Varying right cluster sizes with wrong parameters. Next, we evaluate how sensitive the algorithms are towards wrong parameters. We repeated the experiment for varying sizes of the right clusters that we reported before, but this time we executed the algorithms with incorrect values for their parameters. This should



(a) Vary δ: Left Cluster Quality (b) Vary δ: Right Cluster Quality (c) Vary δ: Running times (sec)

Figure 2.2: Results on synthetic data. Figures 2.2a–2.2c have varying sizes of the right clusters and the algorithm were executed with wrong parameters. Markers are mean values over five different datasets; error bars are one third of the standard deviation over the five datasets.

affect `pcv`, `message` and `lim` since they take the most parameters. The results can be seen in Figures 2.2a–2.2c.

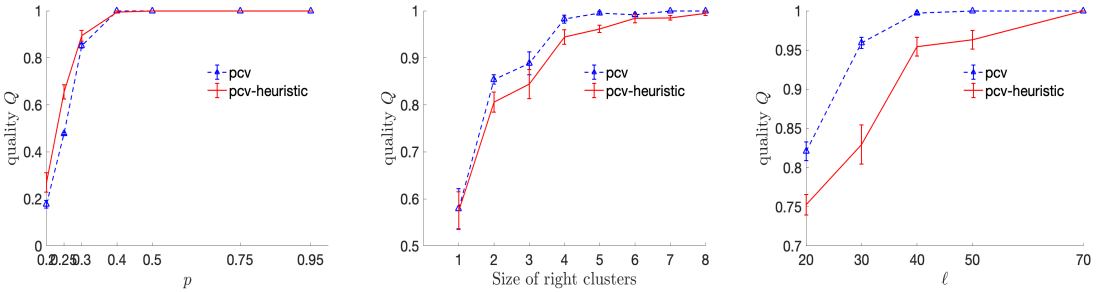
Recall: `pcv` and `lim` take as parameters p, q, k. `message` takes as parameters, p, q, k and the sizes of the clusters on the left sides and the right sides; we were generous and provided `message` with the correct values for the cluster sizes.

The true parameter values were $p = 0.4$, $q = 0.03$ and $k = 8$. We gave the algorithms the incorrect values $\tilde{p} = 0.6$, $\tilde{q} = 0.01$ and $\tilde{k} = 12$.

For the left clusters we see that for clusters of size 1 and 2, `pcv` is better than `message`, for sizes 3 and 4 they are on par and after that `message` is better. For the right side clusters we observe that until size 4 `pcv` is better than `message` and after that they are on par with perfect or almost perfect results. For `lim` we see that its performance improves as the cluster sizes grow.

Compared with the results when parameters are set correctly, the performance of all algorithms was relatively robust. `pcv`'s performance on the left clusters decayed while on the right clusters its results were relatively stable. For `message` the results were very robust and its performance on the right clusters even slightly improved; the latter might be down to much higher running times as reported in Figure 2.2c (apparently the algorithm takes a longer time to converge when run with incorrect parameters). For `lim` we observe that the change in the parameters has only a small influence on its results.

The fact that the performance of `pcv` decayed on the left clusters is down to the following two facts: (1) Rank-12-SVD picks up more noise than the rank 8 SVD. (2) When clustering the left side into 12 clusters instead of 8, k-means will partition the left side into too many sets. Due to (2), each left cluster is smaller than with the correct value for k. This causes the inference of the right side clusters of size at least 5 to be slightly less robust than when the algorithm is run with the correct parameters. However, note that for the right clusters these effects are only minor.



(a) Vary p: Right Cluster Quality (b) Vary δ: Right Cluster Quality (c) Vary ℓ: Right Cluster Quality

Figure 2.3: Results on synthetic data for pcv and pcv with the heuristic from Section 2.4.3. All plots report the quality for the recovery of the right clusters. Figure 2.3a has varying values for p, Figure 2.3b has varying sizes of the right clusters and Figure 2.3c has varying values for ℓ. Markers are mean values over five different datasets; error bars are one third of the standard deviation over the five datasets.

Evaluation of the heuristic. We evaluate the heuristic which was presented in Section 2.4.3 and compare it with the version of pcv which knows all parameters. For the heuristic, we set the candidates for p to $\mathcal{P} = \{0.3, 0.35, 0.4, \dots, 0.95\}$ and the set of candidates for q to $\mathcal{Q} = \{0.01, 0.02, 0.03, \dots, 0.1\}$.

We used the same synthetic datasets as before. We only report the results on the right clusters since the results on the left are exactly the same (because the the clustering of the left side of the graph did not change). The results are stated in Figure 2.3.

We see that (surprisingly) for varying values of p (Figure 2.3a), the heuristic version of pcv is slightly better than pcv with the correct parameters for very small values of p. For varying sizes of the right clusters (Figure 2.3b), the heuristic is slightly worse than the version of pcv which knows the correct parameters. For varying ℓ (size of the left clusters) in Figure 2.3c, the heuristic is visibly worse. The latter is perhaps to be expected because for smaller left clusters the inference on the right clusters might overfit easily when the heuristic is used.

Note that in all experiments the standard deviation of the qualities returned by the heuristic is much higher. This is not surprising since the solution returned by the heuristic depends more heavily on the randomness in the data (whereas pcv with the correct parameters mainly requires a good clustering of the left side of the graph).

Altogether, we observe that when the left clusters and the right clusters are both large enough, then the heuristic delivers results of good quality.

Conclusion. We conclude that pcv was very good at finding tiny clusters even with high destructive noise. In most cases, pcv delivered the solutions of highest quality and pcv was the fastest algorithm.

2.6.3 Real-World Data

pcv is qualitatively evaluated on two real-world datasets. Since the parameters required by pcv are not known, pcv was run with different parameters settings and the quality of the clusters was manually evaluated; the final setting of the parameters is reported for each dataset.

Datasets. The *BookCrossing* dataset[2] originates from Ziegler et al. [221]. It consists of users on the left side of the graph and books on the right side of the graph; if a user rated a book, there exists an edge between the corresponding vertices. The dataset was preprocessed so that all books read by less than 11 users and all users reading less than 11 books were removed. The resulting graph has 6195 users and 4958 books; the number of edges is 83550.

The *4News* dataset is a subset of the *20Newsgroups* dataset; it was preprocessed by Ata Kabán (see [141]). The data contains the occurrences of 800 words (right side of the graph) over 400 posts (left side of the graph) in four different Usenet newsgroups about cryptography, medicine, outer space, and christianity; for each newsgroup there are 100 posts. The graph has 11260 edges.

Qualitative Evaluation. *BookCrossing.* For the BookCrossing dataset, pcv was run with parameters $k = 20$, $\theta = 0.2$ and $L = 10$; pcv finished in less than 2 minutes.

pcv returns 12 user-clusters (i.e., on the left side of the graph) with size at least L. Out of these 12 user-clusters, 9 have a non-empty book-side (right side of the graph). The largest user-cluster contains 4268 vertices and has an empty book-side (right side). We will now discuss some of the clusters with non-empty right sides. All of those clusters have a natural interpretation.

The returned clusters mostly consist of books written by the same authors (as one would expect). Two clusters were consisting of the *Harry Potter* books by Joanne K. Rowling; the first cluster contained the five *Harry Potter* books that were published until 2004 (when the dataset was created) and contains 92 users, the other one consisted of the first three books of the series and contained 60 users. There is one cluster containing four books written by Anne Rice (64 users), one cluster containing seven books written by John Grisham (67 users), and one clusters containing 46 books written by Stephen King (12 users). pcv also returns two clusters containing a single book: *The Da Vinci Code* by Dan Brown (215 users) and *The Lovely Bones* by Alice Sebold (261 users).

4News. For this dataset we observe that it is useful to set the parameter k in the SVD and in the call to k-means to different values. With this, we can obtain more general or more specific clusters: Setting the value k for k-means to a smaller (larger) value, creates less (more) clusters on the left side of the graph. This will also make the right-side clusters more general (specific).

[2] http://www2.informatik.uni-freiburg.de/~cziegler/BX/

We used $k = 30$ for the SVD and $k = 50$ for k-means to obtain relatively specific clusters. The value of k is so large, because the dataset contains many outliers that create a lot of left-side clusters of size 1. Further, we set $\theta = 0.3$, $L = 10$.

For each of the four newsgroups, pcv finds clusters. In total, pcv finds five clusters of which one has an empty right side (225 posts). The cluster (18 posts) returned for the cryptography newsgroup is *public, system, govern, encrypt, decrypt, ke(y), secur(ity), person, escrow, clipper, chip* (a *clipper chip* is an encryption device developed by the NSA). For the medicine newsgroups, pcv finds the cluster (24 posts) *question, stud(y), year, effect, result, ve, call, doctor, patient, medic, read, level, peopl(e), thing*. The cluster (19 posts) *concept, system, orbit, space, year, nasa, cost, project, high, launch, da(y), part, peopl(e)* explains the topics of the outer space newsgroup well. For the christian religion newsgroup we obtain the cluster (24 posts) *christian, bibl(e), read, rutger, god, peopl(e), thing*.

2.7 Conclusion

We presented pcv, the first algorithm which can provably recover tiny clusters in bipartite graphs. In experiments, pcv outperformed state-of-the-art methods on synthetic datasets and it found interesting clusters in real-world data.

An interesting challenge for future research will be to study random graph models, where the clusters on both sides of the graph are allowed to overlap (in this chapter we allowed the right-side clusters to overlap but the left-side clusters had to be mutually disjoint). This would immediately imply theoretical guarantees for the Boolean matrix factorization problem [142] under random inputs.

CHAPTER 3

Biclustering and Boolean Matrix Factorization in Data Streams

We study the clustering of bipartite graphs and Boolean matrix factorization in data streams. We consider a streaming setting in which the vertices from the *left* side of the graph arrive one by one together with all of their incident edges. We provide an algorithm that, after one pass over the stream, recovers the set of clusters on the *right* side of the graph using sublinear space; to the best of our knowledge, this is the first algorithm with this property. We also show that after a second pass over the stream, the left clusters of the bipartite graph can be recovered and we show how to extend our algorithm to solve the Boolean matrix factorization problem (by exploiting the correspondence of Boolean matrices and bipartite graphs). We evaluate an implementation of the algorithm on synthetic data and on real-world data. On real-world datasets the algorithm is orders of magnitudes faster than a static baseline algorithm while providing quality results within a factor 2 of the baseline algorithm. Our algorithm scales linearly in the number of edges in the graph. Finally, we analyze the algorithm theoretically and provide sufficient conditions under which the algorithm recovers a set of planted clusters under a standard random graph model.

3.1 Introduction

Bipartite graphs appear in many areas in which interactions of objects from two different domains are observed. Hence, finding interesting clusters (also called communities) in bipartite graphs is a fundamental and well-researched problem with many applications; this problem is often called *biclustering*. For example, in social networks the two domains could be users and hashtags and an interaction corresponds to a user using a certain hashtag; finding clusters in such a graph corresponds to finding groups of hashtags used by the same users and groups of users using the

same hashtags [200]. Biclustering has many applications across many domains such as computational biology [67, 134], text mining [63] and finance [107].

Many real-world bipartite graphs have three natural properties. First, the numbers of vertices on both sides of the graphs are very large, while at the same time their density is extremely low, i.e., the graphs are very sparse. For example, consider a bipartite graph consisting of users on the left side of the graph and movies on the right side of the graph, where an edge indicates that a user rated a movie. Such graphs often consist of millions of users and movies, but the average degree is constant. Second, the degrees on one side of the graph are usually bounded by a small constant, while on the other side of the graph a few vertices have extremely large degrees. Continuing the example from above, note that users usually do not rate more than 1000 movies, but a small number of popular movies is rated by millions of users. The third property is that the clusters on the high-degree side of the graph are usually relatively small. Again continuing the above example, those groups of movies which are watched by the same users typically do not consist of more than 50 movies.

Furthermore, for many real-world bipartite graphs, it is natural to assume that the left-side vertices appear in a data stream, for example, in Natural Language Processing [84], market basket analysis, network traffic analysis and stock price analysis [50]. For instance, in market basket analysis the left-side vertices correspond to transactions in a supermarket and incident edges indicate which products were bought. To efficiently find interesting clusters in such datasets, we need to develop streaming algorithms for the biclustering problem. Another motivation to study the streaming setting is that current static algorithms do not scale to the previously mentioned real-world graphs with millions of vertices, because these methods have prohibitively high memory consumptions and running times. Streaming algorithms could mitigate this issue due to improved memory efficiency and speed.

Our Contributions. We address this question and provide the first streaming algorithms for the biclustering problem. In particular, we study a streaming setting in which the vertices from the left side of the bipartite graph arrive one after another, together with all of their incident edges. Then after a single pass over the stream, the algorithm must output the set of right-side clusters of the graph. The algorithm is then allowed a second pass over the stream in order to output the left-side clusters. See Section 3.2.1 for the formal definition of the problem.

To obtain our algorithms, we heavily exploit the previously mentioned properties of real-world bipartite graphs. Formally, we assume that there exists a number s such that the degree of all left-side vertices and the size of all right-side clusters is at most s. This implies that, in total, the graph contains $O(ms)$ edges, where m is the number of vertices on the left side of the graph.

We introduce the sofa algorithm which returns the right-side clusters of the graph after a single pass over the stream and using sublinear memory. To the best our of knowledge, sofa is the first algorithm with this property. The running time of sofa is $O(ms \cdot k \log m)$, where k is the number of clusters to be recovered; note that this running time is within a $O(k \log m)$ factor of the size of the graph. During its

running time, sofa uses $O(ks \log m)$ space; observe that this space usage sublinear in the size in the size of the graph as long as $s = o(m/\log m)$ which is realistic in practice (as we argued before). Furthermore, we show that the left-side clusters of the graph can be computed using a second pass over the stream and using space $O(m)$, which is optimal since we have to output a cluster assignment for each of the m left-side vertices of the graph.

We also provide theoretical guarantees for a version of sofa. We show that under a standard random graph model, a version of sofa returns a set of planted ground-truth clusters with information-theoretically optimal memory usage; see Theorem 3.1 for details. We also provide similar yet weaker guarantees for the practical version of sofa.

Next, we show how sofa can be extended to solve the Boolean matrix factorization problem, which is popular in the data mining and machine learning communities. We obtain similar guarantees on space and run-time as above. Unfortunately, we cannot provide any quality guarantees here, because the lower bounds from [52] rule out obtaining non-trivial approximation ratios for practical BMF algorithms (see Section 3.8 for details). Thus, sofa is a heuristic for BMF, but our experiments show that it works well in practice.

We evaluate sofa on synthetic as well as on real-world datasets. On synthetically generated random graphs, our experiments show that sofa returns clusters, that are close to the planted ground-truth clusters and that its running time scales linearly in the number of edges in the graph. On real-world datasets, our experiments show that sofa is orders of magnitudes faster and more memory-efficient than a static baseline algorithm, while at the same time achieving objective function values within factor 2 of the baseline. In concrete terms, sofa can process a graph with millions of vertices, for which the static baseline algorithm runs out of memory, using only 500 MB of RAM and, further, sofa can process a graph with hundreds of thousands of edges within less than three hours, while the baseline algorithm requires several days to finish.

Outline of the Chapter. The chapter is arranged as follows. In Section 3.2 we formally define the problems we study. Then in Section 3.3 we introduce sofa, which performs a single pass over the left side of a bipartite graph and then returns the right-side clusters. We show how the left-side clusters can be recovered during a second pass over the stream in Section 3.4. In Section 3.5, we discuss certain adjustments of the algorithms that we made during the implementation and then we evaluate sofa experimentally in Section 3.6. Section 3.7 contains our theoretical analysis. We discuss related work in Section 3.8 and conclude in Section 3.9.

3.2 Preliminaries

In this section, we formally introduce the problems we study, we discuss their relationship and we introduce an important subroutine of our algorithms.

3.2.1 Biclustering in Random Graphs

We study biclustering of random bipartite graphs. Let $G = (U \cup V, E)$ be a bipartite graph, where U is the set of vertices on the left side of the graph and V is the set of vertices on the right side of the graph. We assume that U is partitioned into subsets $U_1, \dots, U_k$ for $k > 1$ and $V_1, \dots, V_k$ are subsets of V (it is not necessary that the V_j are mutually disjoint or that their union is the set V).

Now let $p, q \in [0, 1]$ be probabilities with $p > q$. In our random model, vertices $u \in U_i$ have edges to vertices $v \in V_i$ with "large" probability and to vertices in $v \in V_j$ with $i \neq j$ with "low" probability. More concretely, we assume that

$$\mathbf{Pr}\left((u, v) \in E\right) = \begin{cases} p, & \text{if } u \in U_i, v \in V_i, \\ q, & \text{if } u \in U_i, v \in V_j, i \neq j. \end{cases} \tag{3.1}$$

Now the computational problem is as follows. We assume that our algorithms obtain as input a graph G generated from the random model above and the parameters k, p and q (but have no knowledge about the sets U_i and V_j). The task is to recover the clusters U_i and V_j from G; that is, the algorithm must output clusters $\tilde{U_1}, \dots, \tilde{U_k} \subseteq U$ and $\tilde{V_1}, \dots, \tilde{V_k} \subseteq V$, such that $\{\tilde{U_1}, \dots, \tilde{U_k}\} = \{U_1, \dots, U_k\}$ and $\{\tilde{V_1}, \dots, \tilde{V_k}\} = \{V_1, \dots, V_k\}$.

We decided to study the above random graph model for two reasons. First, the model has been widely studied theoretically, e.g., in machine learning [150, 219] and in mathematics [2, 220], and similar models have been used to derive practical algorithms [172, 180]. Second, when dropping the random graph assumption and assuming worst-case inputs, biclustering problems are NP-hard [161] and require prohibitively high running times [52].

In the streaming setting, the algorithm's input is a stream of the left-side vertices $u \in U$, where each vertex arrives together with all of its incident edges. We further assume that for some parameter s, each $u \in U$ has at most s incident edges and that $|V_i| \leq s$ for all i. Note that the stream only contains left-side vertices $u \in U$ and does *not* contain the vertices $v \in V$. After single pass over the stream, the algorithm must return the right-side clusters $\tilde{V_i}$. Then, the algorithm is allowed a second pass over the stream to output the left-side clusters $\tilde{U_i}$.

Next, we state our theoretical guarantees. We prove that after a single pass over the left-side vertices of a bipartite graph, the planted right-side clusters can be recovered if some conditions hold. We write $A \triangle B = (A \setminus B) \cup (B \setminus A)$ to denote the symmetric difference.

Theorem 3.1. *Let $G = (U \cup V, E)$ be a random bipartite graph with planted clusters $U_1, \dots, U_k$ and $V_1, \dots, V_k$ as above. Let $p \in [1/2, 0.99]$ and $s = \max_i |V_i|$. There exist constants K_1, K_2, K_3, K_4 such that if*

- $q \leq K_1 ps/n$,
- $|V_i| \geq K_3 \log n$ *for all* i,
- $|U_i| \geq K_2 \log n$ *for all* i,
- $|V_i \triangle V_{i'}| \geq K_4 s$ *for* $i \neq i'$,

then there exists an algorithm which returns clusters $\tilde{V_1}, \dots, \tilde{V_k}$ such that with high

probability $\{\tilde{V_1}, \dots, \tilde{V_k}\} = \{V_1, \dots, V_k\}$. *The algorithm uses* $O(ks)$ *space and has a running time of* $O(mks)$.

Let us briefly discuss this result and for simplicity assume that the V_i are disjoint and have size $|V_i| = s = \Omega(\log n)$. Then the bounds for p and q essentially require that $p > 1/2$, $q \approx ps/n$ and $|U_i| = \Omega(\log n)$. While this is much weaker than bounds derived for static algorithms for this type of random graph model (e.g., [150, 219]), the static algorithms do not use sublinear space. Furthermore, the bounds on p and q are almost optimal when one wants to ensure that a greedy clustering of the left-side vertices succeeds.[1]

We also show that *any* algorithm recovering the planted right-side clusters must use space $\Omega(ks)$. Thus, the space usage of the algorithm from the theorem is optimal. We prove the theorem and the proposition in Section 3.7.

Proposition 3.2. *Any algorithm solving the above biclustering problem requires at least* $\Omega(ks)$ *space.*

3.2.2 Boolean Matrix Factorization (BMF)

In the Boolean Matrix Factorization (BMF) problem, the input is a matrix $B \in \{0,1\}^{m \times n}$ and the task is to find factor matrices $L \in \{0,1\}^{m \times k}$ and $R \in \{0,1\}^{k \times n}$ such that $||B - L \circ R||_2$ is minimized. Here, $\circ$ denotes matrix multiplication under the Boolean algebra, i.e., for all $i = 1, \dots, m$ and $j = 1, \dots, n$,

$$(L \circ R)_{ij} = \bigvee_{r=1}^{k} (L_{ir} \wedge R_{rj}) .$$

In the streaming setting, the algorithm's input is a stream consisting of the rows B_i of B, where we assume that each row B_i has at most s non-zero entries. After a single pass over the stream, the algorithm must output the right factor matrix R. Then, the algorithm is allowed a second pass over the stream to compute the left factor matrix L.

While the biclustering problem and the BMF problem might appear quite different at first glance, they are tightly connected. Indeed, there is a one-to-one correspondence between bipartite graphs $G = (U \cup V, E)$ with $U = \{u_1, \dots, u_m\}$ and $V = \{v_1, \dots, v_n\}$ and Boolean matrices $B \in \{0,1\}^{m \times n}$: The rows of B correspond to the vertices $u_i \in U$ and the columns of B correspond to the vertices $v_j \in V$;

[1] Roughly speaking, the condition on q ensures that the left-side vertices have more "signal edges" than "noise edges". More concretely, in our setting with small right-side clusters V_i of size $|V_i| \approx s \ll n$, we have that $n - s \approx n$. Thus, in expectation every vertex $u \in U_i$ has ps "signal-edges" to vertices from its corresponding right-side cluster V_i and $q(n-s) \approx qn$ "noise-edges" to vertices in $V \setminus V_i$. Now, if $q \gg ps/n$, then u has $qn \gg ps/n \cdot n = ps$ "noise-edges" and, hence, more "noise edges" than "signal-edges". In such a case, the Hamming distances of vertices from the same cluster U_i are essentially identical to the Hamming distances of vertices from different clusters U_i and U_j, $i \neq j$. Therefore, clustering the vertices in U based on their Hamming distance cannot succeed anymore and, hence, the analysis of our algorithm is tight w.r.t. the choice of q.

now one sets $B_{ij} = 1$ iff $(u_i, v_j) \in E$. This is yields a bijective mapping between bipartite graphs and Boolean matrices; B is often called the *biadjacency matrix* of G.

Furthermore, there exists a correspondence between clusterings $U_1, \dots, U_k \subseteq U$ and $V_1, \dots, V_k \subseteq V$ and the factor matrices L and R: The clusters U_i correspond to the columns of L and the clusters V_j correspond to the rows of R. More precisely, consider the r'th column of L and set it to the indicator vector of U_k, i.e., we set $L_{ir} = 1$ iff $u_i \in U_r$. Simililary, we set $R_{rj} = 1$ iff $v_j \in V_r$.

There are two main differences between the problems. First, while in biclustering we try to recover a set of planted ground-truth clusters, in BMF we try to optimize an objective function. However, when $p > 1/2 > q$, a "good" biclustering solution will also provide a good BMF solution and vice versa. Second, in biclustering each vertex $u \in U$ belongs to exactly one cluster U_i (since the U_i partition U). This would correspond to the constraint in BMF that each column of the factor matrix L must contain exactly one non-zero entry. However, in BMF we do not make this assumption and allow each column of L to contain arbitrarily many non-zero entries. Thus, in BMF the vertices $u \in U$ are allowed to be member of multiple clusters $U_{i_1}, \dots, U_{i_t}$ (and the clusters U_i do not have to be mutually disjoint). To address these differences, in Section 3.4 we use different algorithms for computing the left-side clusters U_i for biclustering and for BMF.

3.2.3 Mergeable Heavy Hitters Data Structures

Next, we recap mergeable heavy hitters data structures, which we will use as subroutines in our algorithms.

Let $X = (e_1, \dots, e_N)$ be a stream of elements from a discrete domain A. The *frequency* f_a of an element $a \in A$ is its number of occurrences in the stream, i.e., $f_a = |\{i : e_i = a\}|$. In the *heavy hitters* problem the task is to output all elements with $f_a \geq \varepsilon N$ and none with $f_a < \varepsilon N/2$ after a single pass over the stream for $\varepsilon > 0$.

Misra and Gries [146] provided a data structure which solves the heavy hitters problem using $O(1/\varepsilon)$ space. In fact, their data structure can approximate the frequency of each element $a \in A$ with additive error at most $\varepsilon N/2$. For the rest of the chapter, we will denote Misra–Griess data structures by MG.

Agarwal et al. [10] showed that Misra–Gries data structures are *mergeable*: Let MG_1 and MG_2 be two Misra–Gries data structures which were constructed on two different streams X_1 and X_2. Then there exists a merge algorithm which on input MG_1 and MG_2 constructs a new data structure, that satisfies the same guarantees as a Misra–Gries data structure which was built on the concatenated stream $X_1 \cup X_2$. We write $\mathsf{MG}_1 \cup \mathsf{MG}_2$ to denote such a merged data structure.

Remark. While we use the mergeable version of the Misra–Gries data structure, we could as well other mergeable heavy hitters data structures such as the count-min sketch [57]. See [10] for more details on mergeable data structures.

3.3 First Pass: Recover Right Clusters

We describe two algorithms for computing the right clusters $\tilde{V_j}$. As described in Section 3.2, we assume that the algorithms obtain as input a stream $U = (u_1, \dots, u_m)$ consisting of vertices from the left side of the graph, where each u_i arrives together with all of its at most s edges to vertices on the right side of the graph. After a single pass over U, the algorithm must return clusters $\tilde{V_1}, \dots, \tilde{V_k}$ on the right side of the graph.

It will be convenient to identify the vertices $u \in U$ with bit-vectors $x_u \in \{0,1\}^n$, where we set $x_u(j) = 1$ iff $(u, v_j) \in E$, i.e., $x_u(j) = 1$ iff vertex u is a neighbor of $v_j \in V$. For two vertices $u, u' \in U$, we let $d(x_u, x_{u'}) = |\{j : x_u(j) \neq x_{u'}(j)\}|$ denote the Hamming distance of x_u and $x_{u'}$, i.e., $d(x_u, x_{u'})$ measures the number of vertices in V which are incident upon u or u' but not both.

We will first describe a simplified greedy algorithm to highlight our main ideas; this is the algorithm mentioned in Theorem 3.1. Then we provide a second, more practical, algorithm in Section 3.3.2; we implement and evaluate this algorithm in Sections 3.5 and 3.6.

3.3.1 Warm Up: Greedy Biclustering

We start by discussing a simplified greedy algorithm to explain the main idea of our approach. This greedy algorithm has the guarantees stated in Theorem 3.1.

Before describing the algorithm, let us first make two observations about the properties of the random graph model in Section 3.2.1: (1) Suppose we know a planted left-side cluster U_i and we want to recover its corresponding right-side cluster V_i. Then observe that by Equation 3.1 every vertex $v \in V_i$ has $p|U_i|$ neighbors in U_i and every vertex $v \notin V_i$ has $q|U_i|$ neighbors in U_i. Thus, if U_i is large enough and p is sufficiently larger than q, we can find a threshold θ such that with high probability all $v \in V_i$ have more than $\theta|U_i|$ neighbors in U_i and all $v \notin V_i$ have less than $\theta|U_i|$ neighbors in U_i. Hence, recovering the cluster V_i essentially boils down to identifying those vertices in V_i which are frequently neighbors of vertices in U_i. In other words, we want to find the heavy hitters among the neighbors of vertices in U_i. (2) The second insight is that when processing the stream, the vertices $u, u' \in U_i$ from the same cluster will have similar neighborhoods in V and, hence, $d(x_u, x_{u'})$ is small. More concretely, assume that $d(x_u, x_{u'}) < \alpha$ for some suitable parameter α. On the other hand, if $u \in U_i$ and $u'' \in U_j$ with $i \neq j$, their neighborhoods will be quite different and $d(x_u, x_{u''}) > \alpha$ is large. Thus, a greedy clustering of the vertices $u \in U$ based on the distances of their corresponding vectors x_u suffices to recover the U_i. In Section 3.7, we show how θ and α can be picked under the conditions from Theorem 3.1.

Roughly speaking, the algorithm works as follows. It assumes that it obtains parameters θ and α with the above properties as input. Now the algorithm greedily forms clusters of all left-side vertices which have distance at most α; this corresponds to Observation (2) above. To save memory, the algorithm only stores *a single*

$C \leftarrow \emptyset$
for $u \leftarrow$ next vertex from stream
  $d \leftarrow \min_{c \in C} d(x_u, x_c)$
  if $d > \alpha$ ▷ open u as center
    $C \leftarrow C \cup \{u\}$
    $n_u \leftarrow 1$
  else ▷ Assign u to its closest center $c(u)$
    $c(u) \leftarrow \arg\min_{c \in C} d(x_u, x_c)$
    $\mathsf{MG}(c(u)) \leftarrow \mathsf{MG}(c(u)) \cup \mathsf{MG}(u)$
    $n_{c(u)} \leftarrow n_{c(u)} + 1$
for all $c \in C$ ▷ Postprocessing
  $\tilde{V}_c \leftarrow \{v_j \in V :$ the counter of j in $\mathsf{MG}(c)$ is at least $\theta n_c\}$

Algorithm 2: Greedy-clustering (U, α, θ)

vertex for each cluster. Furthermore, for each cluster consisting of left-side vertices, the algorithm keeps track how many of its edges are incident upon each right-side vertex $v \in V$. Since we do not have enough memory to store a counter for each vertex $v \in V$, the algorithm uses the mergeable heavy hitters data structure from Section 3.2.3 to approximately keep track of how many times each right-side vertex appeared; this corresponds to Observation (1) above.

Now we describe the algorithm more formally and present its pseudocode in Algorithm 2. The algorithm obtains as input U, a distance parameter α and a rounding threshold θ. It maintains a set of *centers* C which is initially empty. For each center $c \in C$, the algorithm stores a heavy hitters data structure $\mathsf{MG}(c)$ with $O(s)$ counters and a counter n_c denoting how many vertices have been assigned to c.

Now the algorithm processes the vertices $u \in U$ as follows. First, it checks whether x_u has Hamming distance more than α from all centers $c \in C$. If this is the case, the algorithm opens u as a new center. That is, it sets $C \leftarrow C \cup \{u\}$ and sets $n_u \leftarrow 1$. Else, there exists a center $c(u) \in C$ with $d(x_u, x_{c(u)}) \leq \alpha$ and the algorithm *assigns* u to $c(u)$. When assigning u to $c(u)$, the algorithm first creates a heavy hitters data structure $\mathsf{MG}(u)$ containing all j such that $(u, v_j) \in E$ (note that the algorithm has access to this information since u arrives together with all of its incident edges). Then it merges $\mathsf{MG}(c(u))$ and $\mathsf{MG}(u)$ and updates $\mathsf{MG}(c(u))$ to this merged heavy hitters data structure. Furthermore, the algorithm increases the counter $n_{c(u)}$ by 1. Then it proceeds with the next point from the stream.

When the algorithm finished processing the stream, it performs a postprocessing step. It iterates over all centers $c \in C$ and sets $\tilde{V}_c$ to all vertices $v_j \in V$ such that the counter of j in $\mathsf{MG}(C)$ is at least θn_c, where θ is the rounding threshold from the input and n_c is the number of vertices that were assigned to c. Then the algorithm outputs the clusters $\tilde{V}_c$ as its solution.

Remark. Note that Algorithm 2 only delivers good results when the parameters α and θ provide exactly those guarantees which we discussed at the beginning of the subsection. In Section 3.7 we show how α and θ can be set when the parameters p,

q and k are known for random graph models as introduced in Section 3.2.1; under this assumption we show that the algorithm indeed returns the planted clusters $V_1, \dots, V_k$ after a single pass over the stream and using essentially optimal space. However, in practice it is unrealistic that one has knowledge about these parameters. Especially setting the parameter α seems troublesome; for example, when setting α incorrectly, one cannot even guarantee to obtain k clusters in total. We show how to resolve this issue in the next subsection.

3.3.2 Biclustering Using Importance Sampling

We introduce the sofa algorithm which constitutes our main contribution; sofa is short for *Streaming bOolean FactorizAtion*. sofa performs a single pass over the vertices $u \in U$ and afterwards returns clusters $\tilde{V}_1, \dots, \tilde{V}_k$. One can view sofa as the more practical version of Algorithm 2, since it it does not require the parameter α which is not available in practice. In a nutshell, we will replace the greedy clustering from Algorithm 2 by the streaming k-Medians algorithm from Braverman et al. [44] which is based on importance sampling. The pseudocode of sofa with all details is presented in Algorithm 3.

Roughly speaking, sofa works as follows. sofa maintains a set of *centers* C which is initially empty; we impose that C is never allowed to contain more than $c_{\max}$ vertices, where $c_{\max}$ is a user-defined parameter. As before, for each center $c \in C$, the algorithm maintains a heavy hitters data structure $\mathsf{MG}(c)$. When sofa processes the vertices from the stream and a new vertex u arrives, sofa computes the distance $d = d(x_u, x_{c(u)})$ from u to the closest center $c(u)$ in C. It then opens u as new center with probability proportional to d; if u is not opened as a center, sofa assigns u to $c(u)$. Thus, if u is "close" to $c(u)$ then u is unlikely to become a new center and more likely to be assigned to $c(u)$; on the other hand, if u is "far away" from $c(u)$ (and, hence, all centers), then u is likely to become a new center. As before, when a vertex u is assigned to $c(u)$, the indices of all neighbors of u are added to $\mathsf{MG}(c(u))$. Next, suppose that after opening a new center, the set C contains $c_{\max}$ centers. Then sofa restarts on the stream which only consists of the $c_{\max}$ centers in C and all unprocessed vertices of the stream. When sofa restarts on the centers of C and one of the previous centers c_i is assigned to another previous center c_j, then sofa merges their corresponding heavy hitters data structures $\mathsf{MG}(c_i)$ and $\mathsf{MG}(c_j)$ as described in Section 3.2. Finally, after processing all vertices from the stream and obtaining a set of centers C together with their heavy hitters data structures, we run a postprocessing step. At this point C can contain more than k centers (but at most $c_{\max}$). We run a static k-Medians algorithm on the vectors x_c for $c \in C$ to obtain a clustering of C into subsets $C_1, \dots, C_k$. For each C_i, we merge the heavy hitters data structures of the centers in C_i and denote this merged data structure as MG_i. As before, we set $\tilde{V}_i$ to all vertices $v_j \in V$ which have a counter of value at least $\theta|C_i|$ in MG_i.

We now elaborate on the details of sofa. At the beginning, sofa initializes a lower bound *LB* on the k-Medians clustering cost of the points x_u in the stream

to 1. It also maintains an approximation of the current cost of the clustering which we denote *cost* and initialize to 0. After that, sofa starts processing the vertices from the stream. We maintain a set of centers C for which we ensure that $|C| < c_{\max}$ at all times. For each center we store a heavy hitters data structure from Section 3.2.3 with $O(s)$ counters.

When starting to process the vertices from the stream, sofa computes a weight $f \leftarrow LB/(k(1+\log n))$. As long as there are unread vertices in the stream, $|C| < c_{\max}$ and *cost* $< 2LB$, sofa proceeds as follows. It reads the next vertex u from the stream and sets d to the distance $d(x_u, x_{c(u)})$ of u to its closest center $c(u)$. Now it opens u as a new center with probability $\min\{w(u) \cdot d/f, 1\}$, where $w(u)$ is the weight of u. sofa maintains as invariant that if u was a previously unprocessed vertex from the stream, then $w(u) = 1$, and, if u was a center before, then $w(u)$ is the number of vertices which were previously assigned to u. If u is opened as a new center, we set $C \leftarrow C \cup \{u\}$. If u is assigned to its closest center $c(u)$, then we increase *cost* by $w(u) \cdot d$, increase the weight of $c(u)$ by $w(u)$ and set $\mathsf{MG}(c(u))$ to the merged heavy hitters data structures of $\mathsf{MG}(c(u))$ and $\mathsf{MG}(u)$.

If at some point $|C| = c_{\max}$ or *cost* $> 2LB$, then sofa doubles *LB*. Furthermore, sofa restarts on the stream which consists of the $c_{\max}$ vertices of C and all unprocessed vertices from U (in this order). Note that the vertices $c \in C$ still have their previously assigned weights $w(c)$, whereas the vertices in the unprocessed part of U all have weight 1.

After sofa finished processing all vertices from the stream, we perform a postprocessing step. We start by running a static $O(1)$-approximate k-Medians algorithm on the points x_c for $c \in C$ which uses only $O(|C| \cdot s)$ space and which runs in time $\mathrm{poly}(|C| \cdot s)$; this can be done, for example, using the local search algorithm by Arya et al. [19]. This provides us with a clustering of C into disjoint subsets $C_1, \dots, C_k$. Now for each $i = 1, \dots, k$, we set MG_i to the merged heavy hitters data structure of all vertices in C_i and $|C_i|$ to the sum of the weights of all vertices in C_i. Finally, we set $\tilde{V}_i$ to all vertices $v_j \in V$ such that the counter of j in MG_i is at least $\theta|C_i|$.

Space Usage and Running Time. We briefly argue that sofa's space usage is $O(ks \log m)$ and its running time is bounded by $O(mks \log m)$. Observe that the main space usage comes from storing the set of centers C together with a heavy hitters data structure for each center. Recall that we ensure that $|C| \leq c_{\max}$ at all times. Furthermore, each center has $O(s)$ incident edges (by assumption on our input stream) and we set the number of counters for each heavy hitters data structure to $O(s)$. Thus, the total space usage is $O(c_{\max} s)$.

Remark. We use the streaming k-Medians clustering algorithm from [44], because the centers it maintains are points from the stream. Thus, if these points are sparse, the space usage of sofa for storing centers directly benefits from this. Algorithms for streaming k-Means (e.g., [187]) often include steps, which cause the centers to become dense. Thus, if we used such an algorithm as a subroutine, sofa would require more space. Here, however, we focused on setting close to the information-theoretically minimum space usage and, hence, we decided to use the algorithm by [44].

$LB \leftarrow 1$, $cost \leftarrow 0$ ▷ Process the vertices from the stream
while there exist unread vertices in U
  $C \leftarrow \emptyset$
  $f \leftarrow LB/(k(1+\log n))$
  for $u \leftarrow$ next vertex from stream
    $d \leftarrow \min_{c \in C} d(x_u, x_c)$
    openCenter $\leftarrow$ True, with probability $\min\{w(x) \cdot d/f, 1\}$, and False, otherwise
    if openCenter = True ▷ open u as center
      $C \leftarrow C \cup \{u\}$
      $w(u) \leftarrow 1$
    else ▷ Assign u to its closest center $c(u)$
      $cost \leftarrow cost + w(u) \cdot d$
      $c(u) \leftarrow \arg\min_{c \in C} d(x_u, x_{c(u)})$
      $w(c(u)) \leftarrow w(c(u)) + w(u)$
      $\mathsf{MG}(c(u)) \leftarrow \mathsf{MG}(c(u)) \cup \mathsf{MG}(u)$
    if $|C| = c_{\max}$ or $cost > 2LB$
      break and raise flag
  if flag raised
    $U \leftarrow$ the stream consisting of the (weighted) vertices in C and all unread vertices of U
    $LB \leftarrow 2LB$
$(C_1, \dots, C_k) \leftarrow$ clustering of C using an $O(1)$-approximate k-Medians algorithm
▷ Postprocessing
for all $i = 1, \dots, k$
  $\mathsf{MG}_i \leftarrow \bigcup_{x \in C_i} \mathsf{MG}(x)$
  $|C_i| \leftarrow \sum_{c \in C_i} w(c_i)$
  $\tilde{V}_i \leftarrow \{v \in V :$ the counter of v in MG_i is at least $\theta |C_i|\}$

Algorithm 3: sofa (U, k, $c_{\max}$, θ)

3.4 Second Pass: Recover Left Clusters

In this section, we present algorithms for computing a clustering $\tilde{U_1}, \dots, \tilde{U_k} \subseteq U$ of the left side of the graph during a second pass over the stream U. We assume that our algorithms obtain as input a set of clusters $\tilde{V_1}, \dots, \tilde{V_k} \subseteq V$ from the right side of the graph. We will present two different algorithms for biclustering and BMF, respectively.

3.4.1 Biclustering

We now present an algorithm which performs a single pass over the stream U and assigns each $u \in U$ to exactly one cluster $\tilde{U_i}$. We will use this algorithm for the biclustering problem, where each vertex $u \in U$ belongs to a unique planted cluster U_i (see Section 3.2.1).

To obtain the clustering $\tilde{U_1}, \dots, \tilde{U_k}$, the algorithm initially sets $\tilde{U_i} = \emptyset$ for all

$i = 1, \dots, k$. Now the algorithm performs a single pass over the stream of left-side vertices $u \in U$. For each u, let $\Gamma(u)$ denote the set of neighbors of u in V, i.e., $\Gamma(u) = \{v \in V : (u, v) \in E\} \subseteq V$. Now the algorithm assigns u to the cluster $\widetilde{U_{i^*}}$ such that the overlap of $\Gamma(u)$ and $\widetilde{V_{i^*}}$ is maximized relative to the size of $\widetilde{V_{i^*}}$. More concretely, the algorithm computes

$$i^* = \arg\max\{|\Gamma(u) \cap \tilde{V}_i|/|\tilde{V}_i| : i = 1, \dots, k\} \tag{3.2}$$

and then assigns u to $\widetilde{U_{i^*}}$.

Space Usage and Running Time. Observe that the algorithm uses space $O(m)$ (where $m = |U|$), since for each vertex $u \in U$, we need to store to which cluster U_i it was assigned. Furthermore, the running time of the algorithm is $O(mks)$: For each of the m vertices, we need to compute i^* as per Equation (3.2). Since we assume that each vertex u has at most $O(s)$ neighbors and that all $\tilde{V}_i$ have size $O(s)$, it takes time $O(s)$ to compute $|\Gamma(u) \cap \tilde{V}_i|/|\tilde{V}_i|$ for fixed i. Thus, computing i^* can be done in time $O(ks)$.

3.4.2 BMF

Next, we present an algorithm, which performs a single pass over the stream and computes clusters $\tilde{U}_1, \dots, \tilde{U}_k$, where every vertex $u \in U$ may be contained in multiple clusters $U_{i_1}, \dots, U_{i_T}$. Recall from Section 3.2.2 that this corresponds to computing a factor matrix L for the the BMF problem.

Our approach for computing the sets $\tilde{U}_i$ is similar to the greedy covering scheme used in [142]. The main idea is that for every $u \in U$, we greedily cover the set $\Gamma(u) \subseteq V$ using the clusters $\tilde{V}_1, \dots, \tilde{V}_k$ similar to the classic set cover problem. However, unlike in standard set cover, we do allow for some amount of "overcovering". Note that this greedily minimizes the symmetric difference of $\Gamma(u)$ and the sets $\tilde{V}_i$ used for covering $\Gamma(u)$; thus, also their Hamming distance is minimized.

Before we present our algorithm, let us first define our score function for the covering process. For sets A, X, Y, we define the *score of A for covering X given that Y was already covered* as *score*$(A \mid X, Y) = |(X \setminus Y) \cap A| - |A \setminus (X \cup Y)|$.

To better understand the score function, consider the case that no elements of X were covered before, i.e., $Y = \emptyset$. Then *score*$(A \mid X, \emptyset) = |X \cap A| - |A \setminus X|$ is the number of elements in X, which get covered by A, minus the number of those elements in A, which do not appear in X (these elements "overcover" X). Now suppose that $Y \neq \emptyset$, i.e., some elements of X were already covered before and these elements are stored in the set Y. Then the score function takes this into account by not adding score for elements in $A \cap X \cap Y$ that are in A and X, but were already covered before. Also, the score function does not subtract score for elements in A that are not in X, but which were already overcovered before (and, hence, are in Y); more precisely, it does not subtract score for the elements in $(A \cap Y) \setminus X$.

We now describe our greedy algorithm for computing the clusters $\tilde{U}_i$. Initially, we set $\tilde{U}_i = \emptyset$ for all i. Now we perform a single pass over the stream U and

for each $u \in U$, we do the following. We initialize $Y_u = \emptyset$ and, as before, let $\Gamma(u)$ denote the set of neighbors of u in V. Now, while there exists an i such that *score*$(\tilde{V}_i \mid \Gamma(u), Y_u) > 0$, we compute

$$i^* = \arg\max_{i=1,\dots,k} \textit{score}(\tilde{V}_i \mid \Gamma(u), Y_u). \tag{3.3}$$

If *score*$(\widetilde{V_{i^*}} \mid \Gamma(u), Y_u) > 0$, we assign u to $\widetilde{U_{i^*}}$ and we set $Y_u = Y_u \cup \widetilde{V_{i^*}}$. Otherwise, we stop covering u and proceed with the next vertex from the stream.

Space Usage and Running Time. The space usage is $O(km)$ since each vertex can be assigned to as many as k clusters. The running time of the algorithm is $O(mk^2 s)$: First, note that evaluating *score*$(\tilde{V}_i \mid \Gamma(u), Y_u)$ takes time $O(s)$ because all sets have size $O(s)$. Second, for a single iteration of the while-loop we need to evaluate the score function $O(k)$ times to obtain i^* and there are at most k iterations. Hence, we need to spend time $O(k^2 s)$ for each of the m vertices in U.

3.5 Implementation

We implemented the sofa algorithm from Section 3.3.2 for recovering the right-side clusters and the two algorithms from Section 3.4 for recovering the left-side clusters. In this section, we present certain adjustments that we made to improve the results of the algorithms and we discuss how to set certain parameters of the algorithms.

We implemented all algorithms in Python. To speed up the computation, the subroutines for finding the closest centers (Line 6 in Algorithm 3) and for finding the clusters with maximum score (Equation (3.3)) were implemented in CPython. We did not use any parallelization, i.e., our implementations are purely single-threaded.

Our code will be available online and when the final version of [153] gets published.

3.5.1 Asymmetric Weighted Hamming Distance

During preliminary tests of sofa on real-world data, we realized that sofa picked extremely sparse centers which often only had a single non-zero entry. This resulted in almost all vertices being assigned to this particular center (because the Hamming distance of a vertex u to a center with a single non-zero entry is the degree of u plus/minus 1 and, due to the low degrees of the left-side vertices u, these distances are usually small) which made the cluster recovery fail.

Hence, we needed to find a way to promote denser centers. To this end, we introduce an asymmetric weighted version of the Hamming distance which we define as follows. Let $c \in C$ be a center maintained by sofa and let u be a vertex which needs to be clustered. For each entry i of x_c and x_u, we assign the following costs: If $x_c(i) = x_p(i)$, then the cost is 0; if $x_p(i) = 1$ and $x_c(i) = 0$ then the cost is 1; if $x_p(i) = 0$ and $x_c(i) = 1$ then the cost is $\alpha < 1$. Now the *asymmetric weighted Hamming distance* of c and p is simply the sum over the costs for all entries of x_c and x_p.

Note that by setting $\alpha = 1$ the above results in the classic (symmetric) Hamming distance. Furthermore, setting $\alpha < 1$ promotes denser centers because the case of $x_c(i) = 1$ and $x_u(i) = 0$ is penalized less than in classic Hamming distance.

For example, consider the vectors $x_{c_1} = (1, 1, 1, 1, 0)$, $x_{c_2} = (0, 0, 0, 0, 1)$ and $x_u = (1, 0, 0, 0, 0)$. In vanilla Hamming distance, u would be assigned to c_2 since their distance is 2 and the distance of c_1 and p is 3. With asymmetric weighted Hamming distance and $\alpha = 0.1$, u is assigned to c_1 because their distance is 0.3 and the distance is u and c_2 is 1.1. Note the assignment of u to c_1 instead of c_2 is also much more suitable for the thresholding step in Line 25 of sofa.

In practice, our experiments showed that setting $\alpha = 0.1$ was a good choice for all datasets and the performance of our algorithms benefitted heavily from using asymmetric weighted Hamming distance.

3.5.2 Biclustering Algorithm

To solve the biclustering problem from Section 3.2.1, we implemented sofa (Algorithm 3) together with the biclustering algorithm from Section 3.4.1 for recovering the left clusters. The only adjustment that we made was to use the k-Means implementation of scikit-learn [166] in order to implement the $O(1)$-approximate k-Medians algorithm in Line 21 of sofa.

3.5.3 BMF Algorithm

To solve the BMF problem from Section 3.2.2, we implemented sofa (Algorithm 3) together with the BMF algorithm from Section 3.4.2 for recovering the left clusters.

During preliminary tests we observed that on some datasets we achieved better results when we completely skipped the k-Median algorithm in Line 21 of sofa. Instead, we compute a cluster $\tilde{V}_c$ for each center $c \in C$. Note that this might lead to more than k clusters $\tilde{V}_c$ but to at most $c_{\max}$. Then we use the BMF algorithm from Section 3.4.2 to compute a cluster $\tilde{U}_c$ for each of the (potentially more than k) clusters $\tilde{V}_c$. While computing the clusters $\tilde{U}_c$, we keep track of the total score of each cluster $\tilde{V}_c$; this can be done by maintaining a counter s_c for each $c \in C$ and increasing s_c by $\mathit{score}(\tilde{V}_c \mid \Gamma(u), Y_u)$ whenever we compute i^* in Equation (3.3). To ensure that our algorithm only returns k clusters when it finishes, we sort the clusters $\tilde{V}_c$ by their score values s_c in non-increasing order and only keep the k clusters with the highest total scores. This ensures that at the end we only return k clusters.

While sofa and the algorithm from Section 3.4.2 return clusters $\tilde{U}_i$ and $\tilde{V}_i$ instead of Boolean factor matrices L and R as required for the BMF problem, we can transform the clusters into factor matrices L and R as discussed in Section 3.2.2. This gives raise to a matrix $\tilde{B} = L \circ R$ which approximates the biadjacency matrix B of the input graph G.

3.5.4 Setting the Rounding Threshold θ

Next, we discuss how to set the rounding threshold θ.

A Heuristic for Determining θ. In Section 2.4.3, we presented a heuristic for setting θ. It essentially works by observing that θ is a function of the parameters p and q of the random graph model from Section 3.2.1. Then it performs a grid search over different values of p and q and picks the pair (p^*, q^*) for which the resulting rounding threshold θ^* maximizes the likelihood of the counters observed in the heavy hitters data structure from Line 23 of sofa. We refer to Section 2.4.3 for the details of the heuristic. We will refer to the version of sofa which uses this heuristic as sofa-auto.

Using Multiple Thresholds. Note that the only place in sofa, where the rounding threshold θ is used, is in the postprocessing step. Thus, given multiple rounding thresholds $\theta_1, \dots, \theta_T$, it is possible to compute a set of clusters $\tilde{V}_1^{(t)}, \dots, \tilde{V}_k^{(t)}$ for each θ_t. Then for each $t = 1, \dots, T$, we can compute corresponding left-side clusters $\tilde{U}_1^{(t)}, \dots, \tilde{U}_k^{(t)}$ using the algorithms from Section 3.4. Note that computing the clusters $\tilde{U}_i^{(t)}$ for all values of $t = 1, \dots, T$ still only requires a single pass over the stream: For each $u \in U$ of the stream, we can run the algorithms for computing $\tilde{U}_1^{(t)}, \dots, \tilde{U}_k^{(t)}$ in parallel for all $t = 1, \dots, T$.

In our experiments we will use the above strategy to generate clusters for multiple thresholds. Then we will evaluate their quality in a separate postprocessing step (see Section 3.6.2). We will refer to the version of sofa which uses multiple thresholds simply as sofa.

3.5.5 Static to Streaming Reduction

Since many static algorithms do not scale to datasets of the size considered in this chapter, we describe a reduction for turning *static* biclustering/BMF algorithms into *2-pass streaming* algorithms. We will use this reduction to compare sofa against static algorithms in our experiments.

In a nutshell, the reduction works as follows. First, we sample a subgraph with $\tilde{m}$ left-side vertices and $\tilde{n}$ right-side vertices, where $\tilde{m} \ll m$ and $\tilde{n} \ll n$ are parameters of the reduction. Then we run the static algorithm on the sampled subgraph to determine a set of right-side clusters $\tilde{V}_1, \dots, \tilde{V}_k$ (see below for details). In the second pass over the stream, we use exactly the same procedure as used by sofa (see Section 3.4) to infer the left-side clusters $\tilde{U}_1, \dots, \tilde{U}_k$.

Now, we elaborate on the first pass over the stream. First, we use reservoir sampling to obtain $\tilde{m}$ left-side vertices from the graph uniformly at random; let $U' = \{u'_1, \dots, u'_{\tilde{m}}\}$ denote this set of left-side vertices. Let V' be the set of right-side vertices which are adjacent to vertices in U'. Note that possibly $|V'| > \tilde{n}$ and let V'' be the set of $\tilde{n}$ vertices in V' with highest degree to vertices in U' (breaking ties arbitrarily). Now we run the static algorithm on the subgraph with the $\tilde{m}$ left-side vertices U' and $\tilde{n}$ right-side vertices V''. This gives raise to clusters $\tilde{V}_1, \dots, \tilde{V}_k$. Next, we add the (low-degree) vertices $v \in V' \setminus V''$ to the clusters $\tilde{V}_i$ by

assigning each v to the cluster $\tilde{V}_i$ which "on average" has the most similar left-side neighborhood compared to v. More concretely, for each vertex $v \in V'$ we define the vector $x_v \in \{0,1\}^{\tilde{m}}$ such that $x_v(i) = 1$ iff $(u'_i, v) \in E$. Next, for each cluster $\tilde{V}_i$ define the vector $x_i = \sum_{v \in \tilde{V}_i} x_v / |\tilde{V}_i|$ which describes the "average left-side neighborhood" of the vertices in $\tilde{V}_i$. Now we assign each $v \in V' \setminus V''$ to $\tilde{V}_{i^*}$ with $i^* = \arg\min_i d(x_i, x_v)$. This yields the final clusters $\tilde{V}_1, \dots, \tilde{V}_k$.

3.6 Experiments

We evaluate sofa on synthetic and on real-world datasets. We conducted the experiments on a workstation with 4 Intel i7-3770 processors at 3.4 GHz and 16 GB of main memory.

3.6.1 Synthetic Datasets

We start by evaluating our biclustering version of sofa from Section 3.5.2 on synthetic data. We ran sofa with different numbers of centers $c_{\max} \in \{100, 200\}$ and with 100 and 200 counters in the heavy hitters data structures.

We compare sofa against three different algorithms. First, a version of the algorithm from [150] which does not use any spectral preprocessing; this algorithm is denoted static sofa. static sofa can be viewed as a non-streaming version of sofa, i.e., it performs the clustering offline using k-Means (instead of streaming k-Median) and then it performs the thresholding step (Line 25) using the exact frequency counts (instead of the approximate frequency counts from the heavy hitters data structures). Thus, static sofa essentially provides an upper bound on how good the streaming version of sofa can potentially get. Next, we turn the static biclustering algorithms by Dhillon [63] and Zha et al. [218] into streaming algorithms via the reduction from Section 3.5.5, where we set $\tilde{m} = \tilde{n} = 5000$, i.e., we sample subgraphs with 5000 vertices on both sides. We denote these algorithms RSdhillon and RSzhaEtAl, where RS stands for *random subgraph*.

Data Generation and Quality Measure. We generated the synthetic data as follows. We start with an empty graph and then for each ground-truth cluster U_i, we insert ℓ vertices (see below for which values of ℓ we used in the experiments). Then we inserted 8000 vertices on the right side of the graph (i.e., $|V| = n = 8000$). To generate the ground-truth clusters V_i, we simply picked r vertices uniformly at random from V for each i (see below for how r was set in the experiments). Now the random edges were inserted exactly as described in the random graph model from Section 3.2.1.

When not mentioned otherwise, we have set the parameters for the graph generation as follows: $n = 8000$, $k = 50$, $\ell = 200$ (and, hence, $|U| = m = k \cdot \ell = 10\,000$), $p = 0.7$, $r = 30$. Furthermore, we set q such that in expectation every left-side vertex obtains 20 random neighbors.

To evaluate the output of the algorithms, let $U_1, \dots, U_k$ be the planted ground-truth clusters and let $\tilde{U}_1, \dots, \tilde{U}_k$ be the clusters returned by one of the algorithms. We define the *quality* Q of the clustering $\tilde{U}_1, \dots, \tilde{U}_k$ as

$$Q = \frac{1}{k} \sum_{i=1}^{k} \max_{j=1,\dots,s} J(U_i, \tilde{U}_j) \in [0, 1],$$

where $J(A, B) = |A \cap B|/|A \cup B|$ is the Jaccard coefficient. That is, for each ground-truth cluster U_i, we find the cluster $\tilde{U}_j$ which maximizes the Jaccard coefficient of U_i and $\tilde{U}_j$. The quality is then simply the sum over the Jaccard coefficients for all ground-truth clusters U_i, normalized by k. Clearly, higher values for Q imply a clustering closer to the planted clustering. For example, if the clusters $\tilde{U}_j$ match *exactly* the ground-truth clusters U_i then $Q = 1$. We evaluate the quality of the clusters $\tilde{V}_i$ in exactly the same way.

Experiments. Next, let us discuss the outcomes of our experiments in different scenarios, where each time we vary one of the parameters. For each set of parameters we generated 15 different datasets and we will be reporting averages and standard deviations for the recovery quality of the algorithms. Our results are reported in Figure 3.1.

Varying Amount of Signal. First, let us consider a varying amount of signal, i.e., we set $p \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$. One can see in Figures 3.1a and 3.1b that the quality of all sofa-versions improves as p increases. Furthermore, static sofa achieves the best quality for recovering the left and right clusters. The second-best sofa-version is sofa with 200 counters and 200 centers and achieves between 0.05 and 0.1 less quality than static sofa; we ran significance tests and these differences are significant. When only providing 100 centers, sofa has some problems for values $p \in \{0.5, 0.6\}$; this is not surprising since we planted 50 clusters and thus only maintaining 100 centers is quite restrictive for sofa. The right-side recovery of RSdhillon and RSzhaEtAl is relatively constant, where RSdhillon is performing on a high level; we explain the flatness of the curves by the spectral methods used in the algorithms, which "denoise" the data well even for small p. The left-side recovery of both algorithms is clearly worse than those of the sofa-versions. Regarding the running times (Figure 3.1c), we see that all versions of sofa are about a factor 3 faster than static sofa; note that sofa with 100 centers is also significantly faster than the versions of sofa with 200 centers. RSdhillon and RSzhaEtAl are about factor 1.5–2 slower than sofa.

Varying Size of Right Clusters. Next, we varied the sizes $r \in \{15, 20, 30, 50\}$ of the planted right clusters V_i. We can see (Figures 3.1d and 3.1e) that most algorithms benefit from larger r and that once again static sofa is the best method, followed by sofa with 200 counters and 200 centers. When the right clusters are very small (sizes 15, 20), sofa is much worse than static sofa and RSdhillon. Indeed, for small values of r, the vertices become much harder to cluster for sofa, because the Hamming distances of the vertices get dominated by noise. However, for $r \geq 30$, the version of sofa with 200 counters and 200 centers only has a 0.1 gap in quality com-

pared to static sofa. Furthermore, observe that the performance of sofa with only 100 counters in the heavy hitters data structures drops dramatically for $r = 50$; this is caused by the frequency estimations of the right-side vertices getting too inaccurate due to the too small number of counters in the heavy hitters data structures. RSdhillon's quality is again relatively constant at roughly the same level as before, while RSzhaEtAl clearly benefits from larger cluster sizes. The running times of the algorithms (Figure 3.1f) slightly rise as r increases since the datasets contain more non-zero entries.

Varying Size of Left Clusters. Finally, we varied the size ℓ of the left clusters U_i and set $\ell \in \{100, 150, 200, 300, 400, 500, 600\}$. Note that this implies that we are also varying the number of left-side vertices of the bipartite graph and, hence, also the total number of edges in the graph. Figures 3.1g and 3.1h show that the recovery quality is relatively unaffected from this change in ℓ and that the ranking of the algorithms is as before. However, note that the running times of static sofa increase much more rapidly than those of the streaming algorithms. For example, for $\ell = 100$ the running times of sofa and static sofa differ by a factor of less than 2 but for $\ell = 600$ this is already approximately 7.

Conclusion. We conclude that sofa can achieve recovery qualities close to the static baseline even when its number of centers is only $4k$ and its number of counters is within factor 4 of the size of the right-side clusters. Furthermore, sofa's runtime scales much better than the static baseline's. While RSdhillon delivered good quality for right-side recovery, its left-side recovery was rather poor. RSzhaEtAl performs badly overall; we blame this on the data being too sparse, which does not allow the algorithm to find good cuts.

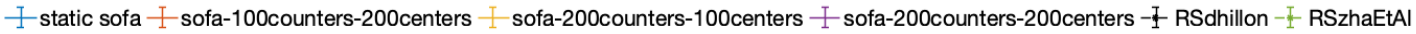
3.6.2 Real-World Datasets

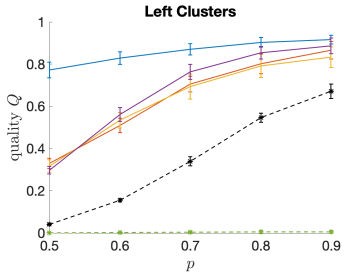
For the real-world experiments, it is more realistic to allow the left-side clusters U_i to overlap. Thus, for the real-world experiments, we use the version of sofa which solves the BMF problem from Section 3.5.3.

Methods and Measures. For these experiments, we use sofa and sofa-auto. For sofa, we set the threshold θ using a line search and we use the values $\theta \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$. The remaining parameters were set as follows: $c_{\max} = 20k$, where k is the desired number of clusters; $s = P_{99}$, the 99th quantile of the degrees on the left-side vertices (see Table 3.1 for the values for each dataset); and we set the number of counters in the heavy hitters data structures to $\max\{3s, 0.05n\}$.

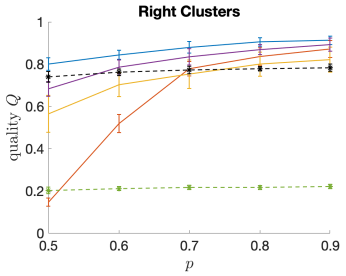
As for the synthetic datasets, we compare sofa against RSdhillon and RSzhaEtAl. We used $\tilde{m} = \tilde{n} = 15000$ in the reduction. With these parameters, RSdhillon and RSzhaEtAl have running times comparable to sofa and already for $\tilde{m} = \tilde{n} = 20000$, our workstation would often run out of memory. Further, we compare against the static (i.e., non-streaming) algorithm basso[2], which is an efficient implementation of the asso algorithm [142]. basso has one hyperparameter, τ. We try values

[2]basso v0.5 from `http://cs.uef.fi/~pauli/basso/`

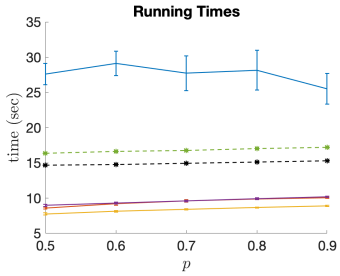




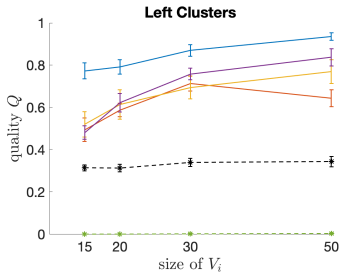
(a) Vary p: Left Cluster Quality



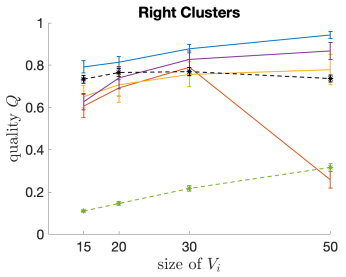
(b) Vary p: Right Cluster Quality



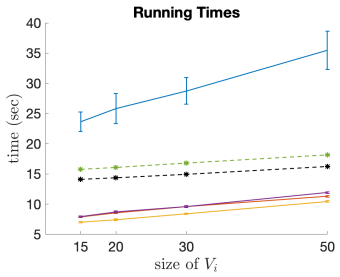
(c) Vary p: Running times (sec)



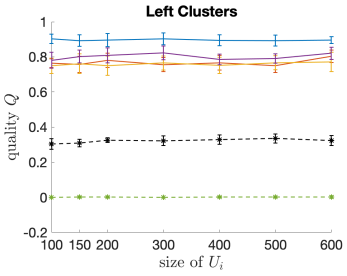
(d) Vary $|V_i|$: Left Cluster Quality



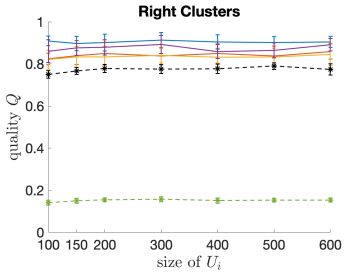
(e) Vary $|V_i|$: Right Cluster Quality



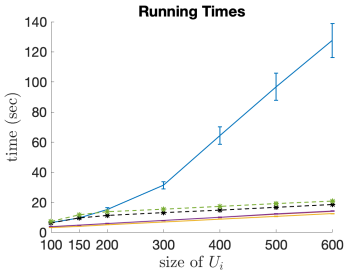
(f) Vary $|V_i|$: Running times (sec)



(g) Vary $|U_i|$: Left Cluster Quality



(h) Vary $|U_i|$: Right Cluster Quality



(i) Vary $|U_i|$: Running times (sec)

Figure 3.1: Results on synthetic data. Figures 3.1a–3.1c have varying p, Figures 3.1d–3.1f have varying sizes of the right clusters V_i, Figures 3.1g–3.1i have varying sizes of the left clusters U_i. Markers are mean values over 15 different datasets; error bars are one standard deviation over the 15 datasets.

$\tau \in \{0.2, 0.4, 0.6, 0.8\}$ and report the results with the best value. For run-time and memory usage analysis, we report average values over different thresholds. The time complexity of basso is $O(k|U|^2|V|)$ and thus we flipped U and V in the input for basso when $|U| > |V|$.

For all datasets, we computed clusterings consisting of $k = 50, 100, 200$ clusters. Since for the real-world datasets no information about the ground-truth clusters is

Table 3.1: Real-world dataset properties. Datasets are considered as bipartite graphs $G = (U \cup V, E)$ and density is $|E|/(|U| \cdot |V|)$. Average degree $\overline{\deg}$ and the 99th percentile degree P_{99} are calculated from U and rounded to the nearest integer.

Dataset	$\lvert U\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	density	$\overline{\deg}$	P_{99}
20News	18 773	61 056	1 766 780	0.0015	94	548
Reuters	38 677	19 757	978 446	0.0013	25	498
Book	105 282	340 550	1 149 779	< 0.0001	11	174
Movie	138 493	26 744	20 000 263	0.0054	144	1113
Flickr	395 979	103 631	8 545 307	0.0002	22	268
Wiki	1 562 433	1 170 854	19 753 078	< 0.0001	17	177

available, we use relative Hamming gain and recall as quality measures to evaluate the obtained clusterings. Formally, let B be the biadjacency matrix of the bipartite graph and let $\tilde{B}$ an approximation thereof. The *relative Hamming gain* is defined as $1 - |\{(i,j) : B_{ij} \neq \tilde{B_{ij}}\}|/|\{(i,j) : B_{ij} = 1\}|$, and it indicates how much better $\tilde{B}$ approximates B than a trivial (all-zeros) matrix would. The *recall* is defined as $|\{(i,j) : B_{ij} = 1 \wedge \tilde{B_{ij}} = 1\}|/|\{(i,j) : B_{ij} = 1\}|$, and it indicates the fraction of edges (1s) in B which are "covered" correctly by the matrix $\tilde{B}$ returned by one of the algorithm.

Explanation of Datasets. In our experiments, we used six real-world datasets. Their basic properties are described in Table 3.1. Notice that all datasets are very sparse, and their left-side degrees (even in the 99th percentile) are small compared to the number of vertices on the right side of the graph. This empirically validates two of the three properties we discussed in the introduction.

Let us briefly discuss each of the datasets. 20News[3] contains newsgroup postings on the left side and words on the right side; edges indicate a word appearing in a posting. The datasets Reuters and Flickr were taken from the KONECT[4] [123] website. Reuters has articles from the news organization Reuters on the left side and words on the right. Flickr encodes the group memberships (right) of Flickr users (left). Wiki[5] is from the SuiteSparse Matrix Collection[61] and consists of Wikipedia pages on both sides of the graph; an edge (u, v) indicates that page u links to page v (note that this relationship is asymmetric). Book[6][221] is a rating matrix consisting of users on the left side and books on the right side; an edge indicates that a user rated book. Movie[7] is a rating matrix between users and movies [95].

Experiments. Results for relative Hamming gain and recall are presented in Tables 3.2 and 3.3, respectively. Note that basso did not finish on the Wiki dataset,

[3] http://qwone.com/~jason/20Newsgroups/

[4] http://konect.uni-koblenz.de

[5] https://www.cise.ufl.edu/research/sparse/matrices/Gleich/wikipedia-20051105

[6] http://www2.informatik.uni-freiburg.de/~cziegler/BX/

[7] https://grouplens.org/datasets/movielens/20m/

Table 3.2: Relative Hamming gain different real-world datasets

k	Algorithm	Relative Hamming gain					
		20News	Reuters	Book	Movie	Flickr	Wiki
50	sofa-auto	0.0298	0.0450	0.0198	0.0805	0.0380	0.0617
	sofa	0.0424	0.0454	0.0212	0.1188	0.0453	0.0695
	basso	0.0545	0.1005	0.1226	0.1394	0.0719	–
	RSdhillon	0.0042	0.0273	0.0008	0.1056	0.0040	0.0001
	RSzhaEtAl	0.0001	0.0274	0.0008	0.0297	0.0000	0.0000
100	sofa-auto	0.0411	0.0792	0.0298	0.1028	0.0486	0.0730
	sofa	0.0574	0.0777	0.0333	0.1367	0.0668	0.0824
	basso	0.0793	0.1097	0.1783	0.1739	0.1068	–
	RSdhillon	0.0059	0.0307	0.0028	0.1378	0.0137	0.0262
	RSzhaEtAl	0.0006	0.0342	0.0030	0.0696	0.0000	0.0000
200	sofa-auto	0.0624	0.1253	0.0427	0.1247	0.0663	0.0861
	sofa	0.0930	0.1254	0.0472	0.1598	0.0817	0.1061
	basso	0.1171	0.1334	0.2531	0.2376	0.1556	–
	RSdhillon	0.0092	0.0402	0.0024	0.1771	0.0203	0.0270
	RSzhaEtAl	0.0014	0.0291	0.0017	0.1104	0.0007	0.0001

because it ran out of memory.

The results for relative Hamming gain in Table 3.2 show that, when it is able to finish, basso is always the best method. This is to be expected as it can make unlimited passes over the data. On all datasets except Book and for all values of k, the results of sofa and basso are within factor at most 2.2. For $k = 200$, the results of sofa are at most 50% worse than those of basso on 20News, Reuters and Movie. With Book, on the other hand, sofa is significantly worse (up to factor 5.8) but still much better than RSdhillon and RSzhaEtAl. We believe this results from Book being too sparse; indeed, the 50% percentile of the degrees of the left vertices in book is 1 and thus sofa's clustering seems to fails. Overall, the results of sofa and sofa-auto improve significantly as k increases, showing that it can be used for small and large values of k alike. RSdhillon and RSzhaEtAl perform well when $|V|$ is small (e.g., Movie and Reuters), but as soon as $|V|$ increases, their results decays dramatically (e.g., Book, Flickr and Wiki); this appears to be a limitation of the random sampling approach.

The results concerning the recall in Table 3.3 look very similar to relative Hamming gain: For all datasets except Book, sofa has approximately 50% of the recall of basso, and in Book it is again significantly worse. For Wiki, sofa has results that are comparable to other datasets, thus, the size of Wiki datasets does not seem to affect the quality. For RSdhillon and RSzhaEtAl we observe a similar behavior as above.

Using the heuristic in sofa-auto to set the threshold typically leads to slightly worse results than setting it using line search. Given that the heuristic is usually 3–4 times as fast, there seems to be a tradeoff which version one should pick.

Table 3.3: Recall in different real-world datasets

k	Algorithm	Recall					
		20News	Reuters	Book	Movie	Flickr	Wiki
50	sofa-auto	0.0446	0.0649	0.0201	0.1262	0.0480	0.0657
	sofa	0.0483	0.0652	0.0214	0.1779	0.0474	0.0700
	basso	0.0683	0.1677	0.1226	0.2855	0.0760	–
	RSdhillon	0.0069	0.0316	0.0009	0.1999	0.0088	0.0001
	RSzhaEtAl	0.0004	0.0447	0.0014	0.0614	0.0001	0.0000
100	sofa-auto	0.0570	0.0991	0.0307	0.1597	0.0636	0.0777
	sofa	0.0649	0.0987	0.0341	0.2030	0.0721	0.0840
	basso	0.0959	0.1907	0.1783	0.3143	0.1124	–
	RSdhillon	0.0103	0.0430	0.0060	0.2400	0.0246	0.0302
	RSzhaEtAl	0.0017	0.0500	0.0040	0.1182	0.0002	0.0000
200	sofa-auto	0.0788	0.1441	0.0435	0.1926	0.0837	0.0924
	sofa	0.0991	0.1442	0.0479	0.2353	0.0906	0.1087
	basso	0.1321	0.2100	0.2532	0.3521	0.1603	–
	RSdhillon	0.0159	0.0619	0.0030	0.2812	0.0317	0.0299
	RSzhaEtAl	0.0022	0.0454	0.0027	0.1644	0.0021	0.0002

The running times of the algorithms are presented in Table 3.4. For sofa and sofa-auto, presented is the total running time (with full line search in sofa); for basso, the presented time is the *average time for a single value* of the threshold parameter τ. Still, basso is consistently the slowest method, often by orders of magnitude. The run-times of RSdhillon and RSzhaEtAl scale well in k, since the size of the sampled subgraph and, hence, the time spent on the static computation, is largely unaffected by the choice of k.

The memory usages of the algorithms are presented in Table 3.5. basso again needs significantly more resources. sofa and sofa-auto can compute clusterings of graphs with millions of vertices and edges, while never using more than 500 MB of RAM. RSdhillon and RSzhaEtAl have relatively large memory footprints (using gigabytes of memory) due to the spectral methods they use.

Overall, the real-world experiments show that sofa can achieve results that are not too far from a static baseline method, while using only a fraction of resources.

3.7 Theoretical Guarantees

We prove the theoretical guarantees of our algorithms.

3.7.1 Proof of Theorem 3.1

For all proofs we assume that the conditions from Theorem 3.1 hold. The concrete values of the constants K_j are set inside the proofs. We start by characterising the

Table 3.4: Algorithm run-time on different real-world datasets

k	Algorithm	Run-time in CPU minutes					
		20News	Reuters	Book	Movie	Flickr	Wiki
50	sofa-auto	2.1	3.2	1.7	45.9	9.7	14.1
	sofa	6.2	10.3	5.5	120.0	24.0	42.9
	basso	22.7	13.2	2951.8	598.1	4667.8	–
	RSdhillon	28.1	23.1	16.4	27.8	21.0	49.7
	RSzhaEtAl	36.0	75.2	75.4	35.9	98.5	76.3
100	sofa-auto	5.2	8.3	4.7	102.2	19.9	25.8
	sofa	15.6	25.4	16.5	311.6	52.7	70.4
	basso	24.6	13.6	3003.8	932.3	5066.0	–
	RSdhillon	26.9	23.7	18.1	31.2	23.0	55.5
	RSzhaEtAl	41.6	81.2	80.7	39.7	172.3	63.7
200	sofa-auto	12.2	34.8	14.2	229.1	63.7	57.1
	sofa	43.5	142.8	60.4	959.0	161.4	157.5
	basso	26.7	14.3	3097.4	1441.2	5574.1	–
	RSdhillon	25.3	23.1	20.8	42.2	25.8	68.3
	RSzhaEtAl	39.4	90.0	68.6	51.5	350.8	100.9

distances of vertices from the same cluster U_i and vertices from different clusters U_i and $U_{i'}$.

Lemma 3.3. *Let $u, u' \in U_i$ and let $u'' \in U_{i'}$ for $i' \neq i$. Then with probability at least $1 - m^{-3}$,*

$$\begin{aligned} d(x_u,x_{u'}) &< 1.01\left[2|V_i|p(1-p) + 2(|V \setminus V_i|)q(1-q)\right], \\ d(x_u,x_{u''}) &> 0.99[|V_i \triangle V_{i'}|(p(1-q) + q(1-p)) \\ &\quad + 2|V_i \cap V_{i'}|p(1-p) + 2|V \setminus (V_i \cup V_{i'})|q(1-q)]. \end{aligned}$$

Proof. First, recall that the neighbors of u, u' and u'' are random variables such that if $u \in U_i$ then $\mathbf{Pr}\left((u, v_j) \in E\right) = p$, if $v_j \in V_i$, and $\mathbf{Pr}\left((u, v_j) \in E\right) = q$, if $v_j \in V \setminus V_i$. Since u's neighbors are random this implies that the vector x_u is a random vector with $\mathbf{Pr}\left(x_u(j) = 1\right) = \mathbf{Pr}\left((u, v_j) \in E\right)$. Next, observe that we can rewrite the event $\{x_u(j) \neq x_{u'}(j)\}$ as $\{x_u(j) = 1 \text{ and } x_{u'}(j) = 0\} \cup \{x_u(j) = 0 \text{ and } x_{u'}(j) = 1\}$. Together, this implies for vertices from the same cluster,

$$\mathbf{Pr}\left(x_u(j) \neq x_{u'}(j)\right) = \begin{cases} 2p(1-p), & v_j \in V_i, \\ 2q(1-q), & v_j \in V \setminus V_i. \end{cases}$$

Similarly, we obtain for vertices from different clusters,

$$\mathbf{Pr}\left(x_u(j) \neq x_{u''}(j)\right) = \begin{cases} p(1-q) + q(1-p), & v_j \in V_i \triangle V_{i'}, \\ 2p(1-p), & v_j \in V_i \cap V_{i'}, \\ 2q(1-q), & v_j \notin V_i \cup V_{i'}. \end{cases}$$

Table 3.5: Algorithm memory usage on different real-world datasets

k	Algorithm	Memory in GB					
		20News	Reuters	Book	Movie	Flickr	Wiki
50	sofa-auto	0.15	0.12	0.10	0.24	0.21	0.20
	sofa	0.16	0.13	0.10	0.24	0.20	0.22
	basso	0.40	0.66	10.81	1.80	11.48	–
	RSdhillon	8.95	8.70	6.12	8.99	7.16	5.61
	RSzhaEtAl	10.72	10.43	7.26	10.73	8.63	6.57
100	sofa-auto	0.19	0.14	0.11	0.33	0.27	0.30
	sofa	0.20	0.17	0.13	0.33	0.26	0.30
	basso	0.40	0.67	10.95	1.80	11.79	–
	RSdhillon	8.96	8.70	6.09	8.99	7.20	5.54
	RSzhaEtAl	10.71	10.40	7.26	10.73	8.58	6.63
200	sofa-auto	0.25	0.18	0.13	0.49	0.36	0.43
	sofa	0.26	0.22	0.17	0.50	0.36	0.42
	basso	0.40	0.67	10.99	1.80	12.22	–
	RSdhillon	8.96	8.68	6.00	8.98	7.18	5.57
	RSzhaEtAl	10.72	10.46	7.30	10.73	8.54	6.63

Next, using linearity of expectation we get that

$$\begin{aligned}\mathbf{E}\left[d(x_u, x_{u'})\right] =& \sum_{j=1}^{n} \mathbf{Pr}\left(x_u(j) \neq x_{u'}(j)\right)\\ =& 2|V_i|p(1-p) + 2(|V \setminus V_i|)q(1-q),\\ \mathbf{E}\left[d(x_u, x_{u''})\right] =& |V_i \triangle V_{i'}|(p(1-q) + q(1-p))\\ &+ 2|V_i \cap V_{i'}|p(1-p)\\ &+ 2|V \setminus (V_i \cup V_{i'})|q(1-q).\end{aligned}$$

Since $|V_i| \geq K_3 \log n$ and $|V_i \triangle V_{i'}| \geq K_4 s \geq K_3 K_4 \log n$,

$$\begin{aligned}\mathbf{E}\left[d(x_u, x_{u'})\right] &\geq 2p(1-p)|V_i| \geq 2K_3 p(1-p)\log n,\\ \mathbf{E}\left[d(x_u, x_{u''})\right] &\geq |V_i \triangle V_{i'}|(p(1-q) + q(1-p))\\ &\geq K_4 p(1-q)s \geq K_3 K_4 p(1-q)\log n.\end{aligned}$$

A Chernoff bound and setting K_3 large enough implies the lemma (we will set K_4 later independently of K_3). □

Next, we show that when setting $\alpha = 0.49 K_4 s$ in Algorithm 2, the algorithm clusters all left-side vertices correctly.

Lemma 3.4. *The following events hold w.h.p.: (1) When Algorithm 2 finishes, $|C| = k$ and for all i, C contains exactly one center c with $c \in U_i$. (2) For all i, there exists a center $c_i \in C$ s.t. all points $u \in U_i$ were assigned to c_i.*

Proof. First, we condition on the event from Lemma 3.3 occurring for each pair of vertices from U for the rest of the proof. A union bound implies that this happens with probability at least $1 - m^{-1}$.

Second, consider $u, u' \in U_i$. Then

$$\begin{aligned} d(x_u, x_{u'}) &< 1.01\left[2sp(1-p) + 2nq(1-q)\right] \\ &\leq 1.01\left[s/2 + 2n\frac{K_1 s}{n}\right] \leq 1.01(1/2 + 2K_1)s, \end{aligned}$$

where we used $p(1-p) \leq 1/4$ and $q \leq K_1 ps/n \leq K_1 s/n$.

Third, for $u \in U_i$ and $u'' \in U_{i'}$ for $i \neq i'$,

$$\begin{aligned} d(x_u, x_{u''}) &> 0.99[|V_i \triangle V_{i'}|(p(1-q) + q(1-p)) \\ &\quad + 2|V_i \cap V_{i'}|p(1-p) + 2|V \setminus (V_i \cup V_{i'})|q(1-q)] \\ &\geq 0.99[K_4 sp(1-q) + 0 + 0] \geq 0.98K_4 s/2, \end{aligned}$$

where we used that $|V_i \triangle V_{i'}| \geq K_4 s$ and further $p(1-q) \geq p - K_1 p^2 s/n \geq p - K_1 p^2 \geq \frac{0.98}{0.99} \cdot \frac{1}{2}$, since $p \geq 1/2$ and since we can pick K_1 small enough to satisfy the last inequality.

Pick K_1, K_4 with $K_4 \geq \frac{2.02}{0.98}(1/2 + 2K_1)$. Then $d(x_u, x_{u''}) > 0.98K_4 s/2 \geq 1.01(1/2 + 2K_1)s > d(x_u, x_{u'})$.

Next, we show that Algorithm 2 satisfies the properties of the lemma with $\alpha = 0.98K_4 s/2$. To prove (1), suppose a vertex $u \in U_i$ is processed and for all $c \in C$, $d(x_u, x_c) > \alpha$. Then C cannot contain any point $u' \in U_i$ (if C contained such a point, then the previous computation and the event we conditioned on imply $d(x_u, x_{u'}) \leq \alpha$). Thus, opening u as a new center is the correct choice and C contains exactly one center from U_i. To prove (2), suppose that a vertex $u \in U_i$ is processed and $d(x_u, x_c) \leq \alpha$ for some $c \in C$. The previous computation and the event we conditioned on imply that $c \in U_i$. Thus, all $u \in U_i$ are assigned to the same $c \in C$. □

Next, we show that all left-side vertices have degree $O(s)$.

Lemma 3.5. *With probability at least $1 - n^{-2}$, each vertex $u \in U$ has degree $O(s)$.*

Proof. Let $u \in U_i$ and let $d(u)$ be the degree of u. Then we get that $\mathbf{E}[d(u)] = p|V_i| + q|V \setminus V_i| \leq ps + (K_1 s/n)n = O(s)$. Since $\mathbf{E}[d(u)] \geq p|V_i| \geq K_3 p \log n$, we can apply a Chernoff bound to obtain that for large enough K_3 it holds that $d(u) \in [0.99\mathbf{E}[d(u)], 1.01\mathbf{E}[d(u)]]$ with probability at least $1 - n^{-2}$. □

Now we show that Algorithm 2 indeed returns the correct right-side clusters if we set $\theta = 0.75p$.

Lemma 3.6. *With high probability Algorithm 2 returns clusters $\tilde{V_1}, \ldots, \tilde{V_k}$ such that $\{\tilde{V_1}, \ldots, \tilde{V_k}\} = \{V_1, \ldots, V_k\}$.*

Proof. Condition on the events from Lemma 3.4. Let $i \in [k]$ and suppose $c \in C$ satisfies $c \in U_i$. We show $\tilde{V_c} = V_i$.

Consider the heavy hitters data structure $\mathsf{MG}(c)$. Recall that when a vertex $u \in U$ is assigned to c, we added all $j \in [n]$ to $\mathsf{MG}(c)$ with $(u, v_j) \in E$. Hence, the stream X of numbers that were processed by $\mathsf{MG}(c)$ satisfies that the frequency f_j of j is exactly $f_j = |\{u \in U_i : (u, v_j) \in E\}|$.

From the random graph model we get that $\mathbf{E}\left[f_j\right] = p|U_i|$ if $v_j \in V_i$ and $\mathbf{E}\left[f_j\right] = q|U_i|$ if $v_j \notin V_i$. Using a Chernoff bound and $|U_i| \geq K_2 \log n$, we get that when K_2 is large enough, $f_j > 0.99p|U_i|$ if $v_j \in V_i$ and $f_j < 1.01q|U_i| \leq 0.5p|U_i|$ if $v_j \notin V_i$ with probability at least $1 - n^{-2}$. Using a union bound, we get that the previous event holds for all $j \in [n]$ simultaneously with probability at least $1 - n^{-1}$. We condition on this event for the rest of the proof.

The total number of points inserted into $\mathsf{MG}(c)$ is $|X| = \sum_{u \in U_i} d(u)$ and using Lemma 3.5 and a union bound, $|X| = O(|U_i|s)$ with high probability. Thus, if we run $\mathsf{MG}(c)$ with $\varepsilon = Cp/(2s)$ for some suitable constant C, we get that $\mathsf{MG}(c)$ uses space $O(1/\varepsilon) = O(s)$ and provides an approximation $\hat{f}_j$ of each f_j within additive error $\varepsilon|X| \leq 0.1p|U_i|$.

Thus, if $v_j \in V_i$ then $\hat{f}_j \geq f_j - \varepsilon|X| \geq 0.89p|U_i|$ and if $v_j \notin V_i$ then $\hat{f}_j \leq f_j + \varepsilon|X| \leq 0.6p|U_i|$. Setting $\theta = 0.75p$ we get that the algorithm satisfies $\tilde{V_c} = V_i$. □

Lemma 3.7. *W.h.p. the space usage of Algorithm 2 is $O(ks)$ and its running time is $O(mks)$.*

Proof. Conditioning on Lemma 3.4, the algorithm only stores k centers. Storing a single center takes space $O(s)$ to store its neighbors by Lemma 3.5. Furthermore, for a single center we need to store its heavy hitters data structure. As we argued in the proof of Lemma 3.6 it suffices to use the heavy hitters data structure with $O(s)$ counters for each center. Thus, the total space usage is $O(ks)$.

Observe that for each $u \in U$ the running time is dominated by computing $d = \min_{c \in C} d(x_u, x_c)$. As there are only k centers $c \in C$ and since all $u \in U$ and $c \in C$ have only $O(s)$ neighbors, we can compute d in time $O(ks)$. Merging the heavy hitters data structures can be done in constant amortized time. Thus, the total running time for the pass over the stream is $O(mks)$ since $|U| = m$. In the postprocessing step, we only spend time $O(ks)$ because each of the heavy hitters data structures only contains $O(s)$ counters. □

3.7.2 Proof of Proposition 3.2

Any algorithm to solve the biclustering problem must be able to output the planted clusters $V_1, \ldots, V_k$. Suppose that each V_i consists of s vertices and that all V_i are mutually disjoint. Then there are $\binom{n}{ks}$ possibilities to pick the V_i. Thus, any algorithm that is able to return the V_i exactly must use at least $\log \binom{n}{ks} = \Omega(\log n^{ks}) = \Omega(ks \log n)$ bits. Since the standard word RAM model of computation is considering words of size $\Theta(\log n)$, this yields a lower bound of $\Omega(ks)$ space.

3.8 Related Work

Random graph models for bipartite graphs as presented in Section 3.2 are usually studied under the name bipartite stochastic block models (SBMs) [1]. This problem has received attention in the past [131, 207] and recently it was shown that in bipartite graphs even very small clusters can be recovered [150, 173, 219]. Furthermore, if all clusters have size $\Omega(n)$, algorithms achieving the information-theoretically optimal recovery thresholds were presented [2, 3, 220]. However, these algorithms do not work in the streaming setting and (on the hardware we used) none of them would be able to process the real-world datasets we considered in Section 3.6.

Yun et al. [215] studied SBMs in a streaming setting and provided algorithms using $O(n^{2/3})$ bits of space when the clustering does not have to be stored explicitly. However, their algorithm does not apply to bipartite graphs and it assumes that all clusters have size $\Omega(n)$ which is unrealistic in bipartite graphs as we discussed in the introduction.

Alistarh et al. [15] consider a biclustering problem in random graphs which is similar to the one studied in this chapter. They provide guarantees for recovering the left-side clusters of the graph, but they do not provide recovery guarantees for the right-side clusters. Furthermore, their data generating model is more simplistic than the one used in this chapter and their algorithm can require up to $O(kn)$ space in practice.

The BMF problem was introduced in the data mining community by Miettinen et al. [142] and has been popular in this community ever since [104, 119, 133, 140, 144, 162]. Recently, the problem was also studied in the machine learning community [122, 129, 172, 180, 181] and the theory community [28, 74]. The only streaming algorithm for BMF is by Bhattacharya et al. [31], who provided a 4-pass streaming algorithm which computes a $(1+\varepsilon)$-approximate solution for BMF. However, their algorithm is of rather theoretical nature since it requires space $O(n \cdot (\log m)^{2k} \cdot 2^{\tilde{O}(2^{2k}/\varepsilon^2)})$ and since it uses exhaustive enumeration steps which are slow in practice. Chandran et al. [52] showed that under a standard assumption in complexity theory, any approximation algorithm for BMF requires time $2^{2^{\Omega(k)}}$ or $(mn)^{\omega(1)}$; this essentially rules out practical algorithms for BMF with approximation guarantees.

We are not aware of any algorithm which (like sofa) performs a single pass over the left-side vertices of a bipartite graph and then returns a clustering of the right-side vertices.

3.9 Conclusion

We presented sofa, the first algorithm which after single pass over the left-side vertices of a bipartite graph returns the right-side clusters using sublinear memory. We showed that after a second pass over the stream, sofa solves biclustering and BMF problems. Our experiments showed that sofa is orders of magnitude faster and more memory-efficient than a static baseline algorithm while still providing high quality

results. Furthermore, we proved that under a standard random graph model, a version of sofa can find the planted clusters under a natural separation condition. In future work it will be interesting to consider streaming settings in which the edges arrive one by one. Since the main building blocks of sofa (coresets and mergeable heavy hitters data structures) extend to distributed settings, it will be interesting to make sofa distributed.

CHAPTER 4

Conditional Hardness of Approximate Counting Problems

A fundamental task in data mining is to compute pattern frequencies in datasets. For example, how often does a specific itemset occur in a given binary database, and how many triangles does a given graph have? Recent results in fine-grained complexity theory suggest that we cannot determine these quantities *exactly* other than by exhaustive enumeration or by using fast matrix multiplication. Both of these methods are often impractical. However, in practice it is often sufficient to compute these quantities *approximately*, so long as we have guarantees on the approximation error and success probability. As it turns out, *random sampling* is in many circumstances sufficient, and faster than exhaustive enumeration. Thus, a natural question is this: Is it possible to beat the running time of current random sampling algorithms?

We show that this is not the case for key problems in data mining. For example, suppose that in a binary database with m transactions we want to distinguish between whether the support of an itemset is at least m^γ or less than $m^\gamma/3$ for $\gamma \in (0, 1)$. We show that any algorithm solving this problem takes time at least $m^{1-\gamma-o(1)}$, unless the Strong Exponential Time Hypothesis (SETH) by Impagliazzo and Paturi [109] is false. This lower bound matches the running time of a simple algorithm based on random sampling and a Chernoff bound. We also obtain tight lower bounds with similar running time tradeoffs for approximating important graph quantities, such as the number of triangles, the transitivity and the clustering coefficient.

Furthermore, we give a very simple proof showing that if SETH is true, then deciding whether a SAT-formula has at least $2^{\gamma n}$ satisyfing assignments or none at all takes time $2^{(1-\gamma-o(1))n}$ for all $\gamma \in (0, 1)$.

4.1 Introduction

Many data mining algorithms employ subroutines that count occurrences of patterns in a dataset. For example, it is a common task to compute the support of an itemset in a binary database; algorithms using such a subroutine include the *a priori* algorithm [13] for computing association rules or compression-based algorithms [199] for mining a small set of interesting patterns. In bioinformatics [145, 211] and social network analysis [94, 157, 201], it is an important problem to count triangles in graphs. Many more applications can be found, such as algorithms that need to count occurences of subsequences [217] or subgraphs [208].

In particular, many data mining algorithms, that aim at extracting a set of interesting patterns from a dataset $\mathcal{D}$, build upon the following algorithmic framework:

1. Generate an initial set of candidate patterns $\mathcal{P}$
2. **until** convergence **or** timeout **do**:
   a) Generate a new set of candidate patterns $\mathcal{P}'$
   b) For each candidate pattern $P \in \mathcal{P}'$, compute the support of P in $\mathcal{D}$
   c) If *score*$(\mathcal{P}') <$ *score*$(\mathcal{P})$, then $\mathcal{P} \leftarrow \mathcal{P}'$

In the above paradigm, the function *score*$(\cdot)$ refers to the objective function that shall be minimized. For example, in compression-based algorithms such as *krimp* [199], *score*$(\mathcal{P})$ encodes how well the set $\mathcal{P}$ compresses the database, or in Boolean matrix factorization algorithms such as *asso* [142], *score*$(\mathcal{P})$ encodes the Hamming distance of the dataset and a low-rank matrix factorization. Almost all of the *score*$(\cdot)$ functions we are aware of have in common that they heavily rely on knowing the support for each pattern $P \in \mathcal{P}'$. Thus, Step 2b is crucial to compute *score*$(\mathcal{P}')$, but it is usually considered expensive because it requires a full pass over the dataset.

Next, observe that in the above paradigm all steps are algorithm- and problem-specific except Step 2b, which only depends on the type of the data. Thus, if we could provide a very fast subroutine for computing the support of a pattern in a dataset, we could speed up extremely many data mining algorithms. Hence, a fundamental question is as follows:

How fast can we compute the support of a pattern?

Unfortunately, existing results in fine-grained complexity theory show that if the support of a pattern is supposed to be computed *exactly*, i.e., when no error is allowed, then we cannot be much faster than a full pass over the dataset. For example, Williams [202] showed that computing the support of an itemset in a binary database *exactly* essentially requires reading the whole database, unless a popular conjecture in complexity theory is false (see below for details).

However, for our application of speeding up existing data mining algorithms, it often suffices to only compute the supports *approximately*. That is, we can still speed up many algorithms if we can answer the following question:

Is the support of pattern $P \in \mathcal{P}'$ at least K or less than $K/3$?
(If the support of P is between $K/3$ and K, return either answer.)

Note that the above subroutine indeed helps us to distinguish frequent patterns (with support at least K) from truly infrequent patterns (with support at most $K/3$).

With a very efficient subroutine for the above problem, practical algorithms could still be sped up as follows: Whenever the support of a pattern P needs to be computed, we first run the fast subroutine for deciding whether P's support is at least K or less than $K/3$. Then if the support of P is less than $K/3$, we can discard P and proceed with the next pattern. Else, we compute the support of P exactly (with a full pass over the dataset). In practice, this approach can still provide significant speedups and thus the above gap problem and versions thereof have been studied in many different settings [130, 136, 174, 175, 194].

For gap-problems of the above type, we know that simple random sampling algorithms provide faster running times than a full pass over the database. In this chapter, we study whether the existing algorithms can be improved or whether they are essentially optimal.

4.2 Our Results

We study the computational complexity of the above gap-problem for different types of data. We prove that for approximately computing the support of an itemset in a binary database and for approximately counting the number of triangles in a graph, one cannot do much better than very simple random sampling algorithms, unless popular conjectures in computational complexity are false. We also provide a matching upper and lower bound for approximately counting the number of solutions of a SAT formula.

While our upper bounds can solve the gap problem from Section 4.1, our lower bounds hold even for the even more idealized version of the gap problem where we need to decide whether the support of a pattern is at least K or 0. That is, the lower bounds hold even when we only have to distinugish between frequent patterns and patterns which do not appear in the database *at all*.

In the following, we formally define the problems we study and discuss our results in detail. We will also summarize the existing hardness results for computing the supports of patterns exactly.

4.2.1 Approximating the Frequencies of Itemsets

We start by studying support estimation in *binary databases* $\mathcal{D}$, that is, databases where each row has only attributes that are either true or false. The corresponding counting problem is this: Given a set T of attributes, how many rows of $\mathcal{D}$ have all attributes in T set to true? In frequent itemset mining, the attributes are usually called items, and sets of attributes are called itemsets.

Definition 4.1 (FIS). *Let $[d] := \{1, \dots, d\}$ be the set of all items. An itemset T is a subset of $[d]$. An $m \times d$ database $\mathcal{D}$ is a multiset of itemsets $T_1, \dots, T_m \subseteq [d]$, where the itemsets $T_i \in \mathcal{D}$ are called transactions. The support $\#\mathsf{supp}(T)$ of an itemset T is*

$\#\mathsf{supp}(T) = |\{i : T \subseteq T_i\}|$, *i.e.,* $\#\mathsf{supp}(T)$ *is the number of transactions in* $\mathcal{D}$ *that contain* T *as a subset. The frequencies of itemsets problem (FIS) is defined as follows: Given a database* $\mathcal{D}$ *and a set* $\mathcal{T}$ *of itemsets, return* $\#\mathsf{supp}(T)$ *for all* $T \in \mathcal{T}$*.*

Note that FIS takes a *set* $\mathcal{T}$ of itemsets as input and produces an output of $|\mathcal{T}|$ numbers. This way of defining the problem models the situation where we receive a batch of $|\mathcal{T}|$ itemsets and need to compute all of their supports at once.

Clearly, by performing a pass over the database $\mathcal{D}$ for each itemset $T \in \mathcal{T}$, FIS can be solved in time[1] $\tilde{O}(|\mathcal{T}| \cdot m)$ when $d \leq \operatorname{poly}\log m$. Is there a faster way? Probably not: Williams [202] proved that every algorithm for this problem must use time[2] at least $|\mathcal{T}| \cdot m^{1-o(1)}$ when $d \geq \omega(\log m)$, unless the Strong Exponential Time Hypothesis (SETH) is false (see Conjecture 4.3 for a formal definition). Thus, there does not seem to be an algorithm that computes the support of an itemset for worst-case binary databases much faster than performing one pass over the whole database — not even when all itemsets in $\mathcal{T}$ are presented as a batch.

Note that in FIS, we need to compute $\#\mathsf{supp}(T)$ *exactly*. Now what is the situation for the simpler problem in which we only need to distinguish whether $\#\mathsf{supp}(T) \geq K/3$ or $\#\mathsf{supp}(T) = 0$? We now formally define this problem. In the definition, we parameterize K as $K = m^\gamma$ (i.e., $\gamma = (\log K)/(\log m)$).

Definition 4.2 (gap$\#$FIS(γ))**.** *Let* $0 < \gamma < 1$*. The input to the gap*$\#$*FIS*(γ) *problem consists of an* $m \times d$ *database* $\mathcal{D}$ *and a set of itemsets* $\mathcal{T}$*. For each* $T \in \mathcal{T}$*, output* 1 *if* $\#\mathsf{supp}(T) \geq m^\gamma$*, and* 0 *if* $\#\mathsf{supp}(T) = 0$*.*

A natural approach to solve gap$\#$FIS(γ) is using random sampling. When given an itemset $T \in \mathcal{T}$, the random sampling algorithm uniformly samples a row $T_i \in \mathcal{D}$ from the database and increments a counter each time $T \subseteq T_i$ holds. By repeating the sampling $\tilde{O}(m^{1-\gamma})$ times, it is possible to distinguish between $\#\mathsf{supp}(T) \geq m^\gamma$ and $\#\mathsf{supp}(T) = 0$ with high probability. Since checking $T \subseteq T_i$ takes time $O(d)$, the random sampling algorithm runs in time $\tilde{O}(|\mathcal{T}|dm^{1-\gamma})$. See Lemma 4.16 for the details of this procedure.

To argue that the running time of the random sampling algorithm can probably not be significantly improved, we rely on the following hypothesis from fine-grained complexity, which essentially states that solving SAT takes time $2^{(1-o(1))n}$, where n is the number of variables of the formula.

Conjecture 4.3 (Strong Exponential Time Hypothesis (SETH) [109, 110] + Sparsification Lemma [49])**.** *For all* $\varepsilon > 0$*, there exists a positive integer* k *such that deciding whether a* k*-SAT formula with with* n *variables and* $O(n)$ *clauses cannot be done in time* $O(2^{(1-\varepsilon)n})$*.*

[1] We use the $\tilde{O}(\cdot)$ notation as a variant of $O(\cdot)$ notation that ignores polylogarithmic factors.

[2] When we write that an algorithm "requires time at least $T^{1-o(1)}$", this can equivalently be rephrased as follows: "For all $\varepsilon > 0$, there is no algorithm running in time $T^{1-\varepsilon}$." That is, it is impossible to obtain algorithms which are polynomially faster than time $O(T)$.

We prove that, if SETH is true, then no algorithm for gap#FIS(γ) can be polynomially faster than the previously described random sampling algorithm.

Theorem 4.4. *Let $0 < \gamma < 1$. Then:*

- *There is a randomized algorithm that solves gap#FIS(γ) for $m \times d$ databases in time $\tilde{O}(|\mathcal{T}| \cdot dm^{1-\gamma})$ with high probability if $|\mathcal{T}| \leq \mathrm{poly}(m)$.*
- *If $d = \log^2 m$, $2 \leq t \leq (\log m)^{o(1)}$ and $|\mathcal{T}| = \mathrm{poly}(m)$, then any algorithm for gap#FIS(γ) requires a running time of at least $|\mathcal{T}| \cdot m^{1-\gamma-o(1)}$ unless Conjecture 4.3 is false. This holds even in the case that, for each $T \in \mathcal{T}$ and $T_i \in \mathcal{D}$, either $T \subseteq T_i$ or $|T \cap T_i| \leq |T|/t$.*

Let us briefly discuss the theorem. (1) The upper and the lower bound from the theorem match: The lower bound instance considers databases with $d = \log^2 m$ items. In this case, the upper bound becomes $\tilde{O}(|\mathcal{T}| \cdot m^{1-\gamma})$ and, hence, matches the lower bound of $|\mathcal{T}| \cdot m^{1-\gamma-o(1)}$ up to $m^{o(1)}$ factors. (2) The lower bound holds even if we have that for each $T \in \mathcal{T}$ and $T_i \in \mathcal{D}$, either $T \subseteq T_i$ or $|T \cap T_i| \ll |T|$, i.e., either T is a subset of T_i or T and T_i only have "very few" items in common. Thus, even these two properties do not help us to obtain much faster algorithms, unless SETH is false. (3) The lower bound holds even when the number of itemsets $\mathcal{T}$ is much larger than the number of transactions in the database (the lower bound holds even $\mathcal{T}$ is extremely large, e.g., even for $|\mathcal{T}| = m^{1909}$).

We also consider the online data structure version of the gap#FIS(γ) problem. In this version of the problem, we are given the $m \times d$ database $\mathcal{D}$ in advance and we can preprocess it for $\mathrm{poly}(m)$ time. Then we obtain the itemsets $T \in \mathcal{T}$ one after another and we need to support a *query operation* that outputs whether $\#\mathsf{supp}(T) \geq m^\gamma$ or $\#\mathsf{supp}(T) = 0$. We show that even for this data structure version of the problem, each query operation must take time $m^{1-\gamma-o(1)}$, unless SETH is false (see Lemma 4.19 and Proposition 4.20).

Our lower bounds also apply to the problems of estimating the frequencies of (sub-)graphs in vertex-labeled graph databases and of (sub-)sequences in sequence databases. This follows from the reductions in Section 5.4.2 and also [154, 209].

4.2.2 Approximating the Number of Triangles and Related Measures

Next, we discuss our results for triangle counting. We start by defining three *exact* triangle counting problems.

Definition 4.5. *Let $G = (V, E)$ be a graph. A triangle is a triplet $\{u, v, w\} \subseteq V$ such that $(u, v), (v, w), (w, u) \in E$. We define the following quantities for $u, v \in V$:*

- *#Triangle(G): Return the number of triangles in G.*
- *#Triangle(G, u): Return the number of triangles in G containing u.*
- *#Triangle(G, u, v): Return the number of triangles in G containing u and v.*

All three variants of $\#\mathsf{Triangle}(\cdot)$ can be solved using exhaustive search, in time $O(n^3)$, $O(n^2)$, and $O(n)$, respectively. While exhaustive search is useful when the input is small, it is natural to wonder whether faster algorithms are possible.

This is indeed the case when we are allowed to use fast matrix multiplication: If A is the adjacency matrix of G, then $\frac{1}{6}\sum_{u,v}(A^3)_{u,v}$ is equal to the number of triangles in G. Thus, triangle counting reduces to matrix multiplication [149], which can be solved in time $O(n^\omega)$, where $\omega < 2.373$ is the optimal exponent of fast matrix multiplication [77, 203]. Unfortunately, due to the large constant hiding in the O-notation, fast matrix multiplication is usually considered impractical [126], except in highly engineered settings for extremely large data [116]. Therefore, it is preferable to develop so-called *combinatorial*[3] algorithms, that is, algorithms that "do not rely on current methods for fast matrix multiplication", since the hope is that such algorithms are more practical.

When prohibiting current fast matrix multiplication methods, the fastest known algorithm for triangle counting [213] runs in time $O(n^3/\log^4 n \cdot \mathrm{poly}\log\log(n))$, i.e., it is only polylogarithmically faster than the trivial exhaustive search algorithm. However, there are reasons to believe that this result cannot be much improved. Vassilevska Williams and Williams [205] proved that deciding whether a graph contains a triangle (that is, to decide whether $\#\mathsf{Triangle}(G) > 0$) not only reduces to Boolean matrix multiplication (BMM), but is in fact equivalent to it, in the sense that $O(n^{3-\varepsilon})$-time combinatorial algorithms for triangle detection imply $O(n^{3-\varepsilon'})$-time combinatorial algorithms for the multiplication of two $n \times n$ Boolean matrices. A popular conjecture in fine-grained complexity states that any *combinatorial* algorithm for multiplying two $n \times n$ Boolean matrices requires time $n^{3-o(1)}$. The conjecture is called the *the BMM conjecture* and it was used, for example, in [7, 59, 127, 177].

Conjecture 4.6 (Boolean Matrix Multiplication Conjecture (BMM))**.** *Any combinatorial algorithm computing the matrix-matrix product of two Boolean $n \times n$ matrices requires time $n^{3-o(1)}$.*

Thus, due to Conjecture 4.6 and the equivalence of BMM and triangle counting, there may be no practical algorithm that solves $\#\mathsf{Triangle}(G)$ significantly faster than exhaustive search (see Lemma 4.25 for details).

While the above discussion essentially settles the situation for *exact* triangle counting problems, what if we again consider a gap versions as above? Let us first formally define these problems.

Definition 4.7 (gap$\#$Triangle)**.** *Let $G = (V, E)$ be a graph with n vertices and $u, v \in V$. We define the following problems:*

- *gap$\#$Triangle(γ) for $\gamma \in (0, 3)$: Output 1 if $\#\mathsf{Triangle}(G) \geq n^\gamma$, output 0 if $\#\mathsf{Triangle}(G) = 0$.*

[3] We note that the term "combinatorial" is not well-defined [97]; informally, it means that the algorithm must not use current methods for fast matrix multiplication and should have small constants in the $O(\cdot)$-notation.

- *gap#Triangle(γ, u) for $\gamma \in (0, 2)$: Output 1 if #Triangle$(G, u) \geq n^{\gamma}$, output 0 if #Triangle$(G, u) = 0$.*
- *gap#Triangle(γ, u, v) for $\gamma \in (0, 1)$: Output 1 if #Triangle$(G, u, v) \geq n^{\gamma}$, output 0 if #Triangle$(G, u, v) = 0$.*

The following theorem summarizes our results for the gap#Triangle-problems. We again obtain that if Conjecture 4.6 is true, then the existing combinatorial random sampling algorithms cannot be significantly improved.

Theorem 4.8. *Let $G = (V, E)$ be a graph with n vertices and $u, v \in V$. We obtain the following lower bounds for combinatorial algorithms under Conjecture 4.6:*

- *Solving gap#Triangle(G, γ) takes time at least $n^{3-\gamma-o(1)}$ for $\gamma \in (0, 3)$.*
- *Solving gap#Triangle(G, γ, u) takes time at least $n^{2-\gamma-o(1)}$ for $\gamma \in (0, 2)$.*
- *Solving gap#Triangle(G, γ, u, v) takes time at least $n^{1-\gamma-o(1)}$ for $\gamma \in (0, 1)$.*

All lower bounds are matched up to $n^{o(1)}$ factors by simple sampling algorithms.

Note that for some values of γ, the theorem implies lower bounds for *sublinear time* algorithms. For example, consider the lower bound for gap#Triangle(G, γ) with $\gamma = 0.5$. Then the lower bound states that it takes time $n^{0.5-o(1)}$ to decide whether #Triangle$(G) \geq n^{2.5}$ or #Triangle$(G) = 0$. However, time $n^{0.5-o(1)}$ is not enough to read the whole graph G or even to read all neighbors of a vertex. Hence, in our reduction we ensure that we can support the following operations in $\tilde{O}(1)$ time: (1) Sample a vertex uniformly at random, (2) sample an edge uniformly at random, (3) given a vertex, sample a neighbor uniformly at random, (4) query the degree of a vertex, (5) given two vertices $u, v \in V$, return whether $(u, v) \in E$. Indeed, the random sampling algorithms from the tight upper bounds only use these operations.

Furthermore, we study dynamic versions of the gap#Triangle problems. In these dynamic versions, we allow vertex insertions and query operations asking for the number of triangles incident upon a single or multiple vertices. We derive lower bounds under the Online Matrix Vector (OMv) conjecture [97].

Definition 4.9 (Online Boolean Matrix-Vector Multiplication (OMv))**.** *Given an $n \times n$ Boolean matrix M and a sequence $(v_1, \ldots, v_n)$ of n-dimensional Boolean vectors that are provided in an online fashion, one has to return the result of the matrix-vector product Mv_i before the next vector v_{i+1} is provided.*

Conjecture 4.10 (OMv Conjecture [97])**.** *Any algorithm that solves OMv with an error probability of at most $1/3$ requires time $n^{3-o(1)}$.*

In the dynamic setting, we show that unless the OMv conjecture is false, for dynamic versions of gap#Triangle(G, γ, u) and gap#Triangle(G, γ, u, v) either the update operations must take time $n^{2-3\gamma/2-o(1)}$ and $n^{1-3\gamma-o(1)}$, respectively, or the query operations must take time $n^{2-\gamma-o(1)}$ and $n^{1-\gamma-o(1)}$, respectively. Note that since we use the OMv conjecture to derive our lower bounds, the lower bounds

hold for *any* type of algorithm (i.e., we can drop the combinatorial assumption). See Proposition 4.28 in Section 4.6.2 for the formal statement of the results.

Using our lower bounds for the (static) gap#Triangle problems, we further derive lower bounds for approximating graph metrics such as the transitivity $T(G)$ [94, 157] and the clustering coefficient $C(G)$ [201]. To formally define these quantities, let $\tau(v) = \binom{d(v)}{2}$ where $d(v)$ is the degree of vertex $v \in V$ and set $\tau(G) = \sum_{v \in V} \tau(v)$. Now for a graph with n vertices, we define

$$T(G) = \frac{3\#\mathsf{Triangle}(G)}{\tau(G)} \in [0,1],$$
$$C(G) = \frac{1}{n} \sum_{v \in V} \frac{\#\mathsf{Triangle}(G,v)}{\tau(v)} \in [0,1].$$

For these problems, we obtain the following results.

Theorem 4.11. *Let $\gamma \in (0,3)$ and let G be a graph with n vertices. We obtain the following lower bounds for combinatorial algorithms under Conjecture 4.6:*

- *Deciding whether $T(G) \geq n^{-\gamma}$ or $T(G) = 0$ takes time at least $n^{\gamma - o(1)}$.*
- *Deciding whether $C(G) \geq n^{-\gamma}$ or $C(G) = 0$ takes time at least $n^{\gamma - o(1)}$.*

4.2.3 Approximate #SAT

Finally, we also study an approximate version of #SAT. Suppose that F is a SAT formula with n variables and $O(n)$ clauses and denote the number of solutions of F by $\#\mathsf{solutions}(F)$. Now suppose we need to decide whether $\#\mathsf{solutions}(F) \geq 2^{\gamma n}$ or $\#\mathsf{solutions}(F) = 0$ for $\gamma \in (0,1)$. We show that any algorithm solving this problem takes time $2^{(1-\gamma-o(1))n}$ unless Conjecture 4.3 (SETH) is false. We also give a matching upper bound.

4.2.4 Outline

We present our results for approximate versions of FIS in Section 4.5. In Section 4.6 we present our lower bounds algorithms for approximate triangle counting. We study an approximate counting versions of SAT in Section 4.7.

4.3 Related Work

Enumerating all itemsets $T \subseteq [d]$ with $\#\mathsf{supp}(T) \geq K$ for a given minimum support K has been an important problem since Agrawal and Srikant [13] introduced the *a priori* algorithm in 1994. Gunopulos et al. [87] showed that this problem is #P-hard. Later, Yang [209] showed that even computing the number of *maximal* frequent patterns is #P-hard. This has led to several works studying the enumeration complexity [113] of frequency-based problems, e.g., [42, 117, 154]. We refer to the book by Aggarwal [11] for more references.

The FIS problem over $n \times n$ databases is a special case of the batch partial problem. For batch partial match, a sequence of papers shaved $n^{o(1)}$ factors in the running time [6, 51, 53, 176] but still all of them require time $n^{2-o(1)}$.

Using random sampling to approximate the supports of itemsets in binary databases was first proposed by Mannila et al. [136] and later refined by Toivonen [194]. Liberty et al. [130] and independently Price [168] provided tight space lower bounds for data structures solving gap versions of the FIS problem. Their results show that any algorithm deciding whether the support of an itemset is at least m^γ or less than $m^\gamma/3$ for $m \times d$ databases and $|\mathcal{T}| \leq \text{poly}(m)$ itemsets, must use space $\tilde{\Omega}(dm^{1-\gamma})$; this matches the upper bound from random sampling and a Chernoff bound. If $|\mathcal{T}| = 2^{\Omega(d)}$, they show that one needs $\tilde{\Omega}(d^2 m^{1-\gamma})$ space and this is again matched by the number of samples required by a random sampling algorithm. Riondato and Upfal [174, 175] showed that when one wants solve this problem with $|\mathcal{T}| = 2^{\Omega(d)}$ itemsets and if the data is not worst-case, then one can do with fewer samples (and, hence, less space) than suggested by the lower bounds from [130, 168] using tools from statistical learning theory such as VC dimension and Rademacher Averages.

Enumerating and counting triangles in graphs is a fundamental problems with applications in many fields, such as social network analysis [94, 157, 201] or computational biology [145, 211]. The problem has received considerable attention in static graphs [35, 183, 195], as well as in data streams [48, 111, 137, 191]. We refer to the survey by Latapy [126] for more references.

Fine-grained complexity has recently received a lot of attention. Several hardness conjectures such as SETH, 3SUM, BMM, OMv and many more were proposed and used to derive lower bounds for static (e.g., [25, 26, 45, 132, 205]) as well as for dynamic algorithms (e.g., [7, 97, 98]). Recently, also results for hardness of approximation in P were obtained [5, 54, 179]. We refer to the survey by Vassilevska Williams [204] for more references.

4.4 Preliminaries

Let F and K be numbers (it is instructive to think of $F = \#\mathsf{supp}(T)$ and a frequency threshold K). Our lower bounds are concerned with the question of deciding whether $F \geq K$ or $F = 0$. Our upper bounds will, however, consider a harder problem. More concretely, in the upper bounds our goal will be to compute an approximation $\hat{F}$ of F such that

$$|F - \hat{F}| \leq \max\{K/\beta^2, F/\beta\}$$

for constant $\beta > 1$. Observe that $\hat{F}$ satisfies that if $F \geq K$ then $|F - \hat{F}| \leq F/\beta$ and if $F \leq K/\beta$ then $|F - \hat{F}| \leq K/\beta^2$. In other words, if F is "large" then the error of $\hat{F}$ will be small *relative to* F, while if F is "small" then $\hat{F}$ will have a small *absolute* error of at most K/β^2 (and this error may be very large relative to F). Since $\hat{F}$ is

an approximation of F, our lower bounds also hold for approximate counting in the above sense.

The following lemma shows that if we can compute a number $\hat{F}$ with the above properties, then we can decide whether $F \geq K$ or $F \leq K/\alpha$ for $\alpha > 1$. This immediately implies that $\hat{F}$ can be used to decide whether $F \geq K$ or $F = 0$ and, hence, all hardness results we derive for the latter question carry over to computing numbers $\hat{F}$ with the above properties.

Lemma 4.12. *Let $\alpha > 1$ and let $\beta > \max\{\alpha, (\alpha+1)/(\alpha-1)\}$. Given a number $\hat{F}$ such that $|F - \hat{F}| \leq \max\{K/\beta^2, F/\beta\}$, we can decide whether $F \geq K$ or $F \leq K/\alpha$.*

Proof. If $\hat{F} \geq (1-\beta^{-1})K$, we output $F \geq K$, and, otherwise, we output $F \leq K/\alpha$.

We now argue that we output the correct answer. First, suppose $F \geq K$. Then $|F - \hat{F}| \leq \max\{K/\beta^2, F/\beta\} = F/\beta$ and thus $\hat{F} \geq (1-\beta^{-1})F \geq (1-\beta^{-1})K$. On the other hand, suppose $F \leq K/\alpha$. Then

$$|F - \hat{F}| \leq \max\{K/\beta^2, F/\beta\} \leq \max\{K/\beta^2, K/(\alpha\beta)\} = K/(\alpha\beta)$$

since we assumed that $\beta > \alpha$. Thus,

$$\hat{F} \leq (\alpha^{-1} + (\alpha\beta)^{-1})K \leq \left(\frac{1}{\alpha} + \frac{\alpha-1}{\alpha(\alpha+1)}\right) K = \frac{2}{\alpha+1}K.$$

Now using that $\beta > (\alpha+1)/(\alpha-1)$, we obtain that

$$(1-\beta^{-1})K > \left(1 - \frac{\alpha-1}{\alpha+1}\right) K = \frac{2}{\alpha+1}K.$$

Hence, we always output the correct answer. □

Next, we provide a simple random sampling lemma which we will use to obtain the estimates $\hat{F}$ in our algorithms.

Lemma 4.13. *Consider a bag with n balls. Suppose that each ball is either yellow or black and let F denote the number of yellow balls. Let $\beta > 1$, $s \geq 0$ and $K \geq 1$. Then we can return a number $\hat{F}$ such that $|F - \hat{F}| \leq \max\{K/\beta^2, F/\beta\}$ with probability at least $1 - O(n^{-1-s})$; to compute $\hat{F}$, we need to pick $O(s\beta^6(n/K)\log n) = \tilde{O}(n/K)$ balls from the bag uniformly and independently at random with replacement.*

The proof of the lemma is based on a simple Chernoff bound.

Lemma 4.14 (Chernoff Bound, e.g., [64, Theorem 1.1])**.** *Let $X_1, \ldots, X_n$ be independent random variables taking values in $[0,1]$ and let $X = \sum_i X_i$. Then for $\delta > 0$,*

$$\begin{aligned}
&\mathbf{Pr}\left(X > (1+\delta)\mathbf{E}\left[X\right]\right) \leq \exp(-\delta^2\mathbf{E}\left[X\right]/3), \quad 0 < \delta < 1\\
&\mathbf{Pr}\left(X < (1-\delta)\mathbf{E}\left[X\right]\right) \leq \exp(-\delta^2\mathbf{E}\left[X\right]/2), \quad 0 < \delta < 1\\
&\mathbf{Pr}\left(X > (1+\delta)\mathbf{E}\left[X\right]\right) \leq \exp(-\delta\,\mathbf{E}\left[X\right]/3), \quad \delta \geq 1.
\end{aligned}$$

Proof of Lemma 4.13. Let $\varepsilon = 1/\beta^2$. We draw $\ell = (1+s)\cdot 3\cdot(1/\varepsilon)^2\cdot 100\beta^2 n/K\cdot \log n$ balls from the bag. Let Y be the random variable denoting the number of yellow balls we drew. Then, $\mathbf{E}[Y] = \ell F/n$. We set $\hat{F} = Yn/\ell$ and thus $\mathbf{E}\left[\hat{F}\right] = F$.

Now suppose that $F \geq K/(100\beta^2)$. Then the choice of ε, Lemma 4.14 and a union imply

$$\begin{aligned}
&\mathbf{Pr}\left((1-1/\beta^2)F \leq \hat{F} \leq (1+1/\beta^2)F\right)\\
&= \mathbf{Pr}\left((1-\varepsilon)\mathbf{E}[Y] \leq Y \leq (1+\varepsilon)\mathbf{E}[Y]\right)\\
&\geq 1 - 2\exp(-(1+s)\cdot 100\beta^2 F/K\cdot\log n)\\
&\geq 1 - 2\exp(-(1+s)\log n)\\
&= 1 - 2n^{-1-s}.
\end{aligned}$$

Hence, we obtain that $|F-\hat{F}| \leq F/\beta^2 \leq \max\{K/\beta^2, F/\beta\}$ with probability at least $1 - O(n^{-1-s})$.

If $F < K/(100\beta^2)$ then using the third inequality of Lemma 4.14 with $\delta = K/(2\beta^2 F) - 1$, we get that

$$\begin{aligned}
\mathbf{Pr}\left(\hat{F} > K/(2\beta^2)\right) &= \mathbf{Pr}\left(Y > \ell/(2\beta^2 n)\right)\\
&= \mathbf{Pr}\left(Y > (1+\delta)\mathbf{E}[Y]\right)\\
&< \exp(-(1+s)20\beta^4\log n)\\
&< n^{-1-s}.
\end{aligned}$$

Thus, we get that $|F-\hat{F}| \leq K/\beta^2 \leq \max\{K/\beta^2, F/\beta\}$ with probability at least $1 - O(n^{-1-s})$. □

4.5 Upper and Lower Bounds for Approximating the Supports of Itemsets

In this section, we present our upper and lower bounds for problems related to approximating the supports of itemsets in transactional databases.

Let us briefly recall the problem definition. We assume that the set of items is $[d] = \{1,\dots,d\}$. An *itemset* is a subset of $[d]$ and usually itemsets are denoted $T \subseteq [d]$. A *database* $\mathcal{D} \in \{0,1\}^{m\times d}$ is multiset of itemsets $T_1,\dots,T_m \subseteq [d]$; the itemsets $T_i \in \mathcal{D}$ are often called *transactions*. The support $\#\mathsf{supp}(T)$ of an itemset T is $\#\mathsf{supp}(T) = |\{i : T_i \in \mathcal{D}, T \subseteq T_i\}|$, i.e., $\#\mathsf{supp}(T)$ is the number of transactions in $\mathcal{D}$ which contain T as a subset.

In the following, we will derive upper and lower bounds for approximately computing $\#\mathsf{supp}(T)$. In Section 4.5.1, we consider approximating the support of itemsets $\mathcal{T}$ in a static setting, i.e., we are given a database $\mathcal{D}$ and a set of itemsets $\mathcal{T}$ and we need to return an approximation $\widehat{\mathsf{supp}}(T)$ of $\#\mathsf{supp}(T)$ for all $T \in \mathcal{T}$. After that, in Section 4.5.2, we consider the data structure version of the problem, i.e., we are allowed to preprocess the database $\mathcal{D}$ and after that we obtain queries to return $\widehat{\mathsf{supp}}(T)$ for a set of itemsets $\mathcal{T}$.

4.5.1 Static Bounds

We start with the static setting and provide a simple sampling algorithm which efficiently approximates the supports of itemsets. After that, we prove a lower bound which matches the guarantees obtained by the algorithm.

Upper Bound. Let us first define the approximate counting problem est$\#\mathsf{FIS}_\beta(\gamma)$ for $m \times d$ transactional databases and parameters $\beta > 1$ and $\gamma \in (0,1)$. The intuition for est$\#\mathsf{FIS}_\beta(\gamma)$ is we wish to obtain estimates of $\#\mathsf{supp}(T)$ such that for "infrequent" itemsets (with support less than m^γ/β) there is a small *absolute* error (of at most m^γ/β^2) and for "frequent" itemsets (with support at least m^γ) there is a small *relative* error at most $\#\mathsf{supp}(T)/\beta$, i.e., the error is within a factor of β of the true support.

Definition 4.15 (est$\#\mathsf{FIS}_\beta(\gamma)$)**.** *Let $\gamma \in (0,1)$ and let $\beta > 1$ be a constant. In the est$\#\mathsf{FIS}_\beta(\gamma)$ problem, the input consists of an $m \times d$ database $\mathcal{D}$, parameters γ and β and a set of itemsets $\mathcal{T}$. Now for each $T \in \mathcal{T}$, we must output a number $\widehat{\mathsf{supp}}(T)$ such that*

$$|\widehat{\mathsf{supp}}(T) - \#\mathsf{supp}(T)| < \max\{m^\gamma/\beta^2, \#\mathsf{supp}(T)/\beta\}.$$

Recall that via the discussion at the beginning of Section 4.4 and Lemma 4.12, est$\#\mathsf{FIS}_\beta(\gamma)$ can be used to solve gap$\#\mathsf{FIS}(\gamma)$.

We now show that a simple sampling algorithm can solve est$\#\mathsf{FIS}_\beta(\gamma)$ efficiently.

Lemma 4.16. *If $|\mathcal{T}| = \mathrm{poly}(m)$, the est$\#\mathsf{FIS}_\beta(\gamma)$ problem over $m \times d$ databases can be solved in time $\tilde{O}(|\mathcal{T}| \cdot dm^{1-\gamma})$ with high probability.*

Proof. For each $T \in \mathcal{T}$ we apply Lemma 4.13. More concretely, we consider the database $\mathcal{D}$ as a bag and each transaction $T_i \in \mathcal{D}$ as a ball. We say that a ball T_i is yellow if $T \subseteq T_i$. Observe that the number of yellow balls is $\#\mathsf{supp}(T)$. Now applying Lemma 4.13 with m balls, $F = \#\mathsf{supp}(T)$, $s = \log|\mathcal{T}|/\log m = O(1)$ and $D = m^\gamma$, we obtain an estimate $\hat{F} = \widehat{\mathsf{supp}}(T)$ satisfying $|\widehat{\mathsf{supp}}(T) - \#\mathsf{supp}(T)| \leq \max\{m^\gamma/\beta^2, \#\mathsf{supp}(T)/\beta\}$ with probability at least $1 - O(m^{-1-s}) = 1 - O((m|\mathcal{T}|)^{-1})$. To obtain $\widehat{\mathsf{supp}}(T)$, we sampled $\tilde{O}(m^{1-\gamma}\log m)$ transactions; computing whether a sampled transaction T_i contains T (i.e., to check whether the corresponding ball is yellow) takes time $O(|T| + |T_i|) = O(d)$.

Since we perform the above procedure for each $T \in \mathcal{T}$, the total running time is $\tilde{O}(|\mathcal{T}| \cdot dm^{1-\gamma})$. A union bound implies that with probability at least $1 - O(m^{-1})$ all returned answers are correct. □

Lower Bound. Next, we provide our lower bound for gap#FIS(γ) (Definition 4.2). In this problem, we have to decide for each itemset $T \in \mathcal{T}$ whether $\#\mathsf{supp}(T) \geq m^\gamma$ or $\#\mathsf{supp}(T) = 0$.

Now we prove our lower bound for gap#FIS(γ). Note that via Lemma 4.12, the lower bound also carries over to est#FIS$_\beta(\gamma)$.

Proposition 4.17. *Any algorithm for gap#FIS(γ) for $m \times d$ databases with $d = O(\log^2 m)$ and $|\mathcal{T}| = \mathrm{poly}(m)$ itemsets, requires time $|\mathcal{T}| \cdot m^{1-\gamma-o(1)}$ unless Conjecture 4.3 is false. Our lower bound holds even when for each $T \in \mathcal{T}$ and $T_i \in \mathcal{D}$ we have that either $T \subseteq T_i$ or $|T \cap T_i| \leq |T|/t$ for $2 \leq t \leq (\log n)^{o(1)}$.*

To prove the proposition we need the following lemma, which we prove in Subsection 4.5.3 and which follows from [5, 54, 179].

Lemma 4.18. *Let $N^{\omega(1)} < M < 2^{o(N)}$ and $t \geq 2$ such that $t = (N/\log M)^{o(1)}$. Let $a \in (0, 1]$. Let $\mathcal{A}$ and $\mathcal{B}$ be two collections of sets over a universe $[N]$ with $|\mathcal{A}| = M^a$ and $|\mathcal{B}| = M$. Then no algorithm can distinguish the following two cases in time $O(M^{1+a-\varepsilon})$ for any $\varepsilon > 0$ unless Conjecture 4.3 is false:*

- *(YES case) there exist $A \in \mathcal{A}$ and $B \in \mathcal{B}$ such that $B \subseteq A$,*
- *(NO case) for all $A \in \mathcal{A}$ and $B \in \mathcal{B}$, $|A \cap B| < |B|/t$.*

Now we can prove the proposition.

Proof of Proposition 4.17. We will now take an instance $(\mathcal{A}, \mathcal{B})$ for the problem of Lemma 4.18 and create an instance $(\mathcal{D}, \mathcal{T})$ for gap#FIS(γ). We will have that $\mathcal{B}$ corresponds to the set of itemsets $\mathcal{T}$ and $\mathcal{D}$ contains multiple copies of each set $A \in \mathcal{A}$. We will then show that for $B \in \mathcal{B}$ there exists an $A \in \mathcal{A}$ such that $B \subseteq A$ *if and only if* $\#\mathsf{supp}(T_B) \geq m^\gamma$, where T_B is the itemset in $\mathcal{T}$ corresponding to B.

More concretely, let $c > 0$ be a parameter to be set later. Consider an instance $(\mathcal{A}, \mathcal{B})$ for the problem of Lemma 4.18 with the following parameters: $N = \log^2 M$, $|\mathcal{A}| = M^a$, $|\mathcal{B}| = M$, $t = (\log M)^{o(1)}$ and $a = 1/((1+c)S)$ for an arbitrary constant $S > 0$.

We create an instance for gap#FIS(γ), where $\mathcal{T} = \mathcal{B}$ and where $\mathcal{D}$ contains M^{ac} copies of each set $A \in \mathcal{A}$. Thus, $m = |\mathcal{D}| = M^a \cdot M^{ac} = M^{a(1+c)}$ and $d = N$, where we set $c = \gamma/(1-\gamma)$. Note that $|\mathcal{T}| = M = M^{a(1+c)S} = m^S$.

Now observe that for $B \in \mathcal{B}$ there exists an $A \in \mathcal{A}$ with $B \subseteq A$ *if and only if* for $T_B \in \mathcal{T}$ it holds that $\#\mathsf{supp}(T) \geq M^{ac}$, where T_B is the itemset in $\mathcal{T}$ corresponding to $\mathcal{B}$ (there is always such an itemset since we set $\mathcal{T} = \mathcal{B}$). Thus, $(\mathcal{A}, \mathcal{B})$ is a NO instance iff for all $T \in \mathcal{T}$, $\#\mathsf{supp}(T) = 0$, and $(\mathcal{A}, \mathcal{B})$ is a YES instance iff there exists a $T \in \mathcal{T}$ with $\#\mathsf{supp}(T) \geq M^{ac}$.

By definition, the gap#FIS(γ) algorithm correctly outputs for each $T \in \mathcal{T}$ whether $\#\mathsf{supp}(T) \geq m^\gamma$ or $\#\mathsf{supp}(T) = 0$. Now observe that

$$m^\gamma = M^{a(1+c)\gamma} = M^{a\gamma/(1-\gamma)} = M^{ac}. \tag{4.1}$$

Thus, the gap#FIS(γ) algorithm returns $\#\mathsf{supp}(T) \geq m^\gamma = M^{ac}$ for some $T \in \mathcal{T}$ iff $(\mathcal{A}, \mathcal{B})$ is a YES instance. Also, $(\mathcal{A}, \mathcal{B})$ is a NO instance iff $\#\mathsf{supp}(T) = 0 < m^\gamma$ for all $T \in \mathcal{T}$.

Now suppose the algorithm for gap#FIS(γ) has running time $O(|\mathcal{T}| \cdot m^{1-\gamma-\varepsilon})$. Then the total computation time is

$$\begin{aligned} O(|\mathcal{T}| \cdot m^{1-\gamma-\varepsilon}) &= O(M \cdot M^{a(1+c)(1-\gamma-\varepsilon)}) \\ &= O(M \cdot M^{a(1-\gamma-\varepsilon)/(1-\gamma)}) \\ &= O(M^{1+a-\varepsilon'}) \end{aligned}$$

for $\varepsilon' = a \cdot \varepsilon/(1-\gamma)$. Thus, we have derived a too efficient algorithm for the problem from Lemma 4.18 which contradicts Conjecture 4.3. □

4.5.2 Data Structure Bounds

Next, let us consider the data structure versions of est#FIS$_\beta(\gamma)$ and gap#FIS(γ). The difference between the static and the data structure versions is that in the latter, the algorithms obtain the database $\mathcal{D}$, β and γ first and are allowed to preprocess $\mathcal{D}$. After the preprocessing finished, the data structures must offer a query procedure which is given an itemset T and then returns an estimate of $\#\mathsf{supp}(T)$. More concretely, for the data structure version of gap#FIS$_\beta(\gamma)$, the query routine must return whether $\#\mathsf{supp}(T) \geq m^\gamma$ or $\#\mathsf{supp}(T) = 0$; for the data structure version of est#FIS$_\beta(\gamma)$, the query routine must return an estimate $\widehat{\mathsf{supp}}(T)$ such that $|\widehat{\mathsf{supp}}(T) - \#\mathsf{supp}(T)| \leq \max\{m^\gamma/\beta^2, \#\mathsf{supp}(T)/\beta\}$.

Upper Bound. Let us first look at an efficient algorithm for solving the data structure version of est#FIS$_\beta(\gamma)$.

Lemma 4.19. *The data structure version of est#FIS$_\beta(\gamma)$ over $m \times d$ databases can be solved without any preprocessing and with query time $\tilde{O}(dm^{1-\gamma})$. The queries are answered correctly with high probability.*

Proof. To obtain the claimed algorithm, we completely skip the preprocessing phase. Then for each query, we run exactly the same routine as described in the first paragraph of the proof of Lemma 4.16. This provides the desired the guarantees. □

Lower Bound. Next, we give a matching lower bound for the data structure version of gap#FIS$_\beta(\gamma)$.

Proposition 4.20. *Let $\varepsilon > 0$. Then for the data structure version of gap#FIS(γ) for $m \times d$ databases with $d = O(\log^2 m)$ and $|\mathcal{T}|$ itemsets, there exists no algorithm with preprocessing time $t_p = \mathrm{poly}(m)$ and query time $t_q = O(m^{1-\gamma-\varepsilon})$ unless Conjecture 4.3 is false. The lower bound holds even when for each $T \in \mathcal{T}$ and $T_i \in \mathcal{D}$ we have that either $T \subseteq T_i$ or $|T \cap T_i| \leq |T|/t$ for $2 \leq t \leq (\log n)^{o(1)}$.*

Proof. Let k be such that for the preprocessing time of the data structure we have that $t_p = O(m^k)$. We construct an instance with $|\mathcal{T}| = m^k$. Consider an instance $(\mathcal{A}, \mathcal{B})$ from the problem in Lemma 4.18. Now we and build the same gap#FIS(γ) instance $(\mathcal{D}, \mathcal{T})$ as in the proof of Proposition 4.17, the only difference is that we set $a = 1/(k(1+c))$ and $S = k$; all other parameters are set as before.

Now build the data structure for $\mathcal{D}$ with preprocessing time $t_p = O(m^k)$. Then for each $T \in \mathcal{T}$, we run the query procedure in time t_q. Again, we return that $(\mathcal{A}, \mathcal{B})$ is a YES instance iff one of the queries for $T \in \mathcal{T}$ returns that $\#\mathsf{supp}(T) \geq m^\gamma$; otherwise, we return that $(\mathcal{A}, \mathcal{B})$ is a NO instance. The correctness of the returned result follows from the same arguments as in the proof of Proposition 4.17 and the computation in Equation (4.1) (note that we set $c = \gamma/(1-\gamma)$ as before).

Thus, we only need to analyze the running time of the algorithm. The time spent for answering queries is

$$\begin{aligned} |\mathcal{T}| \cdot t_q &= |\mathcal{T}| \cdot O(m^{1-\gamma-\varepsilon}) \\ &= O\left(M \cdot M^{a(1+c)(1-\gamma-\varepsilon)}\right) \\ &= O\left(M^{1+a(1-\gamma-\varepsilon)/(1-\gamma)}\right) \\ &= O\left(M^{1+a-\varepsilon'}\right), \end{aligned}$$

where $\varepsilon' = a\varepsilon/(1-\gamma)$. Furthermore, the time spent for the preprocessing is

$$t_p = O(m^S) = O(M^{a(1+c)S}) = O(M).$$

Thus, the total running time of the reduction is $O(M^{1+a-\varepsilon'})$. This implies a too efficient algorithm for the problem from Lemma 4.18 and thus contradicts Conjecture 4.3. □

4.5.3 Proof of Lemma 4.18

To prove the lemma, we will use the following result from the literature.

Lemma 4.21 ([5, 54, 179])**.** *Let $N^{\omega(1)} < M < 2^{o(N)}$ and $t \geq 2$ such that $t = (N/\log M)^{o(1)}$. Let $a \in (0, 1]$. Let $\mathcal{A}$ and $\mathcal{B}$ be two collections of sets over a universe $[N]$ with $|\mathcal{A}| = M$ and $|\mathcal{B}| = M$. Then no algorithm can distinguish the following two cases in time $O(M^{2-\varepsilon})$ for any $\varepsilon > 0$ unless SETH is false:*

- *(YES case) there exist $A \in \mathcal{A}$ and $B \in \mathcal{B}$ such that $B \subseteq A$,*
- *(NO case) for all $A \in \mathcal{A}$ and $B \in \mathcal{B}$, $|A \cap B| < |B|/t$.*

Now we prove Lemma 4.18. We proceed similar as in Abboud et al. [4].

Let $\mathcal{A}'$ and $\mathcal{B}'$ be an instance of the problem of Lemma 4.21, i.e., $|\mathcal{A}'| = |\mathcal{B}'| = M$. Now arbitrarily partition $\mathcal{A}'$ into sets $\mathcal{A}'_1, \ldots, \mathcal{A}'_k$ where $k = M^{1-a}$ and where each $\mathcal{A}'_i$ has size $|\mathcal{A}'_i| = M^a$. Observe that there exist $A \in \mathcal{A}'$ and $B \in \mathcal{B}'$ such that $B \subseteq A$ iff there exist $i \in [M^{1-a}]$, $A \in \mathcal{A}'_i$ and $B \in \mathcal{B}'$ such that $B \subseteq A$. Furthermore, for all $A \in \mathcal{A}'$ and $B \in \mathcal{B}'$ we have that $|A \cap B| < |B|/t$ iff for all $i \in [M^{1-a}]$, $A \in \mathcal{A}'_i$ and $B \in \mathcal{B}'$ we have that $|A \cap B| < |B|/t$.

Now assume there exists an algorithm with running time $O(M^{1+a-\varepsilon})$, that solves the problem for sets $\mathcal{A}$ and $\mathcal{B}$ of sizes $|\mathcal{A}| = M^a$ and $|\mathcal{B}| = M$. Then for each $i \in [M^{1-a}]$, we run the algorithm on the instance $\mathcal{A} = \mathcal{A}'_i$ and $\mathcal{B} = \mathcal{B}'$. If any of the M^{1-a} instances is a YES instance, we return that the instance $\mathcal{A}'$, $\mathcal{B}'$ is a YES instance; otherwise, we return that $\mathcal{A}'$, $\mathcal{B}'$ is a NO instance. By the previous paragraph, this returns the correct answer. Furthermore, the total running is $O(M^{1-a} \cdot M^{1+a-\varepsilon}) = O(M^{2-\varepsilon})$ which contradicts SETH by Lemma 4.21.

4.6 Upper and Lower Bounds for Approximate Triangle Counting

In this section, we present our upper and lower bounds for counting the number of triangles in a graph. In this setting we are given a graph $G = (V, E)$ and are interested in triplets $\{u, v, w\} \subseteq V$ such that $(u, v), (v, w), (w, u) \in E$. We study three different types of numbers of triangles here:

- $\#\mathsf{Triangle}(G)$ counts the number of triangles in G,
- $\#\mathsf{Triangle}(G, u)$ counts the number of triangles in G that contain u, and
- $\#\mathsf{Triangle}(G, u, v)$ counts the number of triangles in G that contain u and v.

For our lower bounds, we study the gap versions of these problems, i.e., gap-$\#\mathsf{Triangle}(G, \gamma)$, gap$\#\mathsf{Triangle}(G, \gamma, u)$, gap$\#\mathsf{Triangle}(G, \gamma, u, v)$, where we need to decide whether $\#\mathsf{Triangle}(\cdot) \geq n^\gamma$ or $\#\mathsf{Triangle}(\cdot) = 0$ (see also Definition 4.7).

For our upper bounds, we consider the the estimation versions, i.e., we consider $\mathsf{est}\#\mathsf{Triangle}_\beta(G, \gamma)$, $\mathsf{est}\#\mathsf{Triangle}_\beta(G, \gamma, u)$, $\mathsf{est}\#\mathsf{Triangle}_\beta(G, \gamma, u, v)$, where one has to compute an estimate $\widehat{\#\mathsf{Triangle}}(\cdot)$ of $\#\mathsf{Triangle}(\cdot)$ such that

$$|\widehat{\#\mathsf{Triangle}}(\cdot) - \#\mathsf{Triangle}(\cdot)| < \max\{n^\gamma/\beta^2, \#\mathsf{Triangle}(\cdot)/\beta\}.$$

Note that upper bounds for the estimation versions immediately give the same upper bounds for the gap versions and lower bounds for the gap versions imply lower bounds for the estimation versions (cf. Section 4.4). Thus, we will state our lower bounds in terms of the gap problems and our upper bounds in terms of the estimation problems.

In Section 4.6.1 we consider approximating the number of triangles in a static setting, i.e., we are given a graph G and we need to return approximations of $\#\mathsf{Triangle}(G)$, $\#\mathsf{Triangle}(G, u)$, or $\#\mathsf{Triangle}(G, u, v)$. After that, in Section 4.6.2 we will consider the data structure version of these problems, i.e., we are allowed to preprocess the graph G and after that allow insertions of vertices with adjacent edges and queries to return approximations of $\#\mathsf{Triangle}(G, u)$, or $\#\mathsf{Triangle}(G, u, v)$. In Section 4.6.3, we exploit our lower bounds for the gap-$\#\mathsf{Triangle}(\cdot)$ problems and derive lower bounds for approximating the transitivity and the clustering coefficient of a graph.

4.6.1 Bounds for Static Algorithms

Let us first consider the static setting. We start by providing simple random sampling algorithms which efficiently solve the est#Triangle$_\beta(\cdot)$ problems. After that, we prove matching lower bounds for these problems.

Upper Bounds. The following lemma shows that simple random sampling algorithms solve the est#Triangle$_\beta(\cdot)$ problems efficiently.

Lemma 4.22. *Let $G = (V, E)$ be a graph with n vertices and let $u, v \in V$. The following upper bounds hold with high probability:*

1. *est#Triangle$_\beta(G, \gamma)$ for $\gamma \in [0, 3]$ can be solved in time $\tilde{O}(n^{3-\gamma})$.*
2. *est#Triangle$_\beta(G, \gamma, u)$ for $\gamma \in [0, 2]$ can be solved in time $\tilde{O}(n^{2-\gamma})$.*
3. *est#Triangle$_\beta(G, \gamma, u, v)$ for $\gamma \in [0, 1]$ can be solved in time $\tilde{O}(n^{1-\gamma})$.*

Proof. We first prove Claim 1 using Lemma 4.13. More concretely, we consider a bag that contains each triple $\{x, y, z\} \in \binom{V}{3}$ of vertices in G as a ball, i.e., there are $\binom{n}{3}$ balls. We say that a ball T_i is yellow if its vertices form a triangle in G. Observe that the number of yellow balls is #Triangle(G). Now apply Lemma 4.13 with $\binom{n}{3}$ balls, $F = \#\text{Triangle}(G)$, $s = 0$ and $D = n^\gamma$. This provides an estimate $\hat{F}$ satisfying $|\hat{F} - \#\text{Triangle}(G)| \leq \max\{n^\gamma/\beta^2, \#\text{Triangle}(G)/\beta\}$ with probability at least $1 - O(n^{-1})$. To obtain $\hat{F}$, we sampled $\tilde{O}(n^{3-\gamma})$ triples; computing whether a sampled triple is a triangle can be done in constant time. Thus, the total running time is $\tilde{O}(n^{3-\gamma})$.

The proofs of Claims (2) and (3) work similar to the one above. For Claim (2) we just sample pairs $\{y, z\} \in \binom{V}{2}$ and check whether $\{u, y, z\}$ is a triangle. Similarly, for Claim (3) we sample single vertices z and check whether $\{u, v, z\}$ is a triangle. □

Lower Bounds. Next, we provide lower bounds for exact and approximate triangle counting problems based on the BMM conjecture. To this end we turn to a conjecture that has been shown [205] to be equivalent to Conjecture 4.6 (BMM).

Conjecture 4.23. *Any combinatorial algorithm requires time $n^{3-o(1)}$ to decide whether a graph with n vertices contains a triangle.*

Conjecture 4.23 immediately implies hardness for the problems Triangle(G, u) and Triangle(G, u, v), where we need to decide whether a vertex u or a pair of vertices $\{u, v\}$, respectively, is contained in a triangle.

Lemma 4.24. *Let $G = (V, E)$ be a graph with n vertices and let $u, v \in V$. Then unless Conjecture 4.23 is false, any combinatorial algorithm:*

1. *requires time $n^{2-o(1)}$ to decide Triangle(G, u), i.e., whether G has a triangle containing u.*

2. *requires time $n^{1-o(1)}$ to decide Triangle(G, u, v), i.e., whether G has a triangle containing u and v.*

Proof. To prove the first claim, suppose that we have a combinatorial algorithm solving Triangle(G, u) in time $O(n^{2-\varepsilon})$ for $\varepsilon > 0$. By running this algorithm for all n vertices $u \in V$, we can decide whether G contains a triangle. Thus, we have obtained a combinatorial algorithm deciding Triangle(G) in time $O(n^{3-\varepsilon})$. This contradicts Conjecture 4.23. The proof of the second claim is similar. □

Furthermore, the lower bounds for the decision versions of triangle detection immediately imply lower bounds for exact counting versions #Triangle$(\cdot)$.

Lemma 4.25. *Let $G = (V, E)$ be a graph with n vertices and let $u, v \in V$. Then unless Conjecture 4.23 is false, any combinatorial algorithm:*

1. *requires time $n^{3-o(1)}$ to solve #Triangle(G),*
2. *requires time $n^{2-o(1)}$ to solve #Triangle(G, u),*
3. *requires time $n^{1-o(1)}$ to solve #Triangle(G, u, v).*

Proof. This follows immediately from the fact that if we can compute #Triangle$(\cdot)$ exactly, then we can distinguish between #Triangle$(\cdot) \geq 1$ and #Triangle$(\cdot) = 0$. Thus, we can solve the decision versions of the problems. Now the running time lower bounds follow from Lemma 4.24. □

Next, we use Conjecture 4.23 and Lemma 4.24 to prove lower bounds for algorithms solving gap#Triangle$(\cdot)$. Note that via Lemma 4.12 these lower bounds also hold for algorithms solving est#Triangle$_\beta(\cdot)$.

Proposition 4.26. *Let $G = (V, E)$ be a graph with n vertices and let $u, v \in V$. Then unless Conjecture 4.23 is false, any combinatorial algorithm:*

1. *requires time $n^{3-\gamma-o(1)}$ to solve gap#Triangle(G, γ) for $\gamma \in (0, 3)$,*
2. *requires time $n^{2-\gamma-o(1)}$ to solve gap#Triangle(G, γ, u) for $\gamma \in (0, 2)$, and*
3. *requires time $n^{1-\gamma-o(1)}$ to solve gap#Triangle(G, γ, u, v) with $\gamma \in (0, 1)$.*

Proof. Let $G = (V, E)$ be an instance of Triangle$(\cdot)$ with $|V| = n$ vertices. Let $\delta > 0$ be a parameter which we will set later. We define a graph $G' = (V', E')$ which for each vertex $v \in V$ contains n^δ copies $v_1, \ldots, v_{n^\delta}$ of v. The copies u_i and v_j of two vertices are adjacent in G' iff the original vertices u and v are connected in G, i.e., $E' = \{\{v_i, u_j\} : \{u, v\} \in E, 1 \leq i, j \leq n^\delta\}$. Note that $|V'| = n^{1+\delta}$ and $|E'| = n^{2\delta} \cdot |E|$. Further observe that by construction each triangle $\{u, v, w\}$ in G corresponds to $n^{3\delta}$ triangles $\{\{u_i, v_j, w_k\} : 1 \leq i, j, k \leq n^\delta\}$ in G' and, moreover, each triangle in G' corresponds to a triangle in G. Thus, G contains a triangle iff #Triangle$(G') \geq n^{3\delta}$, and G does not contain a triangle iff #Triangle$(G') = 0$.

To prove Claim (1), set $\delta = \gamma/(3-\gamma)$ and note that $\gamma = 3\delta/(1+\delta)$. From the above arguments we obtain that if G contains a triangle then #Triangle$(G') \geq n^{3\delta} = |V'|^\gamma$ and if G is triangle-free then #Triangle$(G') = 0$. Thus, if the algorithm for gap#Triangle(G', γ) returns that #Triangle$(G') \geq |V'|^\gamma$ then G must

contain a triangle and, otherwise, G is triangle-free. Now, towards a contradiction, assume that there is a combinatorial algorithm for gap#Triangle(G', γ) with running time $O(|V'|^{(1-\varepsilon)(3-\gamma)})$. Since $|V'| = n^{1+\delta} = n^{3/(3-\gamma)}$, this implies that the algorithm for gap#Triangle(G', γ) solves Triangle(G) in time $O(n^{3-\varepsilon})$. This contradicts Conjecture 4.23.

To prove Claim (2), set $\delta = \gamma/(2-\gamma)$ and note that $\gamma = 2\delta/(1+\delta)$. Similar to above we have that each triangle in G containing $u \in V$ corresponds to $n^{2\delta}$ triangles containing $u_1 \in V'$ in G'; more precisely, each triangle $\{u, v, w\}$ in G corresponds to the $n^{2\delta}$ triangles $\{\{u_1, v_j, w_k\} : 1 \leq j, k \leq n^\delta\}$ in G'. Thus, if G contains a triangle then #Triangle$(G', u_1) \geq n^{2\delta} = |V'|^\gamma$, and if G is triangle-free then #Triangle$(G', u_1) = 0$. Thus, the algorithm for gap#Triangle(G', γ, u_1) returns #Triangle$(G', u_1) \geq |V'|^\gamma$ iff G contains a triangle with vertex u. Now, towards a contradiction, assume that there is a combinatorial algorithm for gap#Triangle(G', γ, u_1) with running time $O(|V'|^{(1-\varepsilon)(2-\gamma)})$. Since $|V'| = n^{1+\delta} = n^{2/(2-\gamma)}$, we can solve Triangle$(G, u)$ in time $O(n^{2-\varepsilon})$. This contradicts Conjecture 4.23 via Lemma 4.24.

To prove Claim (3), set $\delta = \gamma/(1-\gamma)$ and observe that $\gamma = \delta/(1+\delta)$. Similar to above, each triangle $\{u, v, w\}$ in G corresponds to the n^δ triangles $\{\{u_1, v_1, w_i\} : 1 \leq i \leq n^\delta\}$ in G'. Thus, #Triangle$(G', u_1, v_1) \geq n^\delta = |V'|^\gamma$ if G contains a triangle and #Triangle$(G', u_1, v_1) = 0$ if G is triangle-free. Hence, the gap#Triangle(G', δ, u_1, v_1) algorithm can be used to decide whether G contains a triangle with vertices u and v. Now, towards a contradiction, assume that there is a combinatorial algorithm for gap#Triangle(G', δ, u_1, v_1) with running time $O(|V'|^{(1-\varepsilon)(1-\gamma)})$. Since $|V'| = n^{1+\delta} = n^{1/(1-\gamma)}$, we can solve Triangle$(G, u, v)$ in time $O(n^{1-\varepsilon})$. This contradicts Conjecture 4.23 via Lemma 4.24.

Note the following two subtleties when constructing G': (A) Consider gap#Triangle(G', γ) with $\gamma < 1$. In this case, $\delta < 1/2$ and G' contains at most $|E'| = n^{2\delta} \cdot |E| = n^{3-\Omega(1)}$ edges. Hence, we can construct G' explicitly in truly subcubic time and a more efficient algorithm for gap#Triangle(G', γ) would still break Conjecture 4.23. (B) For all other cases (i.e., gap#Triangle(G', γ) with $\gamma > 1$, gap#Triangle(G', γ, u) with $\gamma \in (0, 2)$ and gap#Triangle(G', γ, u, v) with $\gamma \in (0, 1)$), the running time bounds of the algorithms do not allow to process the full input graph $G' = (V', E')$ (because G' might contain up to $\Omega(|V'|^2)$ edges while the running time bounds only allow time $o(|V'|^2)$). Hence, we argue that in our reduction we can provide typical oracle calls used by sublinear time algorithms: (1) Sample a vertex or edge, resp., from G' uniformly at random: Sample a vertex or edge, resp., of G uniformly at random then uniformly at random pick one of the copies of this vertex or edge, resp., in G'. (2) Given a vertex $v_i \in V'$, sample one of its neighbors in G': We sample a neighbor $w \in V$ of the corresponding vertex v in G and then uniformly at random pick on of the copies of w_i in G'. (3) We can compute the degree of a vertex v' in G' by computing the degree of the corresponding vertex v in G and multiplying it by n^δ. (4) Given $u_i, v_j \in V'$, check whether $(u_i, v_j) \in E'$: We simply check whether $(u, v) \in E$. All these operations can be

performed in time $\tilde{O}(1)$. □

4.6.2 Bounds for Data Structures

Next, let us consider the data structure versions of est#Triangle$_\beta(\cdot)$ and gap-#Triangle$(\cdot)$. The difference of the static and the data structure versions is that in the latter, the algorithms obtain the graph G, β and γ first and are allowed to preprocess G. After the preprocessing finished, the data structure must offer the following operations: (1) An update procedure that allows to add a vertex together with its adjacent edges into G. (2) A query procedure which, given a vertex u or a pair of vertices (u, v), returns the number of triangles containing a given vertex u or the pair (u, v), respectively.

Upper Bounds. We obtain an upper bound for the data structure version of est-#Triangle$_\beta(\cdot)$.

Lemma 4.27. *Let G be a dynamic graph undergoing vertex insertions and let n be the number of vertices in G after the final insertion. Then the data structure version of est#Triangle$_\beta(\cdot)$ be solved without any preprocessing, worst-case update time $O(n)$ and the following worst-case query times:*

- *$\tilde{O}(n^{2-\gamma})$ for gap#Triangle(G, γ, u) queries with $\gamma \in (0, 2)$, and*
- *$\tilde{O}(n^{1-\gamma})$ for gap#Triangle(G, γ, u, v) queries with $\gamma \in (0, 1)$.*

The queries are answered correctly with high probability.

Proof. To obtain the algorithm, we skip the preprocessing altogether. During an update, we insert the new vertex together with all of its neighbors into the graph data structure in time $O(n)$. For each query, we run the respective routine from Lemma 4.22. □

Lower Bounds. Next, we derive lower bounds for the gap#Triangle$(\cdot)$ problems based on Conjecture 4.10 (OMv). Note that the query time lower bounds we derive are the same as in the static setting and that the lower bounds of this section hold for any type of algorithm, i.e., we can drop the combinatorial assumption that we previously required from Conjecture 4.6 (BMM).

Proposition 4.28. *Let $\varepsilon > 0$. Let G be a dynamic graph undergoing vertex insertions and let n denote the number of vertices in G after the final insertion. Then for the data structure version of gap#Triangle, unless Conjecture 4.10 is false:*

1. *There exists no algorithm for gap#Triangle(G, γ, u) with preprocessing time $t_p = \text{poly}(n)$, amortized update time $t_u = O(n^{2-3\gamma/2-\varepsilon})$, and amortized query time $t_q = O(n^{2-\gamma-\varepsilon})$ for $\gamma \in (0, 4/3)$.*
2. *There exists no algorithm for gap#Triangle(G, γ, u, v) with preprocessing time $t_p = \text{poly}(n)$, amortized update time $t_u = O(n^{2-3\gamma-\varepsilon})$, and amortized query time $t_q = O(n^{1-\gamma-\varepsilon})$ for $\gamma \in (0, 2/3)$.*

In the proof of the proposition we will use the following lemma from [97, Corollary 3.4]. The lemma provides lower bounds for the data structure versions of Triangle(u) and Triangle(u, v). In these problems, the input is a graph G which can be preprocessed. After that, the data structure must offer the following operations. (1) Update operations that allow to insert vertices together with all of their incident edges into the graph. (2) Query operations which return whether a vertex u or a pair of vertices (u, v), resp., is contained in a triangle.

Lemma 4.29 ([97, Corollary 3.4])**.** *Let $\varepsilon > 0$ and let G be a dynamic graph undergoing vertex insertions and let n denote the number of vertices in G after the final insertion. Then unless Conjecture 4.10 is false, the following holds:*

- *There exists no algorithm for the data structure version of Triangle(G, u) with preprocessing time $t_p = \mathrm{poly}(n)$, amortized update time $t_u = O(n^{2-\varepsilon})$, and amortized query time $t_q = O(n^{2-\varepsilon})$.*
- *There exists no algorithm for the data structure version of Triangle(G, u, v) with preprocessing time $t_p = \mathrm{poly}(n)$, amortized update time $t_u = O(n^{2-\varepsilon})$, and amortized query time $t_q = O(n^{1-\varepsilon})$.*

Proof of Proposition 4.28. Consider an instance of the data structure version of Triangle$(\cdot)$ from Lemma 4.29 with a dynamic graph $G = (V, E)$ which after the final vertex insertion contains n vertices.

Let $\delta > 0$ be a parameter which we will set later. First, consider the preprocessing phase and let $G_0 = (V_0, E_0)$ be the graph which is the initial input to the Triangle$(\cdot)$ instance. We now construct a graph $G'_0 = (V'_0, E'_0)$ with n^δ many copies $v_1, \dots, v_{n^\delta}$ of each vertex $v \in V_0$. Then two vertices are adjacent in G'_0 iff the original vertices are connected in G_0, i.e., $E'_0 = \{\{v_i, u_j\} : \{u, v\} \in E_0, 1 \leq i, j \leq n^\delta\}$.

Next, suppose in the k'th update a vertex u is inserted into the graph G of the Triangle$(\cdot)$ instance. Then we insert n^δ copies $u_1, \dots, u_{n^\delta}$ of u into G'. Each u_i has as neighbors the vertices $\{v_j : (u, v) \in E(G)\}$, i.e., u_i is connected to all vertices v_j which are copies of vertices v such that $(u, v) \in E(G)$. Observe that a single vertex insertion in G with $O(n)$ edges corresponds to n^δ vertex insertions insertion G' with $O(n^{1+\delta})$ edge insertion.

Further let $G_k = (V_k, E_k)$ denote the graph G after the k'th vertex insertion. Note that after the k'th insertion into G, the gap#Triangle$(\cdot)$ instance G' is a graph $G'_k = (V'_k, E'_k)$ with n^δ many copies $v_1, \dots, v_{n^\delta}$ of each vertex $v \in V_k$. Also, two vertices are adjacent in G'_k iff the original vertices are connected in G_k, i.e., $E'_k = \{\{u_i, v_j\} : \{u, v\} \in E_k, 1 \leq i, j \leq n^\delta\}$. Observe that by construction for all $k \geq 0$, each triangle $\{u, v, w\}$ in G_k corresponds to $n^{3\delta}$ triangles $\{u_i, v_j, w_r\}$, $1 \leq i, j, r \leq n^\delta$, in G'_k and moreover each triangle in G'_k corresponds to a triangle in G_k.

Now to prove Claim (1), set $\delta = \gamma/(2 - \gamma)$. By the same argument as in the proof of Proposition 4.26, the query for gap#Triangle(G', γ, u_1) returns #Triangle$(G', u_1) \geq |V'|^\gamma$ iff G contains a triangle containing u. Thus, the

data structure for gap#Triangle(G, γ, u_1) can be used as a data structure for Triangle(G, u). Now, towards a contradiction, assume that there is an algorithm for gap#Triangle(G, γ, u_1) with preprocessing time $t_p = \text{poly}(n)$, update time $t_u = O(n^{2-3\gamma/2-\varepsilon})$, and query time $t_q = O(n^{2-\gamma-\varepsilon})$. Note that the update time spent by this algorithm is

$$\begin{aligned} n^{\delta} \cdot O(n^{(1+\delta)(2-3\gamma/2-\varepsilon')}) &= O(n^{\delta+(1+\delta)(2-3\gamma/2-\varepsilon)}) \\ &= O(n^{\gamma/(2-\gamma)+2\cdot(2-3\gamma/2-\varepsilon)/(2-\gamma)}) \\ &= O(n^{2-\varepsilon'}), \end{aligned}$$

where $\varepsilon' = \varepsilon/(2-\gamma)$. Finally, to answer a query Triangle(G, u), we evaluate gap-#Triangle(G', γ, u_1) which requires time

$$O(|V'|^{2-\gamma-\varepsilon}) = O(n^{(1+\delta)(2-\gamma-\varepsilon)}) = O(n^{2/(2-\gamma)\cdot(2-\gamma-\varepsilon)}) = O(n^{2-\varepsilon'}).$$

Hence by Lemma 4.29 such an algorithm contradicts Conjecture 4.10.

To prove Claim (2), set $\delta = \gamma/(1-\gamma)$. By the same argument as in the proof of Proposition 4.26, the query for gap#Triangle(G', γ, u_1, v_1) returns that #Triangle$(G', u_1, v_1) \geq |V'|^{\gamma}$ iff G contains a triangle containing u and v. Thus, the data structure for gap#Triangle(G', γ, u_1, v_1) can be used as a data structure for Triangle(G, u, v). Now, towards a contradiction, assume that there is an algorithm for gap#Triangle(G, γ, u, v) with preprocessing time $t_p = \text{poly}(n)$, update time $t_u = O(n^{2-3\gamma-\varepsilon})$, and query time $t_q = O(n^{1-\gamma-\varepsilon})$. To implement a single update for Triangle(G, u, v) we need to perform n^{δ} updates of gap-#Triangle(G', γ, u_i, v_j) and this can be done in time

$$\begin{aligned} n^{\delta} \cdot O(n^{(1+\delta)(2-3\gamma-\varepsilon)}) &= O(n^{\delta+(1+\delta)(2-3\gamma-\varepsilon)}) \\ &= O(n^{\gamma/(1-\gamma)+(2-3\gamma-\varepsilon)/(1-\gamma)}) \\ &= O(n^{2-\varepsilon'}), \end{aligned}$$

where $\varepsilon' = \varepsilon/(1-\gamma)$. Finally, to answer a query Triangle(G, u, v), we evaluate gap#Triangle(G', γ, u_1, v_1) which takes time

$$O(|V'|^{1-\gamma-\varepsilon}) = O(n^{(1+\delta)\cdot(1-\gamma-\varepsilon)}) = O(n^{(1-\gamma-\varepsilon)/(1-\gamma)}) = O(n^{1-\varepsilon''}).$$

Hence by Lemma 4.29 such an algorithm contradicts Conjecture 4.10. □

4.6.3 Bounds for Clustering Coefficient and Transitivity

Finally, we illustrate how our lower bounds on gap#Triangle$(\cdot)$ can be used to study the limits for approximations of graph metrics, such as the transitivity and the clustering coefficient of a graph.

Proposition 4.30. *Let $\gamma \in (0, 3)$ and let G be a graph with n vertices. Then unless Conjecture 4.6 is false, any combinatorial algorithm:*

- *Requires time* $n^{\gamma-o(1)}$ *to decide whether* $T(G) \geq n^{-\gamma}$ *or* $T(G) = 0$.
- *Requires time* $n^{\gamma-o(1)}$ *to decide whether* $C(G) \geq n^{-\gamma}$ *or* $C(G) = 0$.

Proof. Let us first consider approximating the transitivity $T(G)$. Consider an instance of gap#Triangle$(G, 3-\gamma)$ given by a graph $G = (V, E)$ with $|V| = n$. We now have the following correspondence between gap#Triangle$(G, 3-\gamma)$ and $T(G)$. First, G has no triangle iff $T(G) = 0$. Second, G has at least $n^{3-\gamma}$ triangles iff $T(G') \geq n^{-\gamma}$ (since $\tau(G) \leq n^3$). Towards a contradiction assume we could decide $T(G) \geq n^{-\gamma}$ or $T(G) = 0$ in time $O(n^{\gamma-\varepsilon})$ for $\varepsilon > 0$. This implies an algorithm for gap#Triangle$(G, 3-\gamma)$ with running time $O(n^{\gamma-\varepsilon})$. This contradicts Conjecture 4.6 via Proposition 4.26.

Now we consider approximating the clustering coefficient $C(G)$. Again consider an instance of gap#Triangle$(G, 3-\gamma)$ given by a graph $G = (V, E)$ with $|V| = n$. Now observe that

$$\begin{aligned} C(G) &= \frac{1}{n} \sum_{v \in V} \frac{\#\mathsf{Triangle}(G, v)}{\tau(v)} \\ &\geq \frac{1}{n} \sum_{v \in V} \frac{\#\mathsf{Triangle}(G, v)}{n^2} \\ &= \frac{3\#\mathsf{Triangle}(G)}{n^3}. \end{aligned}$$

Thus, (similar to the proof for the transitivity) we get that (a) G has no triangle iff $C(G) = 0$ and (b) G has at least $n^{3-\gamma}$ triangles iff $C(G') \geq n^{-\gamma}$. Towards a contradiction assume we could decide $C(G) \geq n^{-\gamma}$ or $C(G) = 0$ in time $O(n^{\gamma-\varepsilon})$ for $\varepsilon > 0$. This implies an algorithm for gap#Triangle$(G, 3-\gamma)$ with running time $O(n^{\gamma-\varepsilon})$. This contradicts Conjecture 4.6 via Proposition 4.26. □

4.7 Upper and Lower Bounds for Approximate #SAT

Let F be a SAT formula. Denote the number of satisyfing variable assignments to F by $\#\mathsf{solutions}(F)$. In this section, we show that if SETH is true, then distinguishing between $\#\mathsf{solutions}(F) > 2^{\gamma n}$ and $\#\mathsf{solutions}(F) = 0$ cannot be done faster than in time $2^{(1-\gamma-o(1))n}$. We also provide a matching upper bound.

More concretely, we will be studying the following two problems.

Definition 4.31 (gap#SAT)**.** *Given a SAT formula* F *over* n *variables and* $\gamma \in [0, 1)$*, the gap*$\#SAT_\alpha(\gamma)$ *problem is to output:*

- 1 *if* $\#\mathsf{solutions}(F) \geq 2^{\gamma n}$*,*
- 0 *if* $\#\mathsf{solutions}(F) = 0$*.*

Definition 4.32 (est#SAT)**.** *Given a SAT formula* F *over* n *variables,* $\gamma \in [0, 1)$ *and* $\beta > 1$*, the est*$\#SAT_\beta(\gamma)$ *problem is to output* $\widehat{\#\mathsf{solutions}}(F)$ *such that:*

$$|\widehat{\#\mathsf{solutions}}(F) - \#\mathsf{solutions}(F)| \leq \max\{2^{\gamma n}/\beta^2, \#\mathsf{solutions}(F)/\beta\}.$$

Next, we first present an upper bound for est#SAT and then a lower bound for gap#SAT. Note that the upper bound for the estimation version implies the same upper bound for the gap version; furthermore, the lower bound for the gap version implies a lower bound for the estimation version (see Section 4.4).

Upper Bound. We first give an upper bound for est#SAT using random sampling.

Lemma 4.33. *The est#SAT$_\beta(\gamma)$ problem over SAT formulas with n variables and $O(n)$ clauses can be solved in time $O(2^{(1-\gamma)n}\operatorname{poly}(n))$.*

Proof. Let F be a CNF formula over n variables. To approximate $\#\mathsf{solutions}(F)$, we apply Lemma 4.13. More concretely, we consider a bag containing 2^n balls, i.e., one ball for each truth assignment T_i over the n variables of the formula. We say that a ball T_i is yellow if $T_i(F) = 1$, i.e., if the truth assignment T_i satisfies F. Observe that the number of yellow balls is $\#\mathsf{solutions}(F)$. Now apply Lemma 4.13 with $s = 0$ and $D = 2^{\gamma n}$, we obtain an estimate $\widehat{\#\mathsf{solutions}}(F)$ satisfying $|\widehat{\#\mathsf{solutions}}(F) - \#\mathsf{solutions}(F)| \leq \max\{2^{\gamma n}/\beta^2, \#\mathsf{solutions}(F)/\beta\}$ with high probability. To obtain $\widehat{\#\mathsf{solutions}}(F)$, we sampled $\tilde{O}(2^{(1-\gamma)n}n)$ truth assignments T_i and for each T_i we can test whether $T_i(F) = 1$ in linear time. Thus, the total running time is $O(2^{(1-\gamma)n}\operatorname{poly}(n))$. □

Lower Bound. We next give a lower bound for gap#SAT.

Lemma 4.34. *Let $\gamma \in [0, 1)$ and $\varepsilon > 0$. If there exists an algorithm which for all k and n solves gap#SAT(γ) for k-SAT formulas over n variables in time $O(2^{(1-\varepsilon)(1-\gamma)n})$, then Conjecture 4.3 is false.*

Proof. Let F be a k-SAT formula over $(1-\gamma)n$ variables. We construct a new k-SAT formula F' by adding γn free variables to F (a variable is *free* if it does not appear in any clause). Note that F' has $n = (1-\gamma)n + \gamma n$ variables. Observe that if $\#\mathsf{solutions}(F) \geq 1$, then $\#\mathsf{solutions}(F') \geq 2^{\gamma n}$ since for each solution of F we obtain $2^{\gamma n}$ solutions for F' by assigning different values to the γn free variables. If $\#\mathsf{solutions}(F) = 0$, then $\#\mathsf{solutions}(F') = 0$ since the free variables do not satisfy any additional clause.

Now apply the gap#SAT(γ) algorithm on F' and return that F is satisfiable iff the gap#SAT(γ) algorithm returns 1. Indeed, the answer is correct: If F is satisfiable then $\#\mathsf{solutions}(F) \geq 1$ and, hence, $\#\mathsf{solutions}(F') \geq 2^{\gamma n}$. If F is not satisfiable then we have that $\#\mathsf{solutions}(F') = 0$ and the algorithm for gap#SAT(γ) outputs 0.

The running time for this algorithm is $O(2^{(1-\varepsilon)(1-\gamma)n})$. Thus we have derived an algorithm solving k-SAT over $n' = (1-\gamma)n$ variables in time $O(2^{(1-\varepsilon)n'})$ which contradicts Conjecture 4.3. □

4.8 Conclusion

We showed that for several computational problems, simple random sampling algorithms cannot be significantly improved unless one of several conjectures in computational complexity is false. In particular, we provided matching upper and lower bounds for approximating the support of itemsets in transactional databases and for approximating the number of triangles in a graph.

Our results essentially settle the worst-case complexity of the above approximate counting problems. Hence, it will be interesting to further the study of these problems using beyond worst-case analysis similar to the works of [125, 174, 175].

CHAPTER 5

Reductions for Frequency-Based Data Mining Problems

Studying the computational complexity of problems is a fundamental question in computer science. Yet, surprisingly little is known about the computational complexity of many central problems in data mining. In this chapter, we study frequency-based problems and propose a new type of reduction that allows us to compare the complexities of maximal frequent pattern mining problems in different types of data (e.g., graphs and sequences). Our results extend those of Kimelfeld and Kolaitis [117, ACM TODS'14] to a broader range of data mining problems. Furthermore, we show that, by allowing constraints in the pattern space, the complexities of many maximal frequent pattern mining problems collapse. These problems include maximal frequent subgraphs in labelled graphs, maximal frequent itemsets, and maximal frequent subsequences with no repetitions. In addition to theoretical interest, our results might yield more efficient algorithms for the studied problems.

5.1 Introduction

Computational complexity is a fundamental concept in computer science, with the P vs. NP question being one of the most famous open problems in the field. Yet, outside some NP- and #P-hardness proofs, the computational complexity of central data mining problems is surprisingly little studied. This is perhaps even more true for *frequency-based problems*, that is, for problems where the goal is to enumerate all sufficiently frequent patterns (that admit other possible constraints). Problems such as frequent itemset mining, frequent subgraph mining, and frequent subsequence mining all belong to this family of problems. Often the only computational complexity argument for these problems is the observation that the output can be exponentially large with respect to the input, and hence any algorithm might need exponential time to enumerate the results.

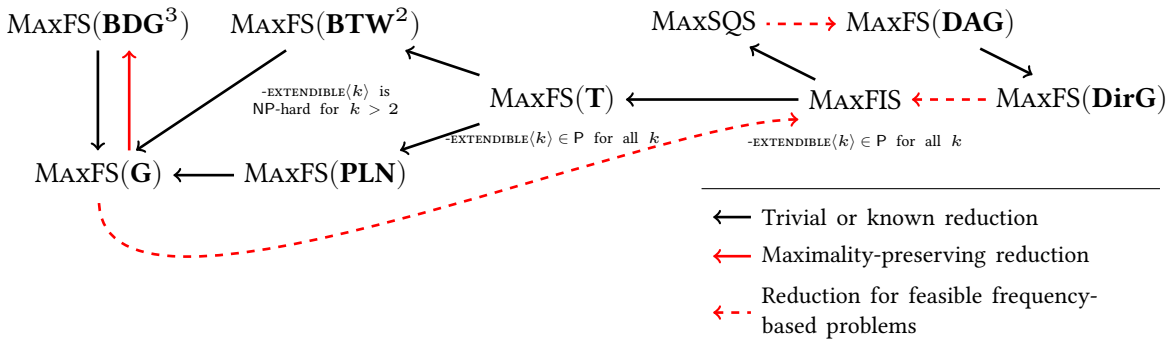


Figure 5.1: The hierarchy of maximal frequency-based problems with the results from this chapter. Arrows point from the "easier" to the "harder" problem. See Section 5.2.2 for the abbreviated problem names used in the picture. Maximality-preserving reductions are defined in Section 5.4 and feasible frequency-based problems are defined in Section 5.5.

We argue that this view is too limited for two reasons. First, there are more fine-grained models of complexity than just the running time. In particular, for enumeration problems we can use the *enumeration complexity* framework of Johnson et al. [113]: in short, instead of studying the total running time with respect to the input size, we can consider it as a function of the total size of input *and output*, or study the time it takes to create a *new* pattern when a set of patterns is already known (see Section 5.2.3 for more details). This framework allows us to argue about the time complexity of enumeration problems with potentially exponential output sizes. Another approach is the counting complexity framework of Valiant [197] (see Section 5.2.4).

The second reason why we argue that the "output is exponential" is a too limited view for the computational complexity is that a significant question in computational complexity is the relationship between the problems, that is, questions like "can we solve problem X efficiently if we can solve problem Y efficiently?" The main tool for answering these kinds of questions are *reductions* between problems. In this work, we introduce a new type of reduction between frequency-based problems called *maximality-preserving reduction* (see Section 5.4). Our reduction maps the maximal patterns of one problem to the maximal patterns of the other problem, thus allowing us to study questions like "can we find the maximal frequent subgraphs on labelled graphs using maximal frequent itemset mining algorithms?"

Surprisingly, the answer to this question turns out to be positive, although it requires that we consider specially constrained maximal frequent pattern mining problems; we call the general class of such problems *feasible frequency-based problems* (see Section 5.5).

Our Contributions. We study a number of maximal pattern mining problems, including maximal frequent subgraph mining in labelled graphs (and in several restricted graph classes), maximal frequent itemset mining, and maximal frequent subsequence mining with no repetitions (see Section 5.2.2 for definitions of these problems). We summarize our results in Figure 5.1: the arrows show which problem can be reduced to which other problem either using non-constraining reductions (black and red lines), or with possible constraints on the feasible solutions (dashed lines). As can be seen in Figure 5.1, all problems can be reduced to each other (potentially with constraints). Given that the constrained reductions are transitive (Lemma 5.12), we can state our main result:

Theorem 5.1 (Informal)**.** *Maximal subgraph mining in labelled graphs (and in several restricted graph classes), maximal frequent itemset mining, and maximal subsequence mining with no repetitions are equally hard problems when we are allowed to constrain the pattern space.*

In some sense, our results unify existing hardness results for frequency-based problems by putting them into a general framework using maximality-preserving reductions. These reductions preserve all interesting theoretical aspects like NP- or #P-hardness, but are still restricted enough to maintain the special properties of the transactions.

In fact, from a practical point of view, our reductions show that if we have an algorithm that can effectively find, say, the maximal frequent itemsets that admit the constraints from the reductions, we can use that algorithm to solve maximal frequent subgraph mining and maximal frequent subsequence mining problems efficiently. Luckily, as we will see in Section 5.6, the constrained maximal patterns are indeed easy to mine in practice. Alternatively, the reductions may be used to guide how ideas from algorithms for one set of problems can be transferred to algorithms for the other set of problems (e.g., from frequent subsequence mining to frequent subgraph mining or vice versa).

Outline of the Chapter. We will cover the basic definitions and frameworks used in this chapter in Section 5.2, where we will also formally define the problems we are working with. Section 5.3 presents related work and existing hardness results for the problems we consider. We introduce the (unconstrained) maximality-preserving reductions in Section 5.4. In particular, the reduction corresponding to the solid red line in Figure 5.1 is presented in Section 5.4.2. The feasible frequency-based problems, and the corresponding constrained reductions (dashed red lines in Figure 5.1) and related results are presented in Section 5.5. In Section 5.6 we show that our reductions can be used in practice and yield efficient algorithms.

5.2 Preliminaries

In this section we cover the basic definitions of frequency-based problems, enumeration problems, and counting complexity. In addition, we present the definitions of the problems we consider in the chapter.

5.2.1 Frequency-based Problems

A *frequency-based* problem $\mathcal{P}$ consists of[1]:

- A set of labels $\mathcal{L}$; for example, $\mathcal{L} = \{1, \dots, n\}$.
- A set *transactions*$(\mathcal{P})$ consisting of possible transactions over the labels $\mathcal{L}$.
- A set *patterns*$(\mathcal{P}) \subseteq$ *transactions*$(\mathcal{P})$ of possible patterns over the labels $\mathcal{L}$.
- A partial order $\sqsubseteq$ over *transactions*$(\mathcal{P})$.

Given a frequency-based problem $\mathcal{P}$, a *database* $D_{\mathcal{P}}$ is a finite multiset of elements from *transactions*$(\mathcal{P})$. For a database $D_{\mathcal{P}}$ and a *support threshold* τ, a pattern $p \in$ *patterns*$(\mathcal{P})$ is called τ*-frequent* if

$$\#\mathsf{supp}(p, D_{\mathcal{P}}) := |\{t \in D_{\mathcal{P}} : p \sqsubseteq t\}| \geq \tau.$$

In other words, a pattern p is τ-frequent if it appears in at least τ transactions of the database. When τ is clear from the context, we will call p only *frequent*. A pattern $p \in$ *patterns*$(\mathcal{P})$ is a *maximal frequent* pattern if p is frequent and all patterns $q \in$ *patterns*$(\mathcal{P})$ with $p \sqsubset q$ are not frequent. Given a database $D_{\mathcal{P}}$, we denote the set of all maximal frequent patterns by $\textsc{Max}(D_{\mathcal{P}}, \tau)$, i.e.,

$$\textsc{Max}(D_{\mathcal{P}}, \tau) = \{p \in \textit{patterns}(\mathcal{P}) : p \text{ is a maximal } \tau\text{-frequent pattern in } D_{\mathcal{P}}\}.$$

When the parameter τ is not part of the input but fixed to some integer, we write $\mathcal{P}^{\tau}$ to denote the resulting problem.

5.2.2 Concrete Frequency-Based Problems

All problems considered in this chapter are frequency-based problems. For the sake of brevity, we only define $\mathcal{L}$, *transactions*$(\cdot)$, *patterns*$(\cdot)$, and $\sqsubseteq$ for each problem (see, e.g., [11] for more thorough definitions).

The *maximal frequent itemset mining* problem, denoted as MaxFIS, is as follows: We have n labels $\mathcal{L} = \{1, \dots, n\}$; *transactions*(MaxFIS) and *patterns*(MaxFIS) are given by $2^{\mathcal{L}}$; $\sqsubseteq$ is the standard subset relationship $\subseteq$.

The *maximal frequent subsequence mining* problem, denoted as MaxSQS, is as follows: $\mathcal{L} = \{1, \dots, n\}$ is the set of labels. A *sequence* $S = \langle S_1, \dots, S_m \rangle$ of length m consists of m events S_i with $S_i \in \mathcal{L}$; we require that *each label appears at most once per sequence*. The sets *transactions*(MaxSQS) and *patterns*(MaxSQS) are the sets consisting of all sequences of arbitrary lengths. For two sequences $S =$

[1] A similar definition was given in Gunopulos et al. [87].

$\langle S_1, \dots S_r \rangle$ and $T = \langle T_1, \dots, T_k \rangle$, we have $T \sqsubseteq S$ if $k \leq r$ and there exist indices $1 \leq i_1 \leq \cdots \leq i_k \leq r$ such that $T_j = S_{i_j}$ for each $j = 1, \dots, k$.

Let $\mathcal{G}$ be a class of *vertex-labelled* graphs, which contain each label at most once. The *maximal frequent subgraph mining* problem, MaxFS($\mathcal{G}$), is as follows: We have n labels $\mathcal{L} = \{1, \dots, n\}$; *transactions*(MaxFS($\mathcal{G}$)) and *patterns*(MaxFS($\mathcal{G}$)) are given by all labelled graphs in $\mathcal{G}$ with labels from $\mathcal{L}$; $\sqsubseteq$ is the standard subgraph relationship for labelled graphs (i.e., we consider arbitrary subgraphs, not necessarily induced subgraphs).

In the remainder of the chapter, we will consider the following graph classes, all of which are *labelled and connected*:

- **T** — undirected trees,
- **BDG**^{b} — undirected graphs of bounded degree at most b,
- **BTW**^{w} — undirected graphs of bounded treewidth at most w,
- **PLN** — undirected planar graphs,
- **G** — general undirected graphs,
- **DAG** — directed acyclic graphs,
- **DirG** — directed graphs.

Throughout the chapter we will only consider labelled graphs *in which each label appears at most once*. In this restricted setting, the subgraph isomorphism problem can be solved in polynomial time. This a necessary condition for our reductions to work since Kimelfeld and Kolaitis [117] showed that for certain *unlabelled* graph classes $\mathcal{G}$, MaxFS($\mathcal{G}$) is not an NP-relation.

5.2.3 Enumeration Problems

An *enumeration relation* $\mathcal{R}$ is a set of strings $\mathcal{R} = \{(x, y)\} \subset \{0,1\}^* \times \{0,1\}^*$ such that

$$\mathcal{R}(x) := \{y \in \{0,1\}^* : (x, y) \in \mathcal{R}\}$$

is finite for every x. A string $y \in \mathcal{R}(x)$ is called a *witness* for x. We call $\mathcal{R}$ an NP-*relation* if (1) there exists a polynomial p such that $|y| \leq p(|x|)$ for all $(x, y) \in \mathcal{R}$, and (2) there exists a polynomial-time algorithm deciding if $(x, y) \in \mathcal{R}$ for any given pair (x, y).

Following [117], we define the following problems for an enumeration relation $\mathcal{R}$:

- $\mathcal{R}$-ENUMERATE: The input is a string x. The task is to output the set $\mathcal{R}(x)$ without repetitions.
- $\mathcal{R}$-EXTEND: The input is a string x and a set $Y \subseteq \mathcal{R}(x)$. The task is to compute a string y such that $y \in \mathcal{R}(x) \setminus Y$ or to output that no such element exists.
- $\mathcal{R}$-EXTENDIBLE: The input is a string x and a set $Y \subseteq \mathcal{R}(x)$. The task is to decide whether $\mathcal{R}(x) \setminus Y \neq \emptyset$.
- $\mathcal{R}$-EXTENDIBLE$\langle k \rangle$: The input is a string x and a set $Y \subseteq \mathcal{R}(x)$ with the restriction that $|Y| < k$. The task is to decide whether $\mathcal{R}(x) \setminus Y \neq \emptyset$.

The problem $\mathcal{R}$-EXTENDIBLE is the decision version of $\mathcal{R}$-EXTEND. Note that by repeatedly running an algorithm for $\mathcal{R}$-EXTEND, one can solve $\mathcal{R}$-ENUMERATE. Further observe that any algorithm solving $\mathcal{R}$-EXTEND can be used to solve $\mathcal{R}$-EXTENDIBLE.

Enumeration Complexity. Johnson et al. [113] introduced different notions for the complexity of enumeration problems. Let $\mathcal{R}$ be an enumeration relation. An algorithm solving $\mathcal{R}$-ENUMERATE is called an *enumeration algorithm*.

For enumeration problems, it might be the case that the output $\mathcal{R}(x)$ is exponentially larger than the input x. Due to this, measuring the running time of an enumeration algorithm only as a function of $|x|$ can be too restrictive; instead, one can include the size of $\mathcal{R}(x)$ in the complexity analysis. Then the running time of an algorithm is measured as function of $|x| + |\mathcal{R}(x)|$. This consideration gives rise to the following definitions:

- An enumeration algorithm runs in *total polynomial time* if its running time is polynomial in $|x| + |\mathcal{R}(x)|$.
- An enumeration algorithm has *polynomial delay* if the time spent between outputting two consecutive witnesses of $\mathcal{R}(x)$ is always polynomial in $|x|$.
- An enumeration algorithm runs in *incremental polynomial time* if on input x and after outputting a set $Y \subseteq \mathcal{R}(x)$, it takes time polynomial in $|x| + |Y|$ to produce the next witness from $\mathcal{R}(x) \setminus Y$.

Note that $\mathcal{R}$-ENUMERATE is in incremental polynomial time iff $\mathcal{R}$-EXTEND is in polynomial time. Additionally, observe that a polynomial total time algorithm can be used to decide if $\mathcal{R}(x) \neq \emptyset$ in polynomial time.

Relationship to Frequency-Based Problems. We note that frequency-based problems are special cases of enumeration problems. Let $\mathcal{P}$ be a frequency-based problem. We define the enumeration relation $\mathcal{R}$ corresponding to $\mathcal{P}$ by setting

$$\mathcal{R} = \{(x, y) : x = (D_{\mathcal{P}}, \tau), y \in \textsc{Max}(D_{\mathcal{P}}, \tau)\},$$

i.e., $\mathcal{R}$ consists of all possible databases $D_{\mathcal{P}}$, support thresholds τ and all maximal frequent patterns y for the tuples $(D_{\mathcal{P}}, \tau)$.

Observe that $\mathcal{R}(x) = \mathcal{R}(D_{\mathcal{P}}, \tau) = \textsc{Max}(D_{\mathcal{P}}, \tau)$ and thus $\mathcal{R}$-ENUMERATE is exactly the same problem as outputting all maximal frequent patterns in $\textsc{Max}(D_{\mathcal{P}}, \tau)$. The problem $\mathcal{R}$-EXTEND is to output a maximal frequent pattern in $\textsc{Max}(D_{\mathcal{P}}, \tau) \setminus Y$ for a given set of maximal patterns Y. The corresponding decision versions of the problems are $\mathcal{R}$-EXTENDIBLE and $\mathcal{R}$-EXTENDIBLE$\langle k \rangle$.

Since $\mathcal{R}$ and $\mathcal{P}$ yield the same enumeration problems, we write $\mathcal{P}$-ENUMERATE, $\mathcal{P}$-EXTENDIBLE, $\mathcal{P}$-EXTEND and $\mathcal{P}$-EXTENDIBLE$\langle k \rangle$. Often we will write $\mathcal{P}$ to denote the problem $\mathcal{P}$-ENUMERATE.

5.2.4 Counting Complexity

For a given enumeration relation $\mathcal{R}$, the function $\#\mathcal{R} : \{0,1\}^* \to \mathbb{N}$ returns the number of witnesses for a given string, i.e., $\#\mathcal{R}(x) = |\mathcal{R}(x)|$ for $x \in \{0,1\}^*$. The

complexity class #P (pronounced "sharp P") contains all functions $\#\mathcal{R}$ for which $\mathcal{R}$ is an NP-relation; it was introduced by Valiant [197]. A function $F : \{0,1\}^* \to \mathbb{N}$ is #P-*hard* if there exists a Turing reduction from every function in #P to F.

For two NP-relations $\mathcal{R}, \mathcal{Q} : \{0,1\}^* \to \mathbb{N}$, a *parsimonious reduction from* $\#\mathcal{R}$ *to* $\#\mathcal{Q}$ is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $\#\mathcal{R}(x) = \#\mathcal{Q}(f(x))$ for all $x \in \{0,1\}^*$. Note that a parsimonious reduction from a #P-hard problem $\mathcal{R}$ to a problem $\mathcal{Q}$ implies that $\mathcal{Q}$ is #P-hard.

An example for a #P-hard problem is counting *the number* of satisfying assignments of a SAT formula. Note an algorithm that counts SAT solutions can also decide if the given formula is satisfiable or not (by checking if the number of satisfying assignments is larger than 0). Hence, #P is a superset of NP.

In fact, Toda and Ogiwara [193] showed that all problems in the polynomial-time hierarchy can be solved in polynomial-time when one has access to an oracle for a #P-hard function.

Observe that an algorithm solving $\mathcal{R}$-ENUMERATE can solve $\#\mathcal{R}$ by counting the number of witnesses in its output.

5.3 Related Work

Counting Complexity. The study of counting problems was initiated when #P was introduced by Valiant [197]. Provan and Ball [169] showed #P-hardness for many graph problems such as counting the number of maximal independent sets in bipartite graphs. Later, more #P-hardness results were obtained for even more restricted graph classes [108, 196].

Johnson et al. [113] introduced the notions of polynomial total time, polynomial delay, and incremental polynomial time to obtain a better understanding of the computational complexity of enumeration problems.

Computational Complexity of Data Mining Problems. Gunopulos et al. [87] introduced a general class of problems similar to frequency-based problems. For this class of problems, they proved #P-hardness for mining frequent itemsets, and provided an algorithm to mine maximal frequent sets.

Yang [209] proved #P-hardness for determining the number of *maximal* frequent itemsets and other problems.

Theorem 5.2 (Yang [209])**.** *All of the following problems are* #P*-complete: MaxFIS, MaxSQS, MaxFS*($\mathbf{T}$)*, MaxFS*($\mathbf{G}$)*.*

Boros et al. [42] showed that given a set of maximal frequent itemsets Y, it is NP-complete to decide whether there exists another maximal frequent itemset that is not contained in Y.

Theorem 5.3 (Boros et al. [42])**.** *MaxFIS-extendible and MaxFIS-extend are* NP*-complete.*

Kimelfeld and Kolaitis [117] proved structural results on mining frequent subgraphs of certain graph classes. Their results allow to distinguish the computational complexities of MaxFS($\mathbf{T}$) and MaxFS($\mathcal{G}$) where $\mathcal{G}$ is either **G**, **PLN**, $\mathbf{BDG}^b$ with $b > 2$, or $\mathbf{BTW}^w$ with $w > 1$. This is also depicted in Figure 5.1.

Theorem 5.4 (Kimelfeld and Kolaitis [117])**.** *For fixed k, MaxFS($\mathbf{T}$)-extendible$\langle k \rangle$ can be solved in polynomial time.*

For fixed τ, MaxFS$^\tau$($\mathcal{G}$)-enumerate can be solved in polynomial time for any class of graphs $\mathcal{G}$ from Section 5.2.2.

The following problems are NP*-complete:*

- *MaxFS($\mathcal{G}$)-extendible for $\mathcal{G} \in \{\mathbf{G}, \mathbf{PLN}, \mathbf{BDG}^b, \mathbf{BTW}^w\}$ with $w \geq 1$ and $b \geq 3$.*
- *MaxFS($\mathcal{G}$)-extendible$\langle k \rangle$ for $\mathcal{G} \in \{\mathbf{G}, \mathbf{PLN}, \mathbf{BDG}^b, \mathbf{BTW}^w\}$ with $w \geq 2$ and $b \geq 3$ and for every $k \geq 3$.*

Kimelfeld and Kolaitis [117] also give computational hardness results for subgraph mining problems in which the set *patterns*$(\cdot)$ is more restricted than the set *transactions*$(\cdot)$. For example, they consider the computational complexity of mining maximal subtrees from planar graphs. They also consider mining unlabelled maximal subgraphs.

Mining Maximal Frequent Patterns. Many practical algorithms were proposed to mine maximal frequent patterns from different types of data such as itemsets [47, 93, 114], subsequences [14], trees [206, 216], and general graphs [124]. However, the main focus of those papers was not to investigate the computational complexity of these problems. See (for example) the book by Aggarwal [11] for many more references to algorithms for efficiently computing maximal frequent patterns.

Constraint-based Pattern Mining. Furthermore, many algorithms were proposed to mine frequent patterns with constraints on the structure of the patterns [38–40, 81, 85, 158, 167]. We cannot review all of them and instead refer to Han et al. [92] for references to papers on constrained pattern mining. Greco et al. [86] presented techniques for mining taxonomies of process models which can also be viewed as constraint-based pattern mining. The work on constraint programming for itemset mining by Raedt et al. [170] and follow-up work (e.g. [89]) can also be used to mine itemsets or other frequency-based problems with constraints.

5.4 Maximality-Preserving Reductions

In this section, we introduce maximality-preserving reductions and state some of their properties in Section 5.4.1. In Section 5.4.2, we prove reductions between the problems MaxFIS, MaxSQS, and MaxFS($\mathcal{G}$) for $\mathcal{G} \in \{\mathbf{T}, \mathbf{BDG}^3, \mathbf{G}\}$. Combining our reductions with the statements from Section 5.3, we arrive at the following theorem.

Theorem 5.5. *Our reductions imply the following hardness results:*

1. *For any fixed k, MaxFIS-extendible$\langle k \rangle$ can be solved in polynomial time.*
2. *For any fixed τ, MaxFIS$^\tau$-enumerate can be solved in polynomial time.*
3. *The problems MaxFS($\mathbf{G}$) and MaxFS($\mathbf{BDG}^3$) exhibit exactly the same hardness w.r.t. the notions of Sections 5.2.3 and 5.2.4. More concretely, let $\mathcal{P}$ be MaxFS($\mathbf{G}$) or MaxFS($\mathbf{BDG}^3$). Then the following statements are true:*
   - *$\mathcal{P}$-enumerate is $\#\mathsf{P}$-hard.*
   - *$\mathcal{P}$-extendible is NP-hard.*
   - *For $k > 2$, the problem $\mathcal{P}$-extendible$\langle k \rangle$ is NP-hard.*
   - *For fixed τ, the problem $\mathcal{P}^\tau$-enumerate is solvable in polynomial time.*

The proof of the theorem follows from our reductions later in this section and the theorems from Section 5.3.

5.4.1 Definition and Properties

We formally define maximality-preserving reductions to make explicit which properties are required by reductions in order to be useful for understanding the complexity of frequency-based problems w.r.t. the notions of Sections 5.2.3 and 5.2.4.

Definition 5.6. *Let $\mathcal{P}$ and $\mathcal{Q}$ be two frequency-based problems, let $D_\mathcal{P}$ be a database for $\mathcal{P}$, and let τ be a support threshold. A maximality-preserving reduction from $\mathcal{P}$ to $\mathcal{Q}$ defines an instance $(D_\mathcal{Q}, \tau)$ using a polynomial-time computable injective function $f\colon$ transactions($\mathcal{P}$) $\to$ transactions($\mathcal{Q}$) with the following properties:*

1. *$f($patterns($\mathcal{P}$)$) \subseteq$ patterns($\mathcal{Q}$).*
2. *For all $p, p' \in$ transactions($\mathcal{P}$), $p \sqsubseteq_\mathcal{P} p'$ if and only if $f(p) \sqsubseteq_\mathcal{Q} f(p')$.*
3. *The inverse $f^{-1}\colon$ transactions($\mathcal{Q}$) $\to$ transactions($\mathcal{P}$) of f can be computed in polynomial time.*
4. *$p \in$ Max$(D_\mathcal{P}, \tau)$ if and only if $f(p) \in$ Max$(D_\mathcal{Q}, \tau)$, where $D_\mathcal{Q} = f(D_\mathcal{P}) = \{f(t) : t \in D_\mathcal{P}\}$. Additionally, for all $q \in$ Max$(D_\mathcal{Q}, \tau)$ the preimage $f^{-1}(q)$ exists.*

Intuitively, the properties can be interpreted as follows: Property 1 asserts that f maps valid patterns from *patterns*($\mathcal{P}$) to valid patterns in *patterns*($\mathcal{Q}$); this condition is necessary if *patterns*($\mathcal{Q}$) $\subsetneq$ *transactions*($\mathcal{Q}$). Property 2 asserts that f maintains subset properties. Property 3 will be necessary to recover patterns in $\mathcal{P}$ from those found in $\mathcal{Q}$. Property 4 requires that the maximal frequent patterns in $D_\mathcal{P}$ are the same as those in $D_\mathcal{Q}$ under the mapping f; here, the database $D_\mathcal{Q}$ is given by applying the function f to each transaction in $D_\mathcal{P}$.

Properties. Observe that Property 4 implies that there exists a bijective relationship between the maximal frequent patterns in $D_\mathcal{P}$ and in $D_\mathcal{Q}$. Hence, we have $|\textsc{Max}(D_\mathcal{P}, \tau)| = |\textsc{Max}(D_\mathcal{Q}, \tau)|$. This shows that maximality-preserving reductions are special cases of parsimonious reductions and that they preserve $\#\mathsf{P}$-hardness.

In fact, maximality-preserving reductions are slightly stronger than parsimonious reductions. They do not only preserve the number of maximal frequent patterns in both databases, but they enable us to recover the maximal frequent patterns in $D_{\mathcal{P}}$ from those in D_Q: By injectivity of f and due to Property 4, we can reconstruct $\textsc{Max}(D_{\mathcal{P}}, \tau)$ in polynomial time from $\textsc{Max}(D_Q, \tau)$. Hence, maximality-preserving reductions can be used to argue about the complexity of extendibility problems as discussed in Section 5.2.3.

Further, note that by choice of D_Q in Property 4, D_Q has the same number of transactions as $D_{\mathcal{P}}$, and that no dependency within different transactions is created by the mapping f. Additionally, by Property 2, the support of a pattern p in $D_{\mathcal{P}}$ is a lower bound on the support of $f(p)$ in D_Q (since for each transaction $t \in D_{\mathcal{P}}$ with $p \sqsubseteq t$, $f(p) \sqsubseteq f(t)$).

However, although the number of transactions and *maximal* frequent patterns in both databases remains the same, the number of *frequent* patterns in D_Q might be exponentially larger than the number of frequent patterns in $D_{\mathcal{P}}$. For example, this is the case in the reduction in Lemma 5.9.

5.4.2 Reductions

In this section, we present three maximality-preserving reductions. Reductions similar to ones in Lemmas 5.7 and 5.8 were already presented by Yang [209], Kimelfeld and Kolaitis [117] and other authors. We only prove Property 4 of maximality-preserving reductions. The proofs of Properties 1–3 are straight-forward and follow from the definitions of the mapping f.

Reduction from MaxFIS to MaxFS($\mathbf{T}$). We show how to mine maximal itemsets by mining maximal subtrees.

Lemma 5.7. *There is a maximality-preserving reduction from MaxFIS to MaxFS($\mathbf{T}$).*

Proof. Consider MaxFIS with labels $\mathcal{L} = \{1, \dots, n\}$. We construct trees over labels from the alphabet $\mathcal{L}' = \{r, 1, \dots, n\}$, where r is the label of the root nodes in the trees. For simplicity, we do not distinguish between vertices and their labels.

Construction of f. An itemset $\{i_1, \dots, i_k\} \in$ *transactions*(MaxFIS) is mapped to a tree of depth 1 with root r and children $i_1, \dots, i_k$, i.e., the tree has an edge (r, i_j) for all $j = 1, \dots, k$.

Maximality-preserving. Observe that there exists a bijection between itemsets $I \subseteq \mathcal{L}$ and trees $f(I)$. Further note that for two itemsets I and J, $I \subseteq J$ if and only if $f(I) \subseteq f(J)$. It follows that an itemset I and a tree $f(I)$ must have the same supports in D_{MaxFIS} and in $D_{\text{MaxFS}(\mathbf{T})}$, respectively. The maximality then follows from the subset-property we observed. □

Reduction from MaxFIS to MaxSQS. We show how to mine maximal itemsets by mining maximal subsequences.

Lemma 5.8. *There exists a maximality-preserving reduction from MAXFIS to MAXSQS.*

Proof. Consider MAXFIS with labels $\mathcal{L} = \{1, \dots, n\}$ and assume the labels are ordered w.r.t. to some arbitrary, but fixed, order $\prec$.

Construction of f. Let $I = \{i_1, \dots, i_m\} \subseteq \mathcal{L}$ be any itemset with m items. Assume w.l.o.g. that the items in I are ordered w.r.t. the fixed order, i.e., $i_j \prec i_{j+1}$. Then I is mapped to the sequence $\langle i_1, \dots, i_m \rangle$ of length m.

Maximality-preserving. Observe that there exists a bijection between itemsets $I \subseteq \mathcal{L}$ and sequences $f(I)$ (under the fixed order). Further observe that for two itemsets I and J, $I \subseteq J$ if and only if $f(I) \sqsubseteq f(J)$. It follows that an itemset I and a sequence $f(I)$ must have the same supports in D_{MAXFIS} and in D_{MAXSQS}, respectively. The maximality then follows from the subset-property we observed. □

Reduction from MAXFS($\mathbf{G}$) to MAXFS($\mathbf{BDG}^3$). We show that mining maximal frequent subgraphs in general undirected graphs reduces to mining maximal frequent subgraphs in undirected graphs with degrees bounded by 3. Note that this is the tightest result we could hope for, since graphs with degree bounded by 2 are simply cycles or line graphs.

Lemma 5.9. *There exists a maximality-preserving reduction from MAXFS($\mathbf{G}$) to the problem MAXFS($\mathbf{BDG}^3$).*

Proof. Construction of f. Let $G = (V, E)$ be a graph with unbounded degree and suppose the vertices have unique labels $\mathcal{L} = \{1, \dots, n\}$. Denote the label of a vertex $v \in V$ by *label*(v). We construct a graph $G' = (V', E')$ with bounded degree 3 over the set of labels $\mathcal{L}' = \{1, \dots, n\}^2$.

Intuitively, the construction of f maps every original vertex $v \in V$ is split onto a line graph consisting of n vertices v_i, where each v_i has an additional non-line-graph-edge in G' iff vertices v and i share an edge in G.

Formally, for each vertex $v \in V$, we insert vertices $v_1, \dots, v_n$ into V' with edges (v_i, v_{i+1}) for $i = 1, \dots, n-1$. Each vertex v_i is labeled by $(\textit{label}(v), i)$. For each edge $(u, v) \in E$, we insert an edge $(u_{\textit{label}(v)}, v_{\textit{label}(u)})$ into G'.

Observe that the resulting graph $G' = f(G)$ indeed has bounded degree 3: Consider any vertex $v_i \in V'$. The vertex has at most 2 neighbors from the line graph $(v_1, \dots, v_n)$. The only additional edge it could have is to vertex $i_{\textit{label}(v)}$.

Maximality-preserving. We need to show that $p \in \text{MAX}(D_{\text{MAXFS}(\mathbf{G})}, \tau)$ iff $f(p) \in \text{MAX}(D_{\text{MAXFS}(\mathbf{BDG}^3)}, \tau)$. First, by construction of f, $\#\mathsf{supp}(f(p), D_{\text{MAXFS}(\mathbf{BDG}^3)}) = \#\mathsf{supp}(p, D_{\text{MAXFS}(\mathbf{G})})$; hence, $f(p)$ is frequent in $D_{\text{MAXFS}(\mathbf{BDG}^3)}$. Next, we show that if p is maximal then $f(p)$ is also maximal. For the sake of contradiction, suppose there exists a maximal frequent pattern q with $f(p) \sqsubset q$ in $D_{\text{MAXFS}(\mathbf{BDG}^3)}$. Then q must contain an edge (u_i, v_j) with $i = \textit{label}(v)$, $j = \textit{label}(u)$, which is not contained in $f(p)$.

Case 1: $u_i \in f(p)$ and $v_j \in f(p)$. Consider the graph $q' = f(p) \cup (u_i, v_j)$. Then $f^{-1}(q')$ exists and is frequent in $D_{\text{MAXFS}(\mathbf{G})}$ by Property 2. This contradicts the maximality of p.

Case 2: W.l.o.g. assume that $u_i \in f(p)$ and $v_j \notin f(p)$. Then, since q is maximal and by construction of f and $D_{\text{MAXFS}(\mathbf{BDG}^3)}$, q must contain the line graph L with vertices $v_1, \ldots, v_n$. Consider the graph $q' = f(p) \cup (u_i, v_j) \cup L$. Again by construction of f and $D_{\text{MAXFS}(\mathbf{BDG}^3)}$, q' has a preimage $p' = f^{-1}(q')$ which is frequent and satisfies $p \sqsubset p'$. This is a contradiction to the maximality of p.

Case 3: $u_i \notin f(p)$ and $v_j \notin f(p)$. Since q is connected and $f(p) \sqsubset q$, one of the first two cases must apply as well.

The second condition of Property 4 (the existence of the preimage $f^{-1}(q)$ for $q \in \text{MAX}(D_{\text{MAXFS}(\mathbf{BDG}^3)}, \tau)$) is also implied by the previous three case distinctions (observe that in the case distinctions, we only considered the preimages of concrete graphs q' and did not use the preimage of q). It left to prove that $f(p) \in \text{MAX}(D_{\text{MAXFS}(\mathbf{BDG}^3)}, \tau)$ implies $p \in \text{MAX}(D_{\text{MAXFS}(\mathbf{G})}, \tau)$. This can be also be done similarly to above by assuming that p is not maximal and then showing that this contradicts the maximality of $f(p)$. □

5.5 Constraining the Set of Patterns

In this section, we generalize frequency-based problems by allowing to constrain the set of patterns using a feasibility function. We introduce maximality-preserving reductions for this class of problems and prove that all problems discussed in this chapter exhibit exactly the same hardness after introducing the feasibility function.

5.5.1 Feasible Frequency-Based Problems

A *feasible frequency-based problem* (FFBP) $\mathcal{P}$ is a frequency-based problem with an additional polynomial-time computable operation $\phi\colon \textit{patterns}(\mathcal{P}) \to \{0, 1\}$ which can be described using constant space. For example, if G is a graph then $\phi(G)$ could compute whether G is connected or whether all vertices of G have constant degree. We note that the operation ϕ is part of the input for the problem and this is the reason for restricting the description length of the function to constant size; otherwise, the description length of the function might be larger than the database for the problem and this would defeat our goal of measuring the input in terms of the size of the database. We call ϕ the *feasibility function*.

Given a feasible frequency-based problem $\mathcal{P}$, a pattern $p \in \textit{patterns}(\mathcal{P})$ is a *feasible frequent pattern* (FFP) if p is frequent and $\phi(p) = 1$. The goal is to find all maximal FFPs; we denote the set of all FFPs by $\text{MAX}(D_{\mathcal{P}}, \tau, \phi_{\mathcal{P}})$. We define MAXFFIS, MAXFSQS, and MAXFFS($\mathcal{G}$) for a graph class $\mathcal{G}$ as before for maximal frequency-based problems.

Note that FFBPs are generalizations of frequency-based problems since by setting $\phi_{\mathcal{P}}$ to the function which is always 1, we obtain the underlying frequency-based problem.

The main result of this section is given in the following theorem.

Theorem 5.10. *The FFBP-versions of all problems discussed in this chapter exhibit exactly the same hardness w.r.t. the notions of Sections 5.2.3 and 5.2.4. More concretely, let $\mathcal{P}$ be any FFBP-problem discussed in this chapter. Then the following statements are true:*

- *$\mathcal{P}$-ENUMERATE is $\#\mathsf{P}$-hard.*
- *$\mathcal{P}$-EXTENDIBLE is NP-hard.*
- *For $k > 2$, the problem $\mathcal{P}$-EXTENDIBLE$\langle k \rangle$ is NP-hard.*
- *For fixed τ, the problem $\mathcal{P}^\tau$-ENUMERATE is solvable in polynomial time.*

Theorem 5.10 shows that the hierarchy given in Figure 5.1 for frequency-based problems collapses when a feasibility function is introduced to the problem. Note that many practical algorithms (like the *a priori* algorithm) for finding maximal frequent patterns allow to add such a feasibility function. Hence, our reductions give a theoretical justification why many of these algorithms can be extended to a broader range of problems.

The proof of the theorem follows from the reductions presented later in this section and the theorems from Section 5.3.

5.5.2 Maximality-Preserving Reductions for FFPPs

We start by defining maximality-preserving reductions between two FFBPs $\mathcal{P}$ and $\mathcal{Q}$.

Definition 5.11. *Let $\mathcal{P}$ and $\mathcal{Q}$ be two FFBPs. Let $D_\mathcal{P}$ be a database for $\mathcal{P}$, let $\phi_\mathcal{P}$ be the feasibility function for $\mathcal{P}$, and let τ be a support threshold.*

A maximality-preserving reduction from $\mathcal{P}$ to $\mathcal{Q}$ defines an instance $(D_\mathcal{Q}, \tau, \phi_\mathcal{Q})$ using an injective function $f\colon$ transactions$(\mathcal{P}) \to$ transactions$(\mathcal{Q})$ which is polynomial-time computable and which satisfies the following properties:

1. *$f($patterns$(\mathcal{P})) \subseteq$ patterns$(\mathcal{Q})$.*
2. *For all $p, p' \in$ transactions$(\mathcal{P})$, $p \sqsubseteq_\mathcal{P} p'$ if and only if $f(p) \sqsubseteq_\mathcal{Q} f(p')$.*
3. *The inverse $f^{-1}\colon$ transactions$(\mathcal{Q}) \to$ transactions$(\mathcal{P})$ of f can be computed in polynomial time.*
4. *$p \in$ MAX$(D_\mathcal{P}, \tau, \phi_\mathcal{P})$ if and only if $f(p) \in$ MAX$(D_\mathcal{Q}, \tau, \phi_\mathcal{Q})$, where $D_\mathcal{Q} = f(D_\mathcal{P}) = \{f(t) : t \in D_\mathcal{P}\}$. Additionally, for all $q \in$ MAX$(D_\mathcal{Q}, \tau, \phi_\mathcal{Q})$ the preimage $f^{-1}(q)$ exists.*

Note that compared to Definition 5.6, we only had to change Property 4 to assert that the maximal patterns are feasible. Further observe that in general the function $\phi_\mathcal{Q} = \phi_\mathcal{Q}(\phi_\mathcal{P}, f, f^{-1})$ constructed in the reduction will depend on $\phi_\mathcal{P}$, f and f^{-1}.

Properties. The rest of this subsection is devoted to proving properties of maximality-preserving reductions for FFBPs. First, we show that maximality-preserving reductions are transitive, i.e., that one can use multiple reductions in a row. Second, we show that maximality-preserving reductions for frequency-based problems imply maximality-preserving reductions for FFBPs.

The following lemma shows that maximality-preserving reductions for FFBPs are transitive. The main challenge will be the construction of the feasibility function.

Lemma 5.12. *Let $\mathcal{P}, \mathcal{Q}, \mathcal{R}$ be FFBPs. Assume there exist maximality-preserving reductions from $\mathcal{P}$ to $\mathcal{Q}$ via a function g and $\phi_{\mathcal{Q}}$, and from $\mathcal{Q}$ to $\mathcal{R}$ via a function h and $\phi_{\mathcal{R}}$. Then there exists a maximality-preserving reduction from $\mathcal{P}$ to $\mathcal{R}$.*

Proof. Let $D_{\mathcal{P}}$ and $\phi_{\mathcal{P}}$ be an instance for $\mathcal{P}$. We construct an instance (D^*, ϕ_*) for $\mathcal{R}$: We set $f\colon$ *transactions*$(\mathcal{P}) \to$ *transactions*$(\mathcal{R})$ to $f(p) = h(g(p))$ for $p \in$ *transactions*$(\mathcal{P})$. For a pattern $r \in$ *patterns*$(\mathcal{R})$, we set $\phi_*(r) = 1$ if and only if the following four conditions are satisfied: (1) $h^{-1}(r)$ and $f^{-1}(r)$ exist; (2) $\phi_{\mathcal{R}}(r) = 1$; (3) $\phi_{\mathcal{Q}}(h^{-1}(r)) = 1$; and (4) $\phi_{\mathcal{P}}(f^{-1}(r)) = 1$.

We check the properties from Definition 5.11. Property 1 and Property 2 are satisfied since f is the composition g and h. Property 3 holds since $f^{-1} = g^{-1} \circ h^{-1}$ and both g^{-1} and h^{-1} can be computed in polynomial time.

The rest of the proof is devoted to proving Property 4.

Let $p \in \textsc{Max}(D_{\mathcal{P}}, \tau, \phi_{\mathcal{P}})$. Then p is feasible w.r.t. $\phi_{\mathcal{P}}$. By the reduction from $\mathcal{P}$ to $\mathcal{Q}$, $g(p) \in \textsc{Max}(D_{\mathcal{Q}}, \tau, \phi_{\mathcal{Q}})$, where $D_{\mathcal{Q}} = g(D_{\mathcal{P}})$. Note that $g(p)$ is feasible w.r.t. $\phi_{\mathcal{Q}}$. Using the reduction from $\mathcal{Q}$ to $\mathcal{R}$, we obtain $r := h(g(p)) \in \textsc{Max}(D_{\mathcal{R}}, \tau, \phi_{\mathcal{R}})$, where $D_{\mathcal{R}} = h(D_{\mathcal{Q}})$; additionally, r is feasible w.r.t. $\phi_{\mathcal{R}}$. Now observe that $r = f(p)$ and that r is feasible w.r.t. the operation ϕ_* defined above. Note that r is frequent in D^* since for each transaction $t \in D_{\mathcal{P}}$ with $p \sqsubseteq_{\mathcal{P}} t$, $r = f(p) \sqsubseteq_{\mathcal{R}} f(t)$ by Property 2 of f. To prove that $r \in \textsc{Max}(D^*, \tau, \phi_*)$, it remains to show that r is maximal. Suppose not. Then there exists a pattern $r' \in \textsc{Max}(D^*, \tau, \phi_*)$ such that $r \sqsubset_{\mathcal{R}} r'$. Since r' is feasible, let $p' = f^{-1}(r')$. By Property 2 of f, we have that $p \sqsubset_{\mathcal{P}} p'$ and that p' is frequent since $p' \sqsubset_{\mathcal{P}} t$ for $t \in D_{\mathcal{P}}$ if and only if $f(p') = r' \sqsubset_{\mathcal{R}} f(t)$. This contradicts the maximality of p. Hence, we proved that $r \in \textsc{Max}(D^*, \tau, \phi_*)$.

Let $r \in \textsc{Max}(D^*, \tau, \phi_*)$. Since r is feasible w.r.t. ϕ_*, there exists $p = f^{-1}(r) \in$ *patterns*$(\mathcal{P})$ that is feasible w.r.t. $\phi_{\mathcal{P}}$. By Property 2, p is frequent in $D_{\mathcal{P}}$. It remains to show that p is maximal. We argue by contradiction. Suppose there exists a frequent pattern p' with $p \sqsubset p'$. Then $f(p') \in \textsc{Max}(D_*, \tau, \phi_*)$ by the previous paragraph, and $r \sqsubset f(p')$ by Property 2 of f. This contradicts the maximality of r. Hence, $p \in \textsc{Max}(D_{\mathcal{P}}, \tau, \phi_{\mathcal{P}})$. □

The next lemma shows that if for two frequency-based problems $\mathcal{P}$ and $\mathcal{Q}$ there exists a maximality-preserving reduction from $\mathcal{P}$ to $\mathcal{Q}$, then there also exists a reduction between the FFBP-versions of these problems.

Lemma 5.13. *Let $\mathcal{P}$ and $\mathcal{Q}$ be two frequency-based problems, and let $\mathcal{P}'$ and $\mathcal{Q}'$ be the FFBP-versions of those problems. Suppose there exists a maximality-preserving reduction from $\mathcal{P}$ to $\mathcal{Q}$ via a mapping g.*

Then there exists a maximality-preserving reduction from $\mathcal{P}'$ to $\mathcal{Q}'$.

Proof. Construction of f. We set $f \equiv g$. Given a pattern $q \in \mathit{patterns}(\mathcal{Q}')$, we set $\phi_{\mathcal{Q}'}(q) = 1$ iff $f^{-1}(q)$ exists and $\phi_{\mathcal{P}'}(f^{-1}(q)) = 1$.

Maximality-preserving. Note that Properties 1–3 of maximality-preserving reductions for f are satisfied since they are satisfied for g. We prove Property 4 of f.

Let $p \in \textsc{Max}(D_{\mathcal{P}'}, \tau, \phi_{\mathcal{P}'})$. We show that $f(p) \in \textsc{Max}(D_{\mathcal{Q}}, \tau, \phi_{\mathcal{Q}})$. Observe that $f(p)$ is feasible w.r.t. $\phi_{\mathcal{Q}'}$ since $f^{-1}(f(p)) = p$ is feasible w.r.t. $\phi_{\mathcal{P}'}$. Note that $f(p)$ is frequent in $D_{\mathcal{Q}'}$ by Property 2 of f. We need to argue that $f(p)$ is also maximal. Suppose this is not the case. Then there exists a pattern $q \in \textsc{Max}(D_{\mathcal{Q}'}, \tau, \phi_{\mathcal{Q}'})$ such that $f(p) \sqsubset q$. Since q is feasible, there exists a feasible pattern $p' = f^{-1}(q) \in \mathit{patterns}(\mathcal{P}')$. By Property 2, we have $p \sqsubset p'$. Additionally, the pattern p' is frequent in $D_{\mathcal{P}'}$: for each transaction $t \in D_{\mathcal{Q}'}$ with $q \sqsubset_{\mathcal{Q}'} t$, $p' \sqsubset_{\mathcal{P}'} f^{-1}(t)$ (by Property 2 of f and definition of $D_{\mathcal{Q}'}$). This contradicts the maximality of p.

Let $q \in \textsc{Max}(D_{\mathcal{Q}'}, \tau, \phi_{\mathcal{Q}'})$. Since q is feasible, $p = f^{-1}(q)$ exists and is feasible w.r.t. $\phi_{\mathcal{P}'}$. We show that $p \in \textsc{Max}(D_{\mathcal{P}'}, \tau, \phi_{\mathcal{P}'})$. Note that p is frequent in $D_{\mathcal{P}'}$ by Property 2 of f. We prove the maximality of p by contradiction. Suppose there exists a pattern $p' \in \textsc{Max}(D_{\mathcal{P}'}, \tau, \phi_{\mathcal{P}'})$ with $p \sqsubset p'$. Then by the previous paragraph the pattern $f(p')$ is a feasible frequent pattern in $D_{\mathcal{Q}'}$ with $q = f(p) \sqsubset f(p')$. This contradicts the maximality of q. □

5.5.3 Reductions

Reduction from graphs to feasible frequent itemsets. We show that any algorithm solving the MaxFFIS-problem can be used to mine maximal frequent subgraphs in general graphs.

Lemma 5.14. *There exists a maximality-preserving reduction from MaxFFS($\mathbf{G}$) to MaxFFIS.*

Proof. Let $D_{\text{MaxFFS}(\mathbf{G})}$ be a database consisting of vertex-labelled graphs from $\mathbf{G}$ with labels from $\{1, \dots, n\}$, let τ be a support threshold, let $\phi_{\text{MaxFFS}(\mathbf{G})}$ be a feasibility function.

Construction of f. For MaxFFIS we use the labels $\mathcal{L} = \{1, \dots, n\}^2$. Let $G = (V, E)$ be a graph from $D_{\text{MaxFFS}(\mathbf{G})}$. We construct an itemset $I(G) := f(G)$ by mapping the graph onto the labels of its edges, i.e., we construct an itemset $I(G) = \{(\mathit{label}(u), \mathit{label}(v)) : (u, v) \in E\}$.

Given an itemset $I \in \mathit{patterns}(\text{MaxFFIS})$, we set $\phi_{\text{MaxFFIS}}(I) = 1$ iff (1) $f^{-1}(I)$ exists and $\phi_{\text{MaxFFS}(\mathbf{G})}(f^{-1}(I)) = 1$, and (2) for each pair of tuples $(a, b), (c, d) \in I$ there exists a sequence $(a, b) = (e_1, e_1'), \dots, (e_k, e_k') = (c, d)$ of tuples $(e_i, e_i') \in I$ with the following property: For each pair of consecutive tuples (e_i, e_i') and (e_{i+1}, e_{i+1}'), there exists $\ell \in \{1, \dots, n\}$ with $\ell \in \{e_i, e_i'\}$ and $\ell \in \{e_{i+1}, e_{i+1}'\}$. Intuitively, Condition (2) of ϕ_{MaxFFIS} asserts that the graphs corresponding to feasible itemsets I must be connected.

Maximality-preserving. Note that any feasible frequent itemset in D_{MaxFFIS} corresponds to a frequent *connected* graph in $D_{\text{MaxFFS}(\mathbf{G})}$ due to the choice of ϕ_{MaxFFIS}. Observe that there exists a bijection between connected subgraphs G and feasible

itemsets $I(G) \subseteq \mathcal{L}'$. Further observe that for two frequent subgraphs G and H, $G \subseteq H$ if and only if $f(G) \subseteq f(H)$. It follows that a graph G and the itemset $I(G)$ have the same supports in $D_{\text{MAXFFS}(\mathbf{G})}$ and D_{MAXFFIS}, respectively. The maximality then follows from the subset-property we observed. □

Note that the reduction simplifies when $\phi_{\text{MAXFFS}(\mathbf{G})} \equiv 1$, i.e., when we consider the reduction from frequency-based problem MAXFS($\mathbf{G}$) to the FFBP MAXFFIS. Then the mapping f stays the same and ϕ_{MAXFFIS} only needs to check Condition (2). We believe that many algorithms for mining itemsets can be augmented with this feasibility function ϕ_{MAXFFIS} to mine graph patterns as we will discuss further in Section 5.6.

Next, observe that while Condition (2) looks rather technical, it can be easily implemented using a graph traversal or a union–find data structure. Additionally, when computing the union of two feasible patterns, an algorithm only needs to check if both patterns share any label.

Note also that the reduction above works as well for directed graphs (we just need to distinguish between edge labels $(\mathit{label}(u), \mathit{label}(v))$ and $(\mathit{label}(v), \mathit{label}(u))$). This immediately gives us the following lemma.

Lemma 5.15. *There exists a maximality-preserving reduction from MAXFFS(**DirG**) to MAXFFIS.*

Reduction from sequences to feasible DAGs. To finish the hierarchy of Figure 5.1, we need one more reduction, from MAXFSQS to MAXFFS(**DAG**).

Lemma 5.16. *There exists a maximality-preserving reduction from MAXFSQS to the problem MAXFFS(**DAG**).*

Proof. Let D_{MAXFSQS} be a database of sequences over labels from $\mathcal{L}$, let τ be a support threshold, and let ϕ_{MAXFSQS} be a feasibility function. Recall that a sequence contains each label at most once.

Construction of f. For MAXFFS(**DAG**) we use the same labels $\mathcal{L}$. Consider a sequence $S \in \mathcal{L}^r$ of length r such that $S_i \neq S_j$ for all $i \neq j$. This sequence is mapped to the graph $G(S)$ with vertices $V(S) = \{S_1, \dots, S_r\}$, where each vertex S_i is labelled by $\mathit{label}(S_i)$. The graph contains the directed edges

$$E(S) = \{(S_i, S_j) : i \in \{1, \dots, k-1\}, j > i\}.$$

Given a DAG $p \in \mathit{patterns}(\text{MAXFFS}(\mathbf{DAG}))$, we set $\phi_{\text{MAXFFS}(\mathbf{DAG})}(p) = 1$ iff $f^{-1}(p)$ exists and $\phi_{\text{MAXFSQS}}(f^{-1}(p)) = 1$.

Maximality-preserving. Clearly, Properties 1–3 of maximality-preserving reductions for f are satisfied. We prove Property 4.

Let S be sequence from $\text{MAX}(D_{\text{MAXFSQS}}, \tau, \phi_{\text{MAXFSQS}})$ of length r. We show that $G := f(S) \in \text{MAX}(D_{\text{MAXFFS}(\mathbf{DAG})}, \tau, \phi_{\text{MAXFFS}(\mathbf{DAG})})$. By construction of $D_{\text{MAXFFS}(\mathbf{DAG})}$ and due to Property 2, G is frequent in $D_{\text{MAXFFS}(\mathbf{DAG})}$. We need to argue that G is also

maximal; we do this by contradiction. Suppose there exists a feasible graph H such that $G \subset H$. Observe that adding any edge to G would introduce a cycle. Hence, H must contain more vertices than G. Since H is also feasible, it corresponds to a sequence $S' = f^{-1}(H)$ of length at least $r + 1$. By Property 2, S' is frequent and $S \sqsubset S'$. This contradicts the maximality of S.

Consider any maximal feasible frequent DAG $G \in D_{\text{MAXFFS}(\mathbf{DAG})}$. Since G is feasible, let $S = f^{-1}(G)$. Then the sequence $S = \langle v_1, \ldots, v_r \rangle$ must be frequent in D_{MAXFSQS} by the choice of f and the construction of $D_{\text{MAXFFS}(\mathbf{DAG})}$. Additionally, S must be maximal. Assume it is not. Then there exists a maximal sequence T with $S \sqsubset T$. By the argument of the previous paragraph, the graph $H = f(T)$ is maximal and frequent. But then we also have $G = f(S) \sqsubset f(T) = H$, which contradicts the maximality of G. □

5.6 Algorithms and Experiments

In this section, we discuss the practical consequences of our reductions and show that the reductions can be used to develop efficient real-world algorithms.

5.6.1 Reductions as Algorithms

In addition to providing us the theoretical understanding of the relationships between the problems, the reductions also provide us a direct way to solve a maximal frequent pattern mining problem in one domain by using a solver from the other domain. As an example, consider the reduction from the *frequency-based* problem MAXFS($\mathbf{G}$) to the FFBP MAXFFIS (Lemma 5.14) and let $D_{\text{MAXFS}(\mathbf{G})}$ be the graph database for an instance of MAXFS($\mathbf{G}$) and D_{MAXFFIS} be the transaction database built by the reduction.

The mapping of patterns f is straight forward, as we only need to generate a transaction for each graph, and an item for each unique edge label. The crux of the reduction lies in the feasibility function ϕ: it has to ensure that the returned frequent itemsets correspond to *connected* frequent subgraphs in the original problem. As the feasible frequent itemsets are a strict subset of all of the frequent subsets, we could simply prune out the results at the very end. A naïve algoritm for solving MAXFS($\mathbf{G}$) could then work as follows: (1) build D_{MAXFFIS} following Lemma 5.14; (2) compute all frequent itemsets from D_{MAXFFIS}; (3) prune out the non-feasible frequent itemsets; (4) prune out the non-maximal feasible frequent itemsets.

More efficient implementations are possible, however. In particular, we can add the feasibility constraint in the mining process, thus reducing the number of candidates to consider in each iteration. The connectedness constraint is not monotone, though: it is possible that two itemsets A and B do not correspond to connected subgraphs, while their union does (e.g., $A = \{(a, b), (c, d)\}$ and $B = \{(b, c), (d, e)\}$), and vice versa (e.g., $A = \{(a, b)\}$ and $B = \{(c, d)\}$). On the other hand, if C is a feasible (connected) frequent itemset in D_{MAXFFIS}, then it can be split into subsets of

any size that are frequent and feasible. This means that we can prune all infeasible itemsets at the same time when we prune away all infrequent itemsets. In other words, we can in fact work with *less* candidates (or at least with no more) than if we would be doing standard frequent itemset mining.

The final question in our case study is how to implement the feasibility check efficiently. Let $\mathit{label}(A)$ denote the set of unique (vertex) labels in an itemset A, i.e., $\mathit{label}(A) = \{l : \text{edge } (l, \cdot) \text{ or } (\cdot, l) \text{ is an item in } A\}$. Then $A \cup B$ is a connected (i.e., feasible) itemset iff $\mathit{label}(A) \cap \mathit{label}(B) \neq \emptyset$ and both A and B are connected (i.e., feasible). Hence, if we store the sets $\mathit{label}(A)$ together with the candidate itemsets, we only need to test the disjointness of these two sets to test the feasibility of $A \cup B$.

The above example should make clear that the reductions we presented can yield practical algorithms, and it is not too hard to see that similarly efficient algorithm can be designed following the reduction of Lemma 5.16. However, note that in this reduction it would not be a good idea to add single edges during the candidate generation; an efficient implementation would ensure that whole nodes with edges to all over vertices are added. This ensures that the preimage of the reduction exists at all times and that fewer infeasible candidates are generated.

To further validate our approach, we present some experimental evaluation of the above algorithm in the next subsection. Before that, let us however discuss the general approaches for using the maximality-preserving reductions.

The first observation is that the type of the feasibility constraint obviously has a big impact on the efficiency of the final algorithm. The study of constrained frequent pattern mining is well established (see, e.g., [92] or the references in Section 5.3), and that research gives characterizations of constraints that can be implemented efficiently in standard algorithms. Similarly, the constraint-programming algorithms for data analysis can often be easily adapted for the feasibility constraints used in frequency-based reductions.

The second observation concerns the number of (non-maximal) frequent itemsets. Our reductions are only guaranteed to preserve the maximality, and can, in principle, yield an exponentially larger number of non-maximal frequent (and feasible) itemsets. This would, naturally, make it practically infeasible to use the reductions together with standard frequent pattern mining algorithms. There are a few possible solutions to this. First, some of our reductions do not grow the number of feasible frequent patterns. This is, for example, the case with the reductions in Lemmas 5.7, 5.8, and 5.14. Second, a clever implementation of a reduction would only generate candidates which may be generated by the mapping from the reduction. This can dramatically decrease the number of possible candidates. In fact, if the implementation manages not to generate any candidates which have no preimage under the mapping from the reduction, then the number of possible candidates will not increase at all. We believe that this is possible for all reductions we presented. Third, the maximal frequent patterns can also be found by first finding all the maximal frequent and minimal infrequent patterns [88]. Unfortunately for this approach, we do not yet know the behaviour of minimal infrequent patterns under our reductions. We leave further studies in this for future work.

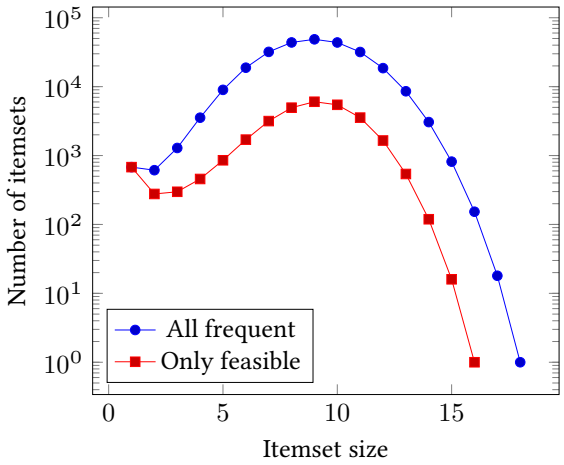


Figure 5.2: The number of frequent itemsets and feasible frequent itemsets when solving the MaxFS($\mathbf{G}$) problem using MaxFFIS algorithms. The y-axis is in logarithmic scale.

5.6.2 Experimental Evaluation

For the experimental evaluation, we implemented the reduction from MaxFS($\mathbf{G}$) to MaxFFIS (Lemma 5.14) in a custom version of the *a priori* algorithm [12]. The constraint on the feasible patterns was straight forward to implement, as discussed above.[2]

We tested our approach on a discussion forum data from the StackExchange forums.[3] The data contains 161 different question-answering forums (we excluded the meta-forums). We concentrated on the most recent year's activity, and constructed one graph for each forum where the users are the vertices and there is an edge between two users if one has answered or commented to the other's question or answer. The vertices are labelled uniquely using the global user-id. The data has 1 627 946 different users, and in total 8 264 675 uniquely-labelled edges. Hence, the dataset does not pose a significant problem for frequent itemset mining algorithms.

We wanted to study the effects the constraint has for the number of candidates. Recall that the constraint is used to enforce that we find only connected subgraphs. In Figure 5.2, we show the number of frequent itemsets and feasible frequent itemsets of different sizes with minimum frequency 3.

As can be seen from Figure 5.2, the total number of frequent itemsets is approximately ten times the number of feasible candidates, indicating that the feasibility constraint allows us to prune significant amounts of candidates (there are no feasible candidates of size 17 or 18). In total, the data has 265 111 frequent itemsets, of which 29 752 were feasible and 549 were maximal feasible itemsets.

[2]The code and sample data are available from `https://people.mpi-inf.mpg.de/~pmiettin/frequency-based-reductions/`.

[3]`https://archive.org/details/stackexchange`

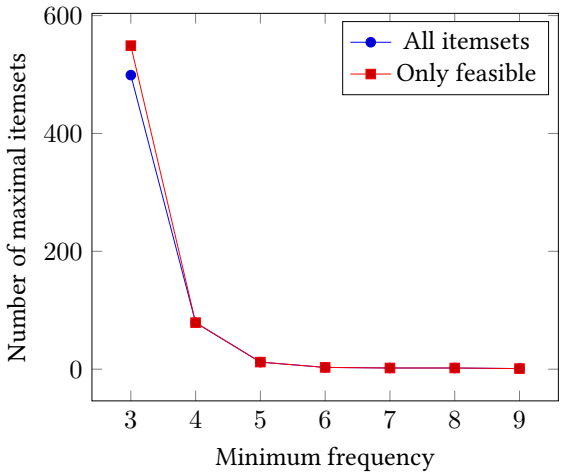


Figure 5.3: The number of maximal feasible frequent itemsets with different minimum support thresholds.

The number of maximal frequent itemsets and maximal feasible frequent itemsets with respect to different minimum thresholds is presented in Figure 5.3. We can see that their numbers are mostly aligned, with the number of maximal itemsets dropping almost exponentially as the minimum threshold increases. No pattern has support higher than 9.

5.7 Conclusion

We studied the computational complexity of several frequency-based problems and provided a unifying framework for the existing computational hardness results. We showed that when considering a generalized version of frequency-based problems, the computational hardness of many frequency-based problems collapses. Additionally, our reductions give a formal explanation why algorithms similar to the *a priori* algorithm can be used for such a wide range of problems by only slightly adjusting the candidate generation.

In the future it will be interesting to study the computational complexity of frequency-based problems in which labels can appear multiple times. A daunting question is whether the following two problems exhibit the same hardness: Mining subsequences without the restriction that each label appears only once, and mining graphs with possibly multiple vertices of the same label.

The reductions we provide hint that many practical algorithms for frequency-based problems can be augmented to solve more complicated problems. We provided such an example in Section 5.6. It will be interesting to see if our insights can lead to more efficient algorithms for the problems we considered or to algorithms which can solve a wider range of problems.

CHAPTER 6

Efficient Distributed Workload (Re-)Embedding

Modern networked systems are increasingly reconfigurable and this is enabling *demand-aware* infrastructures whose resources can be adjusted according to the workload they currently serve. Such dynamic adjustments can be exploited to improve network utilization and hence performance, by *moving* frequently interacting communication partners closer, e.g., collocating them in the same server or datacenter. However, dynamically changing the embedding of workloads is algorithmically challenging: communication patterns are often not known ahead of time, but must be *learned*. During the learning process, overheads related to unnecessary moves (i.e., re-embeddings) should be minimized. We study a fundamental model which captures the tradeoff between the benefits and costs of dynamically collocating communication partners on ℓ servers, in an online manner. Our main contribution is a distributed online algorithm which is asymptotically almost optimal, i.e., almost matches the lower bound (also derived in this chapter) on the competitive ratio of any (distributed or centralized) online algorithm. As an application, we show that our algorithm can be used to solve a distributed union–find problem in which the sets are stored across multiple servers.

6.1 Introduction

Along with the trend towards more *data centric* applications (e.g., online services like web search, social networking, financial services as well as emerging applications such as distributed machine learning [128, 159]), comes a need to *scale out* such applications, and distribute the workload across multiple servers or even datacenters. However, while such parallel processing can improve performance, it can entail a non-trivial load on the interconnecting network. Indeed, distributed cloud

applications, such as batch processing, streaming, or scale-out databases, can generate a significant amount of network traffic [148].

At the same time, emerging networked systems are becoming increasingly flexible and thereby provide novel opportunities to mitigate the overhead that distributed applications impose on the network. In particular, the more flexible and dynamic resource allocation (enabled, e.g., by virtualization) introduces a vision of *workload-aware* infrastructures which optimize themselves to the demand [24]. In such infrastructures, communication partners which interact intensively, may be *moved* closer (e.g., collocated on the same server, rack, or datacenter) in an adaptive manner, depending on the demand. This "re-embedding" of the workload allows to keep communication local and reduce costs. Indeed, empirical studies have shown that communication patterns in distributed applications feature much locality, which highlights the potential of such self-adjusting networked systems [30, 83, 178].

However, leveraging such resource reconfiguration flexibilities to optimize performance, poses an algorithmic challenge. *First,* while collocating communication partners reduces communication cost, it also introduces a *reconfiguration cost* (e.g., due to virtual machine migration). Thus, an algorithm needs to strike a balance between the benefits and the cost of such reconfigurations. *Second,* as workloads and communication patterns are usually not known ahead of time, reconfiguration decisions need to be made in an *online* manner, i.e., without knowing the future. We are hence in the realm of online algorithms and competitive analysis.

We study the fundamental tradeoff underlying the optimization of such workload-aware reconfigurable systems. In particular, we consider the design of an online algorithm which, without prior knowledge of the workload, aims to minimize communication cost by performing a small number of *moves* (i.e., migrations). In a nutshell (more details will follow below), we consider a *communication graph* between n vertices (e.g., virtual machines) which can be perfectly partitioned among a set of ℓ servers (resp. racks or datacenters) of a given *capacity*. We assume that the *communication patterns*, which partition the communication graph, consist of n/ℓ vertices and that once the whole communication graph was revealed, each server must contain exactly one communication pattern.

The communication graph is initially unknown and revealed to the algorithm in an online manner, edge-by-edge, by an adversary who aims to maximize the cost of the given algorithm. The cost here consists of *communication cost* and *moving cost*: The algorithm incurs one unit cost if the two endpoints (i.e., communication partners) of the request belong to different servers. After each request, the algorithm can reconfigure the infrastructure and move communication endpoints from one server to another, essentially *repartitioning* the communication partners; however, each move incurs a cost of $\alpha > 1$.

In other words, we consider the problem of *learning a partition*, i.e., an optimal assignment of communication partners to servers, at low communication and moving cost. Interestingly, while the problem is natural and fundamental, not much is known today about the algorithmic challenges underlying this problem, except

for the negative result that no good competitive algorithm can exist if communication partners can change arbitrarily over time [23]. This lower bound motivates us to focus on the online *learning variant* where the communication partners are unknown but fixed. At the same time, as we will show, the problem features interesting connections to several classic problems. Specifically, the problem can be seen as a *distributed* version of classic online caching problems [189] or an *online* version of the k-way partitioning problem [184].

6.1.1 Our Contributions

We initiate the study of a fundamental problem, how to learn and re-embed workload in an online manner, with few moves. We make the following main contributions.

We present a distributed $O((\ell \log \ell \log n)/\varepsilon)$-competitive online algorithm for servers of capacity $(1+\varepsilon)n/\ell$, where $\varepsilon \in (0, 1/2)$. We allow the servers to have $\varepsilon n/\ell$ more space than is strictly needed to embed its corresponding communication pattern (which is of size n/ℓ); we denote this additional space as *augmentation*. Such augmentation is also needed, as our lower bounds discussed next show.

We show that there are also limitations of what online algorithms can achieve in our model: We derive a lower bound of $\Omega(1/\varepsilon + \log n)$ on the competitive ratio of any deterministic online algorithm given servers of capacity at least $(1+\varepsilon)n/\ell$. This lower bound has several consequences: (1) To obtain $O(\log n)$-competitive algorithms, the servers must have $\Omega(n/(\ell \log n))$ augmentation. (2) If the servers have $\Omega(n/\ell)$ augmentation (e.g., each server has 10% more capacity than the size of its communication pattern), our algorithm is optimal up to an $O(\ell \log \ell)$ factor. Thus, our results are particularly interesting for large servers, e.g., in a wide-area networking context where there is usually only a small number of datacenters where communication partners can be collocated (e.g., $\ell = 20$): if each datacenter ("server") has augmentation $0.1 \cdot n/\ell$, our algorithm is optimal up to constant factors.

The distributed algorithms we present not only provide good competitive ratios but they are also highly efficient w.r.t. the network traffic they cause. In fact, we show that for $\ell = O(\sqrt{\varepsilon n})$ servers, running the algorithms introduces only little overhead in network traffic and that this overhead is asymptotically negligible (see Section 6.5.1).

While the previous algorithms require exponential time, we also present polynomial-time algorithms with a competitive ratio of $O((\ell^2 \log n \log \ell)/\varepsilon^2)$ in Section 6.5.2.

As a sample application of our newly introduced model we present a distributed union–find data structure [78, 192] (also known as disjoint-set data structure or merge-find data structure) in Section 6.7.1: There are n items from a universe which are distributed over ℓ servers; each server can store at most $(1+\varepsilon)n/\ell$ items and each item belongs to a unique set. The operation *union* allows to merge two sets. In our setting, we require that items from the same set must be assigned to the same server. To reduce the network traffic, our goal is to minimize the number of item

moves during union operations. For example, when two sets are merged which are assigned to different servers, then the items of one of the sets must be reassigned to another server. We compare against an optimal offline algorithm which knows the initial assignment of all items and all union operations in advance. We obtain the same competitive ratios as above. We believe that this distributed union–find data structure will be useful as a subroutine for several problems such as merging duplicate websites in search engines [46].

We also show that our algorithms solve an online version of the k-way partition problem in Section 6.7.2.

6.1.2 Organization

We introduce our model formally in Section 6.2. To ease the readability, we first explore centralized online algorithms that efficiently collocate communication patterns for $\ell = 2$ servers in Section 6.3, and then study the general case of $\ell > 2$ servers in Section 6.4. In Section 6.5 we show how the previously derived centralized algorithms can be made distributed and how the algorithm can be implemented in polynomial time at the cost of a slightly worse competitive ratio. We provide the lower bounds in Section 6.6. Section 6.7 provides a distributed union–find data structure and a result for online k-way partitioning; these problems serve as sample applications of the problem we study. After reviewing related work in Section 6.8, we conclude our contribution in Section 6.9.

6.2 Model

We start by formally introducing the model which we will be studying in this chapter. We consider a set of vertices V (e.g., a set of virtual machines) which interact according to an initially unknown communication pattern, which can be represented as a communication graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The vertices of G are partitioned into ℓ sets $V_0, \ldots, V_{\ell-1}$ where each V_i, forming a connected communication component (the workload), has size[1] n/ℓ; the connected components of G coincide with the sets V_i. The sets V_i are the communication patterns which need to be recovered by the online algorithm, henceforth called *ground truth components*.

The communicating vertices V need to be assigned to ℓ *servers* $S_0, \ldots, S_{\ell-1}$. Accordingly, we define an *assignment* (the embedding) which is a function from the vertices to the servers. The *load* of a server S_j is the number of vertices that are assigned to it. An assignment is *valid* if each server has load at most $n/\ell + K$ and we call $n/\ell + K$ the *capacity* of the servers and K the *augmentation*. If $K = 0$, the total server capacity exactly matches the number of vertices. The *available capacity* of a server is the difference between the server's capacity and its load. An assignment

[1] Note that in general n/ℓ is not always an integer and we would have to take rounding into account. However, we ignore this technicality for better readability.

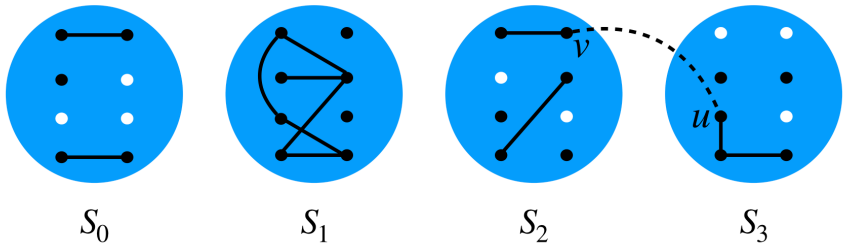


Figure 6.1: An illustration of the model we consider. In the picture there are $\ell = 4$ servers each depicted by a blue circle. Vertices assigned to a server are represented by black dots whereas white dots represent unused server capacities. Note that there are $n = 24$ vertices and each server has capacity $(1+\varepsilon)n/\ell = 8$ for $\varepsilon = 1/3$. In the picture, server S_0 has load 5 and server S_1 has load 8. When two vertices communicated, we draw an edge between them depicted by a black line. Observe how this naturally gives rise to connected components and note that S_1 contains a ground truth component of size $n/\ell = 6$. If the adversary were to insert the edge (u, v) next, the algorithm could, for example, move the connected component containing v to S_3 at cost 2α.

is *perfectly balanced* if each server has load exactly n/ℓ. We assume that when the algorithm starts, we have a perfectly balanced assignment. We will write $V(S_j)$ to denote the set of vertices assigned to server S_j and $V_{\text{init}}(S_j)$ for the set of vertices *initially* assigned to server S_j. We say that an assignment is a *perfect partitioning* if it satisfies $\{V(S_0), \dots, V(S_{\ell-1})\} = \{V_0, \dots, V_{\ell-1}\}$, i.e., the vertices on the servers coincide with the connected components of G.

The communication graph $G = (V, E)$ is revealed by an adversary in an *online manner*, as a sequence of edges $\sigma = (e_1, \dots, e_r)$, where r denotes the number of communication requests and $e_i \in E$ for each i. Note that the adversary can only provide edges which are present in E and that each edge can appear multiple times in the sequence of edges. We assume that the sequence of the edges provided by the adversary reveals the ground truth components V_i, i.e., after having seen all edges in σ the algorithm can compute the connected components of G which (by assumption) coincide with the ground truth components V_i. We present an illustration of the model in Figure 6.1.

Now an online algorithm must iteratively change the assignment such that eventually the assignment is a perfect partitioning.

The reassignment needs to be done while minimizing certain communication and migration cost. If an edge $e = (u, v)$ provided by the adversary has both endpoints in the same server S_i at the time of the request, an algorithm incurs no costs. If u and v are in different servers S_i and S_j, then their communication cost is 1. Reassigning, i.e., *moving*, a vertex u from a server S_i to a server S_j costs $\alpha > 1$.

When measuring the cost of an online algorithm, we will compare against an optimal offline algorithm denoted by OPT. OPT has *a priori* knowledge of the

communication graph $G = (V, E)$ as well as the given the sequence of all edges $\sigma = (e_1, \dots, e_r)$. In other words, OPT can compute the assignment of vertices to servers which provides the minimum migration cost from the initial assignment.

Now let the cost paid by an online algorithm be denoted by ONL and let the cost of the optimal offline algorithm be denoted by OPT. We consider the design of an online algorithm ONL which minimizes the (strict) *competitive ratio* defined as $\frac{\text{ONL}}{\text{OPT}}$.

The Role of Connected Components We will briefly discuss how connected components are induced by subsequence of σ and how we will treat connected components in our algorithms. We then give a reduction which helps us to avoid considering communication costs in our proofs.

Recall that the adversary provides a sequence of edges σ to an algorithm in an online manner. As this happens, an algorithm can keep track of all edges it has seen so far. Let this set of edges be E'. Using the edges in E', the algorithm can compute the connected components $C_1, \dots, C_q$ which are induced by E'. Here, q denotes the current number of connected components.

To obtain a better understanding of the relationship between the connected components C_i and the ground truth components V_j, we make four observations: (1) When the algorithm starts, all connected components $C_i = \{v_i\}$ only consist of single vertices (because σ has not yet revealed any edges). (2) When a previously unknown edge $e = (u, v)$ is revealed which has its endpoints in different connected components C_u and C_v, these connected components get merged. (3) Suppose a subsequence of σ induces $q > \ell$ connected components C_i (i.e., σ has not yet revealed the whole graph G). Then for each ground truth component V_j there exists a subset $\mathbb{C} \subset \{C_1, \dots, C_q\}$ of the connected components such that $V_j = \bigcup_{C \in \mathbb{C}} C$. (4) When an algorithm terminates (and, hence, σ revealed all edges in E), there exists a one-to-one correspondence between the connected components C_i and the ground truth components V_j.

By assumption on the input from the adversary, when all of σ was revealed, E' reveals the ground truth components $V_0, \dots, V_{\ell-1}$. Thus, in total there will be exactly $n - \ell$ edges connecting vertices from different connected components.

All of the algorithms we consider in this chapter have the property that they always assign vertices of the same connected component to the same server. This property implies that the communication cost paid by such an algorithm is bounded by its moving cost (we prove this in the following lemma). Hence, in the rest of the chapter we only need to bound the moving costs of our algorithms to obtain a bound on their total costs.

Lemma 6.1. *Suppose an algorithm $\mathcal{A}$ always assigns all vertices of the same connected component to the same server and pays $\mathcal{C}$ for moving vertices. Then its communication cost is at most $\mathcal{C}$. Furthermore, its total cost is at most $2\mathcal{C}$.*

Proof. Suppose the adversary provides an edge (u, v). We distinguish two cases. *Case 1:* u and v are assigned to the same server. Then $\mathcal{A}$ does not pay any commu-

nication costs. *Case 2:* u and v are assigned to connected components C_u and C_v on different servers. Then the algorithm needs to pay 1 communication cost. However, in this case $\mathcal{A}$ must move C_u or C_v to a different server at the cost of at least $\alpha > 1$. Hence, the moving cost is larger than the communication cost. We conclude that $\mathcal{A}$'s total communication cost is at most $\mathcal{C}$. By summing the two quantities, we obtain the second claim of the lemma. □

While in Lemma 6.1 we have shown that algorithms which always collocate connected components immediately are efficient w.r.t. their total cost, in Section 6.6.1 we show that any efficient algorithm must satisfy a similar (slightly more general) property.

Throughout the rest of the chapter, we write $|C|$ to denote the number of vertices in a connected component C. For a vertex u, we write C_u to denote the connected component C which contains u.

6.3 Online Partition for Two Servers

In this section, we consider the problem of learning a communication graph with few moves with *two* servers. As we will see later, the concepts introduced in this section will be useful when solving the problem with $\ell > 2$ servers. We derive the following result.

Theorem 6.2. *Consider the setting with two servers of capacity $(1+\varepsilon)n/2$ for $\varepsilon \in (0,1)$, i.e., the augmentation is $\varepsilon n/2$. Then there exists an algorithm with competitive ratio $O((\log n)/\varepsilon)$.*

The proof is organized as follows. We first characterize the optimal solution by OPT in Section 6.3.1. We then present an algorithm which is efficient whenever OPT incurs "significant cost", in Section 6.3.2. In Section 6.3.3, we describe an algorithm which is efficient whenever the solution by OPT is "cheap". We prove Theorem 6.2 via a combination of the two algorithms in Section 6.3.4.

6.3.1 Costs of OPT

The following lemma gives a precise characterization of the cost paid by OPT in the two server case. It introduces a parameter Δ which equals the number of vertices moved by OPT and which we will be using throughout the rest of this section.

Lemma 6.3. *Suppose $\ell = 2$ and the vertices initially assigned to the servers S_i are given by the sets $V_{\text{init}}(S_i)$ for $i = 0, 1$. Then the cost of OPT is $2\alpha\Delta$, where*

$$\Delta = \min\{|V_{\text{init}}(S_0) \cap V_0|, |V_{\text{init}}(S_0) \cap V_1|\}.$$

It follows immediately that $\Delta \leq n/4$ (as $|V_{\text{init}}(S_0)| = n/2$).

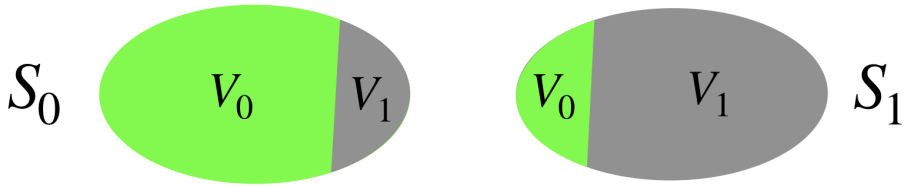


Figure 6.2: The initial assignment considered in the proof of Lemma 6.3. The green and grey areas of the servers correspond to subsets of V_0 and V_1. Server S_0 (S_1) contains most of the vertices from V_0 (V_1). Here, OPT would move the green part from S_1 to S_0 and the grey part from S_0 to S_1.

Proof. Recall that our model forces OPT to provide a final assignment satisfying $\{V(S_0), V(S_1)\} = \{V_0, V_1\}$, i.e., OPT must produce a final assignment which coincides with the ground truth components (even if paying for each communication request individually and not relocating any vertices might be cheaper). Thus, we can assume that OPT performs all vertex moves in the beginning, to avoid paying any communication cost. Since the edge sequence $\sigma = (e_1, \ldots, e_r)$ provided by the adversary is assumed to reveal the connected components V_0 and V_1, OPT can compute V_0 and V_1 before it performs any moves.

As there are only two servers, one of them must contain at least half of the vertices from V_0 in the initial assignment. Now let us first assume that this server is S_0; this setting is illustrated in Figure 6.2. In this case, OPT can move the Δ vertices in $V_{\text{init}}(S_0) \cap V_1$ to S_1 and those in $V_{\text{init}}(S_1) \cap V_0$ to S_0. It is easy to verify that this yields an assignment satisfying $\{V(S_0), V(S_1)\} = \{V_0, V_1\}$ and that the moving cost is minimized. Further, the cost for this reassignment is exactly $2\alpha\Delta$.

The second case where S_1 contains more than half of the vertices from V_0 in the initial assignment is symmetric. □

While in Lemma 6.3 we have presented the lower bound w.r.t. server S_0, we could also express the lower bound in terms of server S_1. We then obtain the following equality:

$$\Delta = \max_{i=0,1} \min_{j=0,1} |V_{\text{init}}(S_i) \cap V_j|.$$

6.3.2 The Small–Large–Rebalance Algorithm

A natural idea to obtain a small number of vertex moves is to proceed as follows. Whenever two vertices u and v belonging to different connected components communicate, the algorithm merges their connected components. If the two components were already assigned to the same server, no vertex moves are required. If u and v are assigned to different servers, we move the smaller connected component

to the server of the larger connected component. This algorithm is efficient in that it never performs more than $O(n \log n)$ vertex moves (see Lemma 6.4).

However, the algorithm could require much augmentation, as it does not account for server capacities. Thus, we propose the following extension called the *Small–Large–Rebalance Algorithm*: Whenever a server exceeds its capacity, the algorithm computes a perfectly balanced assignment of the vertices which respects the previously observed connected components; we call this a *rebalancing step*. We provide pseudocode in Algorithm 4.

Section 6.5.2.1 shows how such a rebalancing step can be implemented in $O(n^2)$ time. Later, we show that there can be at most $O((\log n)/\varepsilon)$ such rebalancing steps which implies that the total running time Algorithm 4 is $O((n^2 \log n)/\varepsilon)$.

Note that Algorithm 4 also works in the setting with ℓ servers for $\ell > 2$. We will analyze this more general algorithm in Section 6.4.4.

6.3.2.1 Analysis

To analyze Algorithm 4, we first consider the algorithm from the first paragraph which does not have the rebalancing step. When the algorithm moves a smaller component to the server of a larger component, we call this a *small-to-large step*.

Lemma 6.4. *Consider the algorithm which always moves the smaller connected component to the server of the larger connected component when it obtains an edge between vertices from different connected components. The algorithm moves each vertex at most $O(\log n)$ times. Its total number of vertex moves is $O(n \log n)$.*

Proof. Consider any vertex v. We use the following accounting: Whenever v is in the a smaller component that is moved, add a token to v. Now observe that whenever v gains a token, the size of its component at least doubles. This implies that v can be in the smaller component only $O(\log n)$ times. Thus, v cannot accumulate more than $O(\log n)$ tokens. Since this holds for each of the n vertices, the total number of moves is $O(n \log n)$. □

The following lemma provides the analysis for Algorithm 4 which performs small-to-large steps and rebalancing steps.

Lemma 6.5. *Suppose both servers have capacity $(1+\varepsilon)n/2$, i.e., the augmentation is $\varepsilon n/2$ for $\varepsilon \in (0, 1)$. Then Algorithm 4 performs $O((\log n)/\varepsilon)$ rebalancing steps and $O((n \log n)/\varepsilon)$ vertex moves.*

Proof. We prove the bound on the number of vertex moves; the claim about the number of rebalancing steps is proved along the way. Note that all vertex moves performed by the algorithm originate from either small-to-large steps or from rebalancing steps. We bound the number of each of these vertex moves separately.

Note that the token-based argument from Lemma 6.4 still applies to the small-to-large steps of Algorithm 4. This implies that the total number of vertex moves due small-to-large steps is $O(n \log n)$.

Input: A sequence of edges $\sigma = (e_1, \ldots, e_r)$

1: **procedure** SMALLLARGEREBALANCE($e_1, \ldots, e_r$)
2: **for** $i = 1, \ldots, r$
3: $(u, v) \leftarrow e_i$
4: **if** C_u and C_v are not assigned to the same server
5: ▷ We must move C_u and C_v to the same server.
6: Assume w.l.o.g. that $|C_u| \leq |C_v|$
7: **if** the server of C_v has available capacity $|C_u|$
8: Move C_u to the server of C_v
9: ▷ Small-to-large step
10: **else** ▷ Rebalancing step
11: Move to a perfectly balanced assignment respecting the connected components
12: Merge C_u and C_v

Algorithm 4: The Small–Large–Rebalance Algorithm

Now consider the vertex moves caused by the rebalancing steps and recall that the initial assignment is perfectly balanced. Whenever a server exceeds its load, the small-to-large steps of the algorithm must have moved at least $\varepsilon n/2$ vertices (because the augmentation of one of the servers is exceeded). This can only happen $O((n \log n)/(\varepsilon n)) = O((\log n)/\varepsilon)$ times since the total number of vertex moves due to small-to-large steps is $O(n \log n)$. Hence, the number of rebalancing steps is at most $O((\log n)/\varepsilon)$. Since each rebalancing step performs $O(n)$ vertex moves, the lemma follows. □

6.3.2.2 More Efficient Rebalancing

We next propose a better rebalancing strategy which makes Algorithm 4 more efficient. So far, we used $\Theta(n)$ vertex moves for each rebalancing operation at the cost of $\Theta(\alpha n)$. We now bring the rebalancing cost down to $O(\mathrm{OPT})$.

We adjust Algorithm 4 in the following way: Instead of rebalancing by taking *any* perfectly balanced assignment respecting the connected components (Line 11), we choose a perfectly balanced assignment respecting the connected components *which minimizes the number of vertex moves from the initial solution.* We call such an assignment *cheap*.

To find a cheap assignment, the algorithm could simply do the following: (1) Recall the initial assignment. (2) Exhaustively enumerate all perfectly balanced assignments respecting the connected components. (3) Among all of these assignments find one which is cheap. While such a simple algorithm can in principle be computationally costly, we can here exploit the online model of computation which allows us unlimited computational power. In Section 6.5.2 we show how less efficient re-

balancing strategies can be implemented in polynomial time and we obtain slightly worse competitive ratios.

With the improved rebalancing strategy, we obtain Proposition 6.6.

Proposition 6.6. *Suppose all servers have capacity $(1+\varepsilon)n/2$, $\varepsilon > 0$. Then the number of vertex reassignments performed by Algorithm 4 with more efficient rebalancing is $O(n \log n + (\Delta \log n)/\varepsilon)$, where Δ is the number of vertex moves used by OPT.*

Proof. First, note that the number of vertex moves for moving smaller components to larger components (Line 8) is $O(n \log n)$, by exactly the same arguments used in the proof of Lemma 6.5.

Second, we bound the number of vertex moves required for the rebalancing operations. Whenever the algorithm needs to rebalance, we can assume (for the sake of the analysis) that the algorithm makes the following three steps: (1) Roll back all changes done by small-to-large moves (Line 8) *since the last rebalancing operation*. Thus, after rolling back we have the same assignment as after the last rebalancing operation. (2) Roll back to the initial assignment (by undoing the last rebalancing operation). (3) Move to a cheap assignment.

Observe that Step (1) and (2) of the previous three step procedure increase the number of vertex moves only by a constant factor compared to when the algorithm does not roll back: In total, Step (1) only adds additional $O(n \log n)$ vertex moves because each small-to-large move is undone exactly once. Step (2) only doubles the number of vertex moves for moving to cheap assignments as each rebalancing is only undone once.

Thus, we can complete the proof if we can show that the total number of vertex moves for moving from the initial assignment to the cheap assignments is bounded by $O((\Delta \log n)/\varepsilon)$.

By Lemma 6.5, the number of rebalancing steps is bounded by $O((\log n)/\varepsilon)$. Now we argue that for moving from the initial solution to each cheap assignment, the rebalancing moves at most $O(\Delta)$ vertices: Every time the algorithm computes a cheap rebalancing, the final solution obtained by OPT is a perfectly balanced assignment respecting the connected components. Thus, the number of vertex moves to obtain a cheap rebalancing is bounded by the number of moves performed by OPT which is $O(\Delta)$. This finishes the proof. □

6.3.3 The Majority Voting Algorithm

We now present an algorithm which works well whenever the cost paid by OPT is small, i.e., when OPT only needs to move few vertices. The issue with Algorithm 4 from Section 6.3.2 is that during its execution, it might deviate much from the initial assignment (and thus move many vertices). The following algorithm has the property that it always stays close to the initial assignment.

For ease of readability, we will often refer to the two servers as the *left* and *right* servers, respectively, instead of calling them S_0 and S_1.

Input: A sequence of edges $\sigma = (e_1, \dots, e_r)$

procedure MAJORITYVOTING($e_1, \dots, e_r$)
    Color all vertices assigned to the left server yellow and all vertices assigned to the right server black
    for $i = 1, \dots, r$
        $(u, v) \leftarrow e_i$
        Suppose w.l.o.g. that $|C_u| \leq |C_v|$
        if C_u and C_v are on different servers
            Move C_u to the server of C_v ▷ Small-to-large step
        Merge C_u and C_v
        if there exists an $i \in \mathbb{N}$ s.t. $|C_u| < 2^i$, $|C_v| < 2^i$ and $|C_u \cup C_v| \geq 2^i$ ▷ Majority voting step
            if $C_u \cup C_v$ has a yellow majority
                Move $C_u \cup C_v$ to the left server
            if $C_u \cup C_v$ has a black majority
                Move $C_u \cup C_v$ to the right server

Algorithm 5: The Majority Voting Algorithm

Our algorithm starts by coloring vertices on the left server yellow and on the right server black. Throughout the execution of the algorithm, the vertices will keep this initially assigned color. The algorithm then follows the idea of always moving the smaller connected component to the server of the larger connected component; we will refer to this as *small-to-large step*. To stay close to the initial assignment, whenever the number of vertices in a newly merged connected component surpasses a power of 2, the algorithm performs a majority vote and moves the component to the server where more of its vertices originate from. More formally, we say that a set of vertices (e.g., a connected component) has a *yellow (black) majority* if it contains more yellow (black) vertices than black (yellow) vertices. In the majority voting step, the algorithm moves a component with a yellow (black) majority which is currently on the right (left) server to the left (right) server. The pseudocode for this procedure is stated in Algorithm 5.[2]

The reason for introducing the majority voting step is that it keeps the assignments produced by the algorithm during its runtime close to the initial assignment. Due to this property, we can show that the cost of Algorithm 5 is always close to the cost of OPT. The formal guarantees are stated in Proposition 6.7.

[2]Note that in Algorithm 5 the following is possible when a component C_u is merged with a component C_v: C_u is moved from S to S_v due to a small-to-large step and immediately after that $C_u \cup C_v$ is moved back to S due to a majority-voting step. Thus, it would be more efficient to compute the result of the majority-voting step earlier and to move C_v to S immediately (without ever moving C_u to S_v). This modification would be slightly more efficient but it would affect the competitive ratio of the algorithm only by at most a constant factor. Thus, to simplify our analysis, we ignore this modification.

Proposition 6.7. *Let Δ be the number of vertex moves performed by OPT (see Section 6.3.1). Then Algorithm 5 is $O(\log n)$-competitive and the load of both servers is bounded by $n/2 + 4\Delta$.*

We devote rest of this subsection to the proof of the proposition. We start bounding the augmentation. For the proofs recall that V_0 and V_1 are the ground truth connected components of G.

In the following we are interested in what happened to a connected component since its last majority vote. To this end, we decompose it into a sequence of smaller connected components such that first a majority vote is performed and after that, only small-to-large steps are performed. For all of these small-to-large steps, the component will stay on the server that was picked by the majority vote. The following definition makes this notion formal.

Definition 6.8 (Doubling Decomposition)**.** *Let C be a connected component and let $s \in \mathbb{N}$ be such that $2^t \leq |C| < 2^{t+1}$. Consider k disjoint sets of vertices $C_i \subseteq V$ and let $\mathbb{C}_j = \bigcup_{i=1}^{j} C_i$ for $j = 1, \dots, k$.*

A sequence $(C_1, \dots, C_k)$ is a doubling decomposition of C if the following properties hold:

1. *$C = \mathbb{C}_k = \bigcup_{i=1}^{k} C_i$,*
2. *during the execution of the algorithm, first $\mathbb{C}_1 \cup C_2$ are merged, then $\mathbb{C}_2 \cup C_3$ are merged, and, more generally, $\mathbb{C}_{i-1} \cup C_i$ is merged before $\mathbb{C}_i \cup C_{i+1}$,*
3. *for each $i = 1, \dots, k-1$, $|\mathbb{C}_i| \geq C_{i+1}$ and the algorithm moves C_{i+1} to the server of $\mathbb{C}_i$,*
4. *$|C_1| < 2^t$ and $|\mathbb{C}_2| = |C_1 \cup C_2| \geq 2^t$.*

Note that when considering a doubling decomposition, there will be exactly one majority-vote for the components $\mathbb{C}_j$ — the one after C_1 and C_2 are merged. Thus, C and all $\mathbb{C}_j$, $j \geq 2$, will be assigned to the server that was picked in the majority vote of $C_1 \cup C_2$.

The following lemma shows that doubling decompositions are well-defined. Its proof provides the construction of a doubling decomposition for a given connected component.

Lemma 6.9. *Let C be a connected component. Then there exists a doubling decomposition $(C_1, \dots, C_k)$ for C.*

Proof. Suppose (u, v) was the last edge which caused the algorithm to set $C = C_u \cup C_v$. W.l.o.g. assume that $|C_u| \leq |C_v|$ (in case of ties let C_u be the connected component that is moved by the algorithm). Then set $C_k = C_u$ and set $\mathbb{C}_{k-1} = C_v$. Now repeat this procedure for $\mathbb{C}_{k-1}$ in place of C to obtain C_{k-1} and $\mathbb{C}_{k-2}$. Continue this procedure until C_1 is of appropriate size.

Note that Properties 1 and 2 follow immediately from the above construction. Property 3 follows from the definition of small-to-large steps and the choice of C_u above. Property 4 is guaranteed by the stopping criterion of the above recursion. □

Lemma 6.10 will be crucial for the proofs of many upcoming claims in this section. The lemma asserts that when a connected component C is currently assigned to the (say) right server but at the end it will be assigned to the left server, then it must contain relatively many vertices that were initially assigned to the right server.

Lemma 6.10. *Let C be a connected component with $|C| \geq 4$. Suppose that C is currently assigned to server S_i and that C will be assigned to server S_{1-i} when the algorithm terminates. Then C contains at least $|C|/4$ vertices which were initially assigned to S_i.*

Proof. Assume w.l.o.g. that C is currently assigned to the right server and it will be assigned to the left server when the algorithm terminates. We show that at least a $1/4$-fraction of the vertices in C must be black. This implies the lemma.

Let $(C_1, \dots, C_k)$ be a doubling decomposition of C which exists by Lemma 6.9. Observe that C must be assigned to the same server as $C_1 \cup C_2$ after they were merged and after the algorithm processed the majority vote for $C_1 \cup C_2$ (by Properties 3 and 4 of doubling decompositions). Thus, $C_1 \cup C_2$ had a black majority, i.e., it contains at least $|C_1 \cup C_2|/2$ black vertices. Since $|C_1 \cup C_2| \geq |C|/2$, C must contain at least $|C|/4$ black vertices. □

Now we bound the augmentation that is used by Algorithm 5.

Lemma 6.11. *The load of both servers is bounded by $n/2 + 4\Delta$. Hence, Algorithm 5 uses at most 4Δ augmentation.*

Proof. Assume that at some point during the execution of the algorithm the (w.l.o.g.) right server contains more vertices than the left server. We bound the load of the right server.

Recall from Lemma 6.3 that $\Delta \leq n/4$. We start by considering the case where $\Delta = n/4$. In this case, even moving all n vertices to the right server only causes augmentation $n/2 = 2\Delta$.

Now consider the case where $\Delta < n/4$. Since $\Delta < n/4$, the initial assignment of S_1 must contain more vertices from either V_0 or V_1. Thus, exactly one of the ground truth components V_0 and V_1 must have a black majority (as the algorithm colored all vertices initially assigned to S_1 black). We assume w.l.o.g. that V_1 has this black majority. This implies that V_1 has $n/2 - \Delta > n/4 > \Delta$ black vertices and V_0 has Δ black vertices. Further, as the algorithm proceeds, the vertices from V_1 must be moved to the right server.

The right server contains at each point a (potentially empty) set of vertices from V_0 and a (potentially empty) set of vertices from V_1. For the latter set we use the trivial upper bound of $n/2$, while for the earlier set we give a bound of $\Delta/4$. The lemma follows.

Consider a component C which is on the right server and a subset of V_0. By Lemma 6.10, C contains at least $|C|/4$ black vertices.

As there are only Δ black vertices in the ground truth component V_0 and each component $C \subseteq V_0$ on the right server has at least a $1/4$-fraction of black vertices,

it follows that all components on the right server which are subsets of V_0 can only contain 4Δ vertices. □

Having derived the bound for the augmentation, our next goal is to show that the cost paid by the algorithm is bounded by $O(\alpha\Delta \log \Delta)$. We start by bounding the cost paid by the algorithm for each connected component.

The following lemma implies that the algorithm pays nothing for components in which all vertices have the same color.

Lemma 6.12. *Let C be a connected component and suppose all vertices in C have the same color. Then the algorithm has never moved the vertices in C.*

Proof. We prove the claim by induction over $s = |C|$.

Let $|C| = s = 1$. Then C consists of a single vertex. But the algorithm never moves single vertices unless they become part of a larger connected component. Hence, C is not moved.

Now let $|C| = s + 1$. Consider the last edge (u, v) which was inserted that forced the algorithm to merge $C = C_u \cup C_v$. Since in C all vertices have the same color, all vertices in C_u and C_v must have the same color. By induction hypothesis, the vertices in C_u and C_v have never been moved before. Thus, C_u and C_v must be assigned to the same server. This implies that a small-to-large step would not move C_u or C_v. Further, a majority voting step would not move $C_u \cup C_v$ since all vertices vote for the server which they are already assigned to. Thus, no vertices in C are moved. □

Next, we bound the cost paid for any connected component.

Lemma 6.13. *Let C be a connected component. Then the cost (over the entire execution time of the algorithm) paid for the vertices in C is at most $O(\alpha|C| \log |C|)$.*

Proof. Consider a vertex $u \in C$. We perform the following accounting: we assign a token to u each time when it is reassigned to a server and we show that the number of tokens for u is bounded by $O(\log |C|)$. This implies that the total number of reassignments for the vertices in C is $O(|C| \log |C|)$ and the lemma follows.

First, consider the case where u is moved because it is in a smaller connected component (Line 7). Whenever this happens the size of the connected component containing u at least doubled. This can only happen $O(\log |C|)$ times.

Second, consider the case when u is moved because of a majority vote. A majority vote is performed every time when the size of the component containing u doubled. This can only happen $O(\log |C|)$ times and, hence, this can only add another $O(\log |C|)$ tokens for u.

Thus, the total number of tokens assigned to u is $O(\log |C|)$. □

Note that Lemma 6.13 is only useful for components of size at most $O(\Delta)$: If we were to apply the lemma to a component C of size $\Theta(n)$ then the cost would only be bounded by $O(\alpha n \log n)$. However, this can be much worse than our desired

bound of $O(\alpha\Delta \log \Delta)$ when $\Delta \ll n$. Thus, we need a more fine-grained argument to obtain our goal of showing that the cost paid by Algorithm 5 never exceeds $O(\alpha\Delta \log \Delta)$. To do this, we first prove two technical lemmas.

Lemma 6.14. *Suppose C is a component which is moved from S_i to S_{1-i} and the vertices in C are never reassigned after this move. Then C contains at least $|C|/8$ vertices which were initially assigned to S_i.*

Proof. There are only two possible reasons why C is moved: Either due to a small-to-large step (Line 7) or due to a majority voting step (Line 9). We consider both cases separately.

Case 1: C is moved due to a small-to-large step. Then by Lemma 6.10, C must contain at least $|C|/4$ vertices which were initially assigned to S_i.

Case 2: C is moved due to a majority voting step.

First, consider the case when C contains at most 7 vertices. Then at least one vertex was initially assigned to S_i (if all vertices had been initially assigned to S_{1-i}, they would all have the same color and a majority vote would not move C due to Lemma 6.12). Thus, at least a $1/7$-fraction of the vertices were initially assigned to S_i and the lemma holds.

Second, suppose that C contains at least 8 vertices. Consider the last edge (u, v) that caused the merge $C = C_u \cup C_v$. Suppose that the small-to-large step moved C_u to the server of C_v. Note that C_v was assigned to S_i and C_u was moved to S_i. Now apply Lemma 6.10 to C_v. This implies that C_v must contain at least $|C_v|/4$ vertices that were initially assigned to S_i. As $|C_u| \leq |C_v|$, C must contain at least $|C|/8$ vertices that were initially assigned to S_i. □

We are now ready to show that the cost incurred by the majority-voting algorithm never exceeds $O(\alpha\Delta \log \Delta)$.

Lemma 6.15. *The total cost paid by Algorithm 5 is at most $O(\alpha\Delta \log \Delta)$ and the final assignment is a perfect partitioning.*

Proof. When the algorithm finishes, the final assignment must be a perfect partitioning because the connected components were completely revealed. We only need to prove that the cost of the algorithm is $O(\alpha\Delta \log \Delta)$.

Recall that OPT moves exactly 2Δ vertices (Lemma 6.3). We can assume w.l.o.g. that OPT moves Δ vertices from V_0 that were initially assigned to S_1 to S_0 and Δ vertices from V_1 that were initially assigned to S_0 to S_1. We will argue that the cost paid by the algorithm for moving all vertices from V_0 into the S_0 will be $O(\alpha\Delta \log \Delta)$; the same will hold for V_1 and S_1 symmetrically.

Consider time T during the execution of the algorithm where the following happens. A connected component C is reassigned the left server and C has the following properties: (1) C is a subset of V_0 and (2) the vertices in C never leave the left server after time T. Since each vertex of V_0 is assigned to the left server when the algorithm terminates, each vertex of V_0 is contained in a component with

the above properties (when a vertex or component is never moved, we set $T = 0$). We call a component with the above properties *mixed* if it contains at least one black vertex. Note that when mixed component C is assigned to the left server, C contains a black vertex and, hence, C must be moved from the right to the left server.

We now bound the cost for mixed components. Let X be the set of all mixed components and let $C \in X$. Since C is mixed, Lemma 6.14 implies that at least $|C|/8$ vertices of C are black. As the black vertices in mixed components form a partition of the Δ black vertices in V_0 moved by OPT, we obtain that the number of black vertices in mixed components is Δ. Thus, the total number of vertices in all mixed components is $\sum_{C \in X} |C| \leq 8\Delta$.

By Lemma 6.13, the total cost paid for each $C \in X$ until (including) its final move is $O(\alpha |C| \log |C|)$. Since (by assumption) the vertices in C never move between the servers again, their cost never exceeds $O(\alpha |C| \log |C|)$ until the algorithm finishes. Hence, the cost paid by the algorithm for all mixed components is

$$\sum_{C \in X} O(\alpha |C| \log |C|) \leq \sum_{C \in X} O(\alpha |C| \log \Delta) = O(\alpha \Delta \log \Delta).$$

Now consider the vertices of V_0 which are not part of mixed components. These vertices must have been part of components in which all vertices are colored yellow. By Lemma 6.12, these vertices have never been moved. Thus, they do not incur any additional cost for the algorithm. □

Proof of Proposition 6.7. Lemma 6.11 gives the bound for the augmentation used by the algorithm. By Lemma 6.15 and Lemma 6.3, Algorithm 5 obtains a competitive ratio of

$$\frac{\mathrm{ONL}}{\mathrm{OPT}} = \frac{O(\alpha \Delta \log \Delta)}{2\alpha\Delta} = O(\log \Delta) = O(\log n). \qquad \square$$

6.3.4 Bringing It All Together: Theorem 6.2

Proof. Proof of Theorem 6.2. Consider the following algorithm: Run the majority-voting algorithm until we have seen all edges or until at some point it tries to exceed the allowed augmentation. In the latter case, compute a perfectly balanced assignment respecting the connected components and start running Algorithm 4 (Section 6.3.2.2).

To prove the theorem, we distinguish two cases based on Δ.

First, suppose $\Delta < \varepsilon n/4$. By Proposition 6.7, Algorithm 5 uses at most 4Δ augmentation. Thus, in the current case the augmentation used by Algorithm 5 is bounded by $4\Delta < \varepsilon n$ and it is $O(\log n)$-competitive. This proves the theorem for this case.

Second, suppose $\Delta \geq \varepsilon n/4$. In this case we run Algorithm 5 until it tries to exceed the allowed augmentation; this serves as a certificate that $\Delta \geq \varepsilon n/4$. At this point we switch to Algorithm 4.

When we switch algorithms, Algorithm 5 has paid $O(\alpha n \log n)$, by applying Lemma 6.13 to each connected component, and then summing over these costs. For switching to the perfectly balanced reassignment, we only need to pay $O(\alpha n)$ *once*.

By Proposition 6.6, Algorithm 4 never uses more than $O(n \log n + (\Delta \log n)/\varepsilon)$ vertex moves. Using the bound $\Delta \geq \varepsilon n/4$ and the fact that OPT pays $2\alpha\Delta$ (Lemma 6.3), we obtain the desired competitive ratio:

$$\frac{\mathrm{ONL}}{\mathrm{OPT}} = O\left(\frac{\alpha n \log n + (\alpha\Delta \log n)/\varepsilon}{\alpha\Delta}\right) = O\left(\frac{\log n}{\varepsilon}\right). \qquad \square$$

6.4 Generalization to Many Servers

We extend our study to the scenario with ℓ servers. As we will see, while several concepts introduced for the two server case are still useful, the ℓ-server case introduces additional challenges. We derive the following main result.

Theorem 6.16. *Given a system with ℓ servers each of capacity $(1+\varepsilon)n/\ell$ (i.e., augmentation $\varepsilon n/\ell$), for $\varepsilon \in (0, 1/2)$, then there exists an $O((\ell \log n \log \ell)/\varepsilon)$-competitive algorithm.*

Our algorithm will be based on a recursive bipartitioning scheme, described in Section 6.4.1. We will use this bipartitioning scheme to derive a static approximation algorithm of the optimal solution (Section 6.4.2). Then we provide a recursive version of the majority voting algorithm which we will compare against the approximation algorithm (Section 6.4.3). In Section 6.4.4, we analyze the Small–Large–Rebalance algorithm in the ℓ server setting and we conclude by proving Theorem 6.16 in Section 6.4.5

6.4.1 The Bipartition Tree

We establish a recursive bipartitioning scheme of the servers which we will be using throughout the rest of this section. All algorithms in this section which use the recursive bipartitioning create such a bipartitioning at the start of the algorithm, before the adversary provides any edge. After that the bipartitioning will never be changed.

We obtain the bipartition scheme by growing a balanced binary tree on a set of ℓ leaves, where each leaf corresponds to a server S_i. We call this tree the *bipartition tree* and denote it by $\mathcal{T}$.

We denote the internal nodes of $\mathcal{T}$ by $w_1, \ldots, w_s$. For an internal node w_j, we write $T(w_j)$ to denote the subtree of T which is rooted at w_j and we define $S(w_j)$ to be the set of servers which are leaves in $T(w_j)$. We further write $V(w_i)$ to denote the set of vertices which are assigned to the servers in $S(w_j)$. See Figure 6.3 for an illustration.

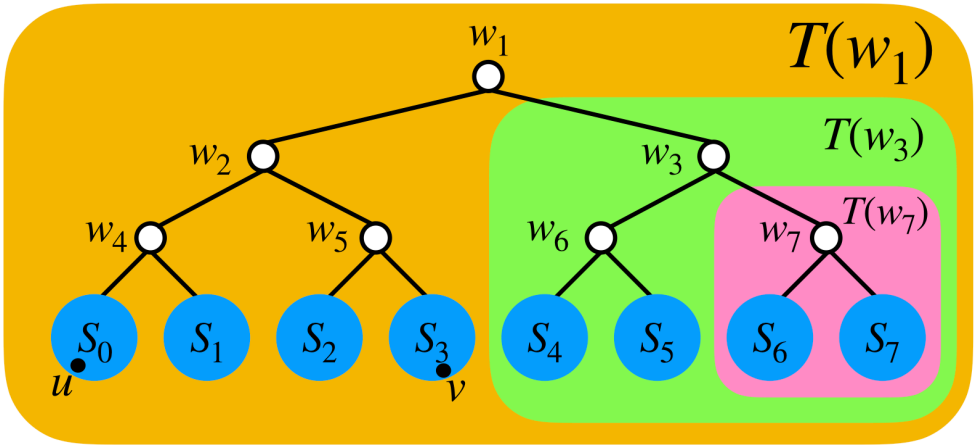


Figure 6.3: An illustration of the bipartition tree $\mathcal{T}$ for servers $S_0, \dots, S_7$. The internal nodes of the bipartition tree are denoted $w_1, \dots, w_7$. We highlighted the subtrees $\mathcal{T} = T(w_1)$, $T(w_3)$, and $T(w_7)$. Here we obtain the server sets $S(w_1) = \{S_0, \dots, S_7\}$, $S(w_3) = \{S_4, \dots, S_7\}$, and $S(w_7) = \{S_6, S_7\}$.

Observe that $\mathcal{T}$ defines a bipartition scheme: let w be an internal node of $\mathcal{T}$ and let w_0, w_1 be its children. Then[3] $S(w_0)$ and $S(w_1)$ are disjoint and their union is $S(w)$. Thus, $\mathcal{T}$ implies a bipartition scheme of the servers and internal nodes correspond to bipartition steps.

Note that since $\mathcal{T}$ is a balanced binary tree, there are $\ell - 1$ internal nodes in total and each server is contained in at most $\lceil \log \ell \rceil$ subtrees of T. Hence, for each server S_j there are at most $\lceil \log \ell \rceil$ internal vertices w such that $S_j \in S(w)$.

In the following we will refer to the internal nodes in $\mathcal{T}$ as *nodes*, whereas the vertices V from the graph G are called *vertices*.

6.4.2 Offline Approximation Algorithm

We are not aware of a concise characterization of the optimal solution used by OPT (unlike in the two-server case in Section 6.3.1). Thus, to get a better understanding of the solution obtained by OPT, we provide an *offline* approximation algorithm, called APPROX, which exploits the previously defined bipartition scheme and which obtains a 2-approximation of the optimal solution. However, unlike the solution obtained by OPT, we allow the approximation algorithm to use *unlimited* augmentation in each server; its only goal is to move all vertices from the same ground truth components to the same server using few vertex moves.[4] Later, APPROX will play a role for the design and analysis of our online algorithm.

Intuitively, APPROX traverses the bipartition tree $\mathcal{T}$ top–down and greedily minimizes the number of vertices "moved over" each server bipartition. We now

[3] If w_j is a leaf corresponding to server S, we set $S(w_j) = \{S\}$.

[4] In this setting, a trivial solution assigns all vertices to the same server at cost $O(\alpha n)$.

Input: All edges $e_1, \dots, e_r$ at once
1: **procedure** APPROX($e_1, \dots, e_r$)
2: Compute the connected components $V_0, \dots, V_{\ell-1}$ of G
3: **for** $i = 0, \dots, \ell - 1$
4: RecursiveStep($\mathcal{T}$, i)
5: **procedure** RecursiveStep(T, i)
6: **if** T contains only a single server S
7: Move all dirty vertices of V_i into S
8: **return**
9: Let $\mathfrak{r}$ be the root of T and denote its children w_0, w_1
10: $A_{ij} \leftarrow V(w_j) \cap V_i$, $j = 0, 1$
11: $n_{ij} \leftarrow |A_{ij}|$ and suppose w.l.o.g. that $n_{i0} \geq n_{i1}$
12: Mark all vertices in A_{i1} dirty
13: RecursiveStep($T(w_0)$, i)

Algorithm 6: The static approximation algorithm APPROX

describe the algorithm in more detail.

APPROX is given the sequence of edges $\sigma = (e_1, \dots, e_r)$ *a priori* and it also knows the initial assignment $V_{\text{init}}(S_0), \dots, V_{\text{init}}(S_{\ell-1})$ of the vertices to the ℓ servers. Using the knowledge about the edges in σ, APPROX starts by computing the connected components of G and thus obtains the ground truth components $V_0, \dots, V_{\ell-1}$.

Now, for each ground truth component V_i, APPROX does the following. Let $\mathfrak{r}$ be the root of $\mathcal{T}$ and let w_0 and w_1 denote its children. Let $A_{ij} = V(w_j) \cap V_i$, $j = 0, 1$, denote the vertices from V_i which are currently assigned to servers in $S(w_j)$. Define $n_{ij} = |A_{ij}|$ and assume w.l.o.g. that $n_{i0} \geq n_{i1}$. The algorithm marks the vertices from A_{i1} as *dirty*. Now the algorithm recurses on the subtree $T(w_0)$ in place of $\mathcal{T}$ and marks more vertices of V_i as dirty. The recursion stops when $S(w_0)$ only contains a single server S. Then the algorithm moves all dirty vertices of V_i into server S.

The pseudocode of APPROX is stated in Algorithm 6.

By overloading notation, we let APPROX denote the cost paid by APPROX. Further, we let APPROX_i denote the cost paid by APPROX to move all vertices from V_i to the same server S.

We now show that APPROX indeed yields a 2-approximate solution of the cost of the optimal offline algorithm.

Lemma 6.17. $\text{APPROX} \leq 2 \cdot \text{OPT}$.

Proof. Fix any $i \in \{0, \dots, \ell - 1\}$. Let OPT_i denote the cost paid by OPT to move the vertices from V_i to the same server. We show that $\text{APPROX}_i \leq 2 \cdot \text{OPT}_i$.

This claim implies the lemma since

$$\mathrm{APPROX} = \sum_i \mathrm{APPROX}_i \leq 2 \sum_i \mathrm{OPT}_i = 2 \cdot \mathrm{OPT}\,.$$

Observe that while APPROX_i proceeds, it traverses $\mathcal{T}$ from root $\mathfrak{r}$ to one of the leaves, and at each step, it increases the level of the current internal node by one.

Using the solution of OPT_i, we can define a similar traversal of $\mathcal{T}$: Let $\mathfrak{r}$ be the root of $\mathcal{T}$ and let w_0 and w_1 be its children. As OPT_i must move all vertices from V_i to the same server S, OPT_i moves the vertices from $A_{ij} = V(w_j) \cap V_i$ to a server S in $S(w_{1-j})$ for $j \in \{0, 1\}$. We call the moved vertices *dirty*. After this move, OPT_i still needs to process the vertices of V_i which were initially assigned to a server in $S(w_{1-j})$ but not to S. We can view this as processing $T(w_{j-1})$. Thus, OPT_i traverses $\mathcal{T}$ until the final server S is reached and marks a subset of V_i dirty.

The previous paragraphs define to two different traversals of $\mathcal{T}$ and two different sets of dirty vertices. Let h be the smallest level where the two traversals picked different internal nodes in $\mathcal{T}$.

Until level $h-1$, both vertices have marked the same W vertices dirty. At levels h and below, we obtain the following bounds. Let w be the internal node at level $h-1$ that is traversed by both algorithms and let w_0, w_1 denote its children at level h. Let $n_{ij} = |V(w_j) \cap V_i|$ be defined as in the definition of APPROX. APPROX_i marks at most $|V(w)| = n_{i0} + n_{i1}$ vertices from $V(w)$ as dirty. Since the two traversals split at level h and APPROX_i moves $n_{i1} \leq n_{i0}$ vertices (by definition), OPT_i moves at least n_{i0} vertices.

Recall that for each algorithm, its sets of dirty vertices and its set of moved vertices are identical. The following calculation proves the claim that $\mathrm{APPROX}_i \leq 2 \cdot \mathrm{OPT}_i$:

$$\frac{\mathrm{APPROX}_i}{\mathrm{OPT}_i} \leq \frac{\alpha(W + n_{i0} + n_{i1})}{\alpha(W + n_{i0})} \leq \frac{W + 2n_{i0}}{W + n_{i0}} \leq 2. \qquad \square$$

6.4.3 The Recursive Majority Voting Algorithm

We now describe an algorithm which works efficiently in the setting with ℓ servers whenever OPT does not perform too many vertex moves. The algorithm can be viewed as a generalization of Algorithm 5 to ℓ servers, by exploiting the previously defined bipartitioning scheme.

6.4.3.1 The Algorithm

The algorithm consists of two parts: A single *global algorithm* and multiple *local algorithms*, one per internal node in $\mathcal{T}$. The global algorithm maintains a recursive bipartitioning scheme (as defined in Section 6.4.1) and runs a local algorithm on each bipartition. The local algorithms are used to "reduce" the setting with multiple servers to the case with two servers.

We now describe the two parts in more detail and state the pseudocode in Algorithm 7. We write S_u to denote the server which vertex u is assigned to.

Global Algorithm. The global algorithm starts by computing the bipartition tree $\mathcal{T}$. On each internal node w of $\mathcal{T}$, the global algorithm instantiates a local algorithm which we describe below.

Furthermore, the global algorithm iterates over all vertices and does the following for each $v \in V$. The algorithm finds all internal nodes w_i such that $v \in V(w_i)$ and labels v with w_i. This labelling of the vertices only takes into account the initial assignment of the vertices and will never be changed throughout the running time of the algorithm. For example, if the vertices u and v in Figure 6.3 are assigned to servers S_0 and S_3 in the initial assignment, their labels will be $\{w_1, w_2, w_4\}$ and $\{w_1, w_2, w_5\}$, respectively.

When the adversary provides an edge (u, v), the global algorithm does the following. It locates the servers S_u and S_v. If $S_u = S_v$, the algorithm merges the components and continues with the next edge. If $S_u \neq S_v$, the global algorithm finds the internal node w in $\mathcal{T}$ which is the lowest common ancestor of S_u and S_v. (For example, in Figure 6.3 the lowest common ancestor for u and v is w_2.) Then the global algorithm gives the edge (u, v) to the local algorithm corresponding to w.

Local Algorithms. A local algorithm is run on an internal node w of $\mathcal{T}$. Let w_0 and w_1 denote the children of w in $\mathcal{T}$. Note that each local algorithm corresponds to a bipartition step where the servers in $S(w)$ are partitioned into subsets $S(w_0)$ and $S(w_1)$.

An instance of the local algorithm only receives edges (u, v) from the global algorithm when (1) their endpoints are assigned to servers $S_u, S_v \in S(w)$ and (2) S_u and S_v are in different sets of the bipartition, i.e., $S_u \in S(w_j)$ and $S_v \in S(w_{1-j})$.

When the global algorithm provides an edge (u, v) with the above properties, the local algorithm locates C_u, C_v, S_u and S_v. Assume w.l.o.g. that $|C_u| \leq |C_v|$. Then C_u is moved to S_v and C_u and C_v are merged.[5] As before, we call this a *small-to-large step.*

Finally, the local algorithm checks whether the new component $C_u \cup C_v$ has size n/ℓ or it surpassed a power of 2, i.e., it checks if $|C_u \cup C_v| = n/\ell$ or there exists an $i \in \mathbb{N}$ s.t. $|C_u| < 2^i$, $|C_v| < 2^i$ and $|C_u \cup C_v| \geq 2^i$. If this is the case, the local algorithm triggers a majority voting step for $C_u \cup C_v$ which we explain next.

Majority Voting Step. When a local algorithm triggers a majority voting step for a connected component C, the algorithm does the following. Let $\mathfrak{r}$ be the root

[5]When C_u changes its server, all local algorithms corresponding to internal nodes w with $S_u \in S(w)$ or $S_v \in S(w)$, must be informed about this move. This can be done by recomputing $V(w)$ for each internal node w. Note that this is just an internal operation of the data structure and does not incur any cost to the algorithm.

Input: A sequence of edges $\sigma = (e_1, \dots, e_r)$

1: **procedure** GlobalAlgorithm($e_1, \dots, e_r$)
2: Create a bipartition tree $\mathcal{T}$ ▷ Initialization phase
3: **for** each internal node w of $\mathcal{T}$
4: Instantiate LocalAlgorithm(w)
5: **for** $v \in V$
6: Label v with each internal node w of $\mathcal{T}$ s.t. $v \in V(w)$
7: **for** $i = 1, \dots, r$ ▷ Processing of the edges
8: $(u, v) \leftarrow e_i$
9: **if** $S_u = S_v$
10: Merge C_u and C_v, **continue**
11: $w \leftarrow$ the lowest common ancestor of S_u and S_v in $\mathcal{T}$
12: LocalAlgorithm(w, (u, v))
13: **procedure** LocalAlgorithm(w, (u, v))
14: $w_0, w_1 \leftarrow$ the children of w in $\mathcal{T}$
15: Suppose w.l.o.g. that $|C_u| \leq |C_v|$
16: Check if moving C_u to S_v triggers the stopping criterion
17: Move C_u to S_v and merge C_u and C_v ▷ Small-to-large step
18: **if** $|C_u \cup C_v| = n/\ell$ **or** there exists an $i \in \mathbb{N}$ s.t. $|C_u| < 2^i$, $|C_v| < 2^i$ and $|C_u \cup C_v| \geq 2^i$
19: MajorityVotingStep($C_u \cup C_v$)
20: **procedure** MajorityVotingStep(C)
21: $\mathfrak{r} \leftarrow$ the root of $\mathcal{T}$
22: $w_0, w_1 \leftarrow$ the children of $\mathfrak{r}$ in $\mathcal{T}$
23: $n_j \leftarrow$ the number of vertices labeled with w_j in C, $j = 0, 1$
24: **if** $n_0 \geq n_1$ $\mathfrak{r} \leftarrow w_0$ **else** $\mathfrak{r} \leftarrow w_1$
25: **if** $S(r)$ contains only one server
26: Check if moving C to $S(r)$ triggers the stopping criterion
27: Assign C the single server in $S(r)$
28: **else** Go to Line 22

Algorithm 7: The Recursive Majority Voting Algorithm

of $\mathcal{T}$ and let w_0 and w_1 be the two child nodes of $\mathfrak{r}$. For $j \in \{0, 1\}$, let n_j denote the number of vertices in C with label w_j. If $n_j \geq n_{1-j}$, the algorithm recurses on w_j in place of w; else, the algorithm recurses on w_{1-j} in place of w. The recursion continues until a leaf in the bipartitioning tree is reached which corresponds to a server S. Then the algorithm moves C to S.

Note that the above majority voting procedure is very similar to what APPROX does for a single ground truth component V_i.

Stopping Criterion. To ensure that the algorithm does not exceed the augmentation of the servers, we add a stopping criterion.

To define the stopping criterion, let w be an internal node of $\mathcal{T}$ with children w_0 and w_1. For $j \in \{0,1\}$, we call w_j *overloaded* if $V(w_j)$ contains at least $\varepsilon n/(\ell\lceil\log\ell\rceil)$ vertices with label w_{1-j}.

Intuitively, the condition states that an internal node w_j is overloaded when its servers $S(w_j)$ obtained "many" vertices which were initially assigned to the other side of the bipartition, $S(w_{1-j})$.

The *stopping criterion* is checked before each component move (i.e., before each small-to-large step and before each majority voting step). It is triggered if the component move would create an assignment in which there exists an overloaded internal node w. When the stopping criterion is triggered, the global algorithm and all local algorithms stop and Algorithm 4 is started instead (we show in Section 6.4.4 that Algorithm 4 also works for ℓ servers).

6.4.3.2 Structural Properties

To obtain a better understanding of the algorithm, we first prove some structural properties about it and defer its cost analysis to Section 6.4.3.3. We consider the setting where each server has capacity $(1+\varepsilon)n/\ell$ for $\varepsilon \in (0, 1/2)$.

In Subsections 6.4.3.2 and 6.4.3.3, we only analyze the cost Algorithm 7 without the cost of Algorithm 4. We analyze the cost of Algorithm 4 for ℓ servers in Sections 6.4.4 and 6.4.5.

We begin by showing that as long as the stopping criterion is not triggered, the vertex assignment created by Algorithm 7 is close to the initial assignment.

Lemma 6.18. *Suppose the stopping criterion is not triggered. Then:*

1. *Each server contains at most $\varepsilon n/\ell$ vertices that were not initially assigned to it.*
2. *Each server contains at least $(1-\varepsilon)n/\ell$ vertices that were initially assigned to it.*

Proof. Consider any server S_j. We show that since the stopping criterion is not triggered, $V(S_j)$ obtains at most $\varepsilon n/(\ell\lceil\log\ell\rceil)$ vertices for each of the $\lceil\log\ell\rceil$ subtrees in $\mathcal{T}$ containing S_j.

As argued in Section 6.4.1, there are at most $\lceil\log\ell\rceil$ internal nodes w of $\mathcal{T}$ such that $S_j \in S(w)$. Since the stopping criterion is not triggered, no internal node of $\mathcal{T}$ is overloaded.

To prove Part (1), consider an internal node w of $\mathcal{T}$ with $S_j \in S(w)$. Let w_0, w_1 be the children of w and suppose $S_j \in S(w_r)$. Observe that $V(S_j)$ can obtain at most $\varepsilon n/(\ell\lceil\log\ell\rceil)$ vertices that were originally assigned to servers in $S(w_{1-r})$ (if it had received more vertices, then w_{1-r} would be overloaded). As there are at most $\lceil\log\ell\rceil$ nodes w with the above property, the number of vertices which were not initially assigned to S_j is bounded by $\varepsilon n/\ell$.

Now let us prove Part (2). Consider an internal node w of $\mathcal{T}$ with $S_j \in S(w)$. Let w_0, w_1 be the children of w and suppose $S_j \in S(w_r)$. Now observe that the

servers in $S(w_{1-r})$ can have obtained $\varepsilon n/(\ell\lceil\log\ell\rceil)$ vertices that were originally assigned to S_j (if they had received more vertices, then w_{1-r} would be overloaded). As there are at most $\lceil\log\ell\rceil$ nodes w with the above property, it follows that the number of vertices assigned to servers $\{S_0,\dots,S_{\ell-1}\}\setminus\{S_j\}$ that were initially assigned to S_j is $\varepsilon n/\ell$. Hence, S_j must contain at least $(1-\varepsilon)n/\ell$ vertices that were initially assigned to it. □

As a corollary of Lemma 6.18 we obtain the following lemma.

Lemma 6.19. *(1) As long as the stopping criterion is not triggered, the load of each server is bounded by $(1+\varepsilon)n/\ell$, i.e., Algorithm 7 uses only $\varepsilon n/\ell$ augmentation.*

(2) When the stopping criterion is triggered, the augmentation still does not exceed $\varepsilon n/\ell$.

Proof. Part (1) of the lemma follows immediately from Part (1) of Lemma 6.18. Let us prove Part (2): The stopping criterion is checked every time before a component is moved. Hence, at the time when the algorithm checks the stopping criterion, the algorithm did not exceed the augmentation bound due to Part (1). If the algorithm triggers the stopping criterion, then the component was not yet moved and the augmentation is still the same as before. □

Define the *final assignment* to be the assignment which is created by Algorithm 7 once it has seen all edges in G. We show that the final assignment of the algorithm provides a perfect partitioning if the stopping criterion is not triggered.

Lemma 6.20. *If Algorithm 7 stops and the stopping criterion is not triggered, then the final assignment is a perfect partitioning.*

Proof. By definition of the algorithm, vertices of the same connected component are always assigned to the same server. When the algorithm finishes, all edges of G were revealed and each component has size n/ℓ. By Lemma 6.18, the augmentation of each server is at most $\varepsilon n/\ell$. Since $\varepsilon < 1/2$, no server can have more than one component assigned. As each component is placed on a server, each component is placed alone on a server. This proves that the algorithm creates a perfect partitioning. □

Indeed, we show that the final assignment of Algorithm 7 is not only a perfect partitioning, but it is the same assignment as the one created by APPROX from Section 6.4.2.

Lemma 6.21. *If Algorithm 7 stops and the stopping criterion is not triggered, Algorithm 7 and APPROX have the same final assignment.*

Proof. By Part (2) of Lemma 6.18, Algorithm 7 moves at most $\varepsilon n/\ell$ vertices out of each server compared to the initial assignment. Hence, in the final assignment each server must still contain at least $(1-\varepsilon)n/\ell > n/(2\ell)$ vertices from its original

assignment since $\varepsilon \in (0, 1/2)$. Thus, in the final assignment each server contains more than half of the vertices that were originally assigned to it.

Consider any server S_j and let $V_{\text{init}}(S_j)$ be the set of vertices initially assigned to S_j. Then there must exist a ground truth component V_i with $|V_i \cap V_{\text{init}}(S_j)| \geq n/(2\ell)$. We show that APPROX and Algorithm 7 both assign this component V_i to S_j. This proves the lemma since this claim holds for any S_j.

First, consider APPROX. Note that at each step of the traversal of $\mathcal{T}$, the majority of the vertices in V_i will vote for the internal node containing server S_j. Hence, APPROX will place V_i on S_j.

Second, consider Algorithm 7. When the algorithm stops, all edges were revealed and the connected components agree with the ground truth components. Now consider the component $C = V_i$. When the C grows to size $|C| = n/\ell$, the algorithm performs a majority voting step (by definition of the algorithm). At this point, more than half of the vertices in C were labeled with S_j (because more than half of the vertices from $C = V_i$ were originally assigned to S_j). Hence, Algorithm 7 will also place V_i on S_j. □

6.4.3.3 Analysis

The rest of this subsection is devoted to proving the following proposition about Algorithm 7.

Proposition 6.22. *Suppose there are ℓ servers and each has capacity $(1+\varepsilon)n/\ell$ for $\varepsilon \in (0, 1/2)$, i.e., the augmentation is $\varepsilon n/\ell$. Algorithm 7 has the following properties:*

1. *If the stopping criterion is not triggered, the algorithm creates a perfect partitioning, its cost is bounded by $O(\mathrm{OPT} \cdot \log n)$ and at no point during its execution it uses more than $\varepsilon n/\ell$ augmentation.*
2. *If the stopping criterion is triggered, the cost of the algorithm is $O(\alpha n \log n)$ plus the cost of Algorithm 4 and the cost of OPT is at least $\Omega(\alpha\varepsilon n/(\ell \log \ell))$.*

We prove the proposition at the end of this section. We start by proving a sequence of lemmata and begin by reasoning about the cost paid by Algorithm 7. As shown in Lemma 6.1 we only need to bound the moving cost paid by Algorithm 7 to bound its total cost.

The following lemma bounds the cost paid for any connected component C.

Lemma 6.23. *Let C be a connected component. Then the cost (over the entire execution time of the algorithm) paid for moving the vertices in C is $O(\alpha|C| \log |C|)$.*

Proof. We can use the same accounting argument as in the proof of Lemma 6.13. That is, we assign a token to a vertex v whenever it is moved. Now, whenever the component C containing v is moved due to a small-to-large step, the size of C doubles. This can only happen $O(\log |C|)$ times. Furthermore, there are only $O(\log |C|)$ majority voting steps involving u: Each majority voting step is triggered because $|C| = n/\ell$ or because $|C|$ surpassed a power of 2; the first event can happen

only once and the second event can happen at most $O(\log |C|)$ times. Hence, v will never accumulate more than $O(\log |C|)$ tokens. Since the above arguments apply for each $v \in C$, the total cost paid for moving the vertices in C is bounded by $O(|C| \log |C|)$. □

Let $f : V \to \{0, \dots, \ell - 1\}$ be the function which maps each vertex to its server in the final assignment by Algorithm 7. That is, when Algorithm 7 processed all edges, each v is assigned to $S_{f(v)}$. For a connected component C, set $f(C) = f(u)$ for $u \in V$. Note that $f(C)$ is well-defined since all vertices of C are assigned to the same $S_{f(C)}$ when the algorithm terminates.

In the following proofs, we will write $\#w(C)$ to denote the number of vertices in a connected component C which are labeled with w. We further write $\overline{\#w(C)}$ to denote the number of vertices in C which are *not* labeled with w, i.e., $\overline{\#w(C)} = |C| - \#w(C)$.

Lemma 6.24 shows that whenever a component C is assigned to a server which is not its final server, it must contain relatively many vertices which were not initially assigned to its final server $S_{f(C)}$.

Lemma 6.24. *Consider any point in the execution of the algorithm at which a connected component C is assigned to server $S \neq S_{f(C)}$. Let w be the lowest common ancestor of S and $S_{f(C)}$ in $\mathcal{T}$ and denote the children of w by w_0 and w_1.*

If $S_{f(C)} \in S(w_j)$ for $j \in \{0, 1\}$, then:

1. *C contains at least $|C|/4$ vertices which do not have label w_j, i.e., $\overline{\#w_j(C)} \geq |C|/4$.*
2. *C contains at least $\overline{\#w_j(C)}$ vertices which were not initially assigned to $S_{f(C)}$.*

Proof. To prove Part (1), consider a doubling decomposition $(C_1, \dots, C_k)$ of C (see Definition 6.8); the decomposition exists by Lemma 6.9 which also applies in the ℓ server setting. After C_1 and C_2 were merged, Algorithm 7 performed a majority voting step and placed $C_1 \cup C_2$ in a server $S \in S(w_{1-j})$. Thus, $\#w_j(C_1 \cup C_2) \leq |C_1 \cup C_2|/2$ (otherwise, the majority voting step would have chosen a server in $S(w_j)$). Since $|C_1 \cup C_2| \geq |C|/2$ and $C_1 \cup C_2 \subseteq C$,

$$\begin{aligned}\overline{\#w_j(C)} &\geq \overline{\#w_j(C_1 \cup C_2)} \\ &= |C_1 \cup C_2| - \#w_j(C_1 \cup C_2) \\ &\geq |C_1 \cup C_2| - |C_1 \cup C_2|/2 \\ &= |C_1 \cup C_2|/2 \geq |C|/4.\end{aligned}$$

For Part (2) note that each vertex which was initially assigned to $S_{f(C)}$ has label w_j (because $S_{f(C)} \in S(w_j)$ by assumption). □

In the following, we show that the cost paid by the algorithm is $O(\mathrm{OPT} \cdot \log n)$ when the stopping criterion is not triggered. We start by showing that when a component is moved for the last time, it contains a large number of vertices which did not originate from the server it is assigned to.

Lemma 6.25. *Let C be a component which is moved to server $S_{f(C)}$ and suppose the vertices of C are never reassigned after this move.*[6] *Then C contains at least $|C|/8$ vertices which were not assigned to $S_{f(C)}$ in the initial assignment.*

Proof. Note that C is moved due to one of two reasons: Either because of a small-to-large step or because of a majority voting step. We distinguish between these cases.

In case of a small-to-large step, C is assigned to a server $S \neq S_{f(C)}$ before the move. Lemma 6.24 implies that C contains at least $|C|/4$ vertices which were not originally assigned to $S_{f(C)}$.

Now suppose that C is moved due to a majority voting step. Let (u, v) be the last edge which was inserted and which triggered the majority voting step for C. Then Algorithm 7 previously merged components C_u and C_v; suppose w.l.o.g. that C_u was moved to C_v and $|C_u| \leq |C_v|$. Prior to the majority voting step, C is assigned to the same server $S \neq S_{f(C)}$ that C_v was assigned to before (u, v) was inserted. Hence, we can apply Lemma 6.24 to C_v and obtain that C_v contains at least $|C_v|/4$ vertices which were not initially assigned to $S_{f(C)}$. Thus, the number of vertices in C which do not originate from $S_{f(C)}$ is at least

$$|C_v|/4 \geq 2|C_v|/8 \geq |C_u \cup C_v|/8 = |C|/8. \qquad \square$$

The next lemma considers the cost paid by Algorithm 7 when the stopping criterion is not triggered.

Lemma 6.26. *Suppose there are ℓ servers and each has capacity $(1+\varepsilon)n/\ell$ for $\varepsilon \in (0, 1/2)$, i.e., the augmentation is $\varepsilon n/\ell$. If the stopping criterion is not triggered and Algorithm 7 stops, then the cost paid by the algorithm is $O(\mathrm{OPT} \cdot \log n)$.*

Proof. Fix some $i \in \{0, \dots, \ell - 1\}$. Recall that APPROX_i denotes the cost paid by APPROX to move the vertices from V_i to the server $S_{f(V_i)}$. We show that for V_i, Algorithm 7 pays $O(\mathrm{APPROX}_i \log n)$. The lemma follows from this claim and Lemma 6.17, since the total cost paid by Algorithm 7 is bounded by

$$\sum_i O(\mathrm{APPROX}_i \cdot \log n) = O(\mathrm{APPROX} \cdot \log n) = O(\mathrm{OPT} \cdot \log n).$$

Consider any ground truth component V_i and let Δ denote the number of vertices APPROX_i moves to server $S_{f(C)}$. Note that as APPROX_i moves Δ vertices into $S_{f(C)}$, we get $\mathrm{APPROX}_i = \alpha\Delta$.

Consider time T of the execution of the algorithm where the following happens. A component C is reassigned to $S_{f(C)}$ and C has the following properties: (1) C is a subset of V_i and (2) the vertices in C never leave server $S_{f(C)}$ after time T. Since each vertex of V_i is assigned to $S_{f(C)}$ when the algorithm terminates, each

[6] Note that when a small-to-large step is performed, two components are merged due to the corresponding edge insertion. In this case, the component C in the lemma is the component which is being moved (i.e., before merging).

vertex of V_i is contained in a component with the above properties (when a vertex or component is never moved, we set $T = 0$). A component C with the above properties is a *mixed* component if C contains at least one vertex which was not initially assigned to $S_{f(C)}$. Note that when a mixed component C is reassigned to $S_{f(C)}$, C contains at least one vertex which was not initially assigned to $S_{f(C)}$ and, hence, C must be moved from a server S_y, $y \neq f(C)$, to $S_{f(C)}$.

We bound the cost for mixed components. Let X be the set of all mixed components of V_i. Recall that Algorithm 7 and APPROX create the same final assignment (Lemma 6.21). Hence, Algorithm 7 moves the same Δ vertices from V_i into $S_{f(V_i)}$ as APPROX. Lemma 6.25 implies that for each $C \in X$ at least $|C|/8$ vertices from C are part of the Δ vertices moved by APPROX. Thus, the union of all $C \in X$ contains at most 8Δ vertices.

By Lemma 6.23, Algorithm 7 pays at most $O(\alpha|C|\log|C|)$ for each $C \in X$ over the entire execution. Thus, its total cost is bounded by

$$\sum_{C\in X} O(\alpha|C|\log|C|) \leq O(\alpha\Delta\log n) = O(\mathrm{ONL}_i \cdot \log n).$$

Consider the vertices of V_i which are not in mixed components. These vertices must have been part of components in which all vertices were originally assigned to $S_{f(V_i)}$. By Lemma 6.12 (which still applies in the ℓ server setting), these vertices were never moved. Thus, they do not incur any cost to the algorithm. □

Next, we show that when the stopping criterion is triggered, the recursive majority voting algorithm pays $O(n\log n)$ and cost of the solution obtained by OPT is $\Omega(\alpha\varepsilon n/(\ell\log\ell))$.

Lemma 6.27. *When the stopping criterion is triggered, (1) the cost paid by Algorithm 7 is $O(\alpha n\log n)$ and (2) the cost paid by* OPT *is $\Omega(\alpha\varepsilon n/(\ell\log\ell))$.*

Proof. Let Y denote the set of all connected components. Part (1) follows from Lemma 6.23 since the total cost paid by Algorithm 7 is

$$\sum_{C\in Y} O(\alpha|C|\log|C|) \leq \sum_{C\in Y} O(\alpha|C|\log n) = O(\alpha n\log n).$$

Now we prove Part (2). Let w be an internal node of $\mathcal{T}$ with children w_0, w_1 and suppose w.l.o.g. that w_0 is overloaded. Since the stopping criterion is triggered, $V(w_0)$ contains at least $\varepsilon n/(\ell\log\ell)$ vertices with label w_1.

Let X be the set of all connected components C with the following properties: C is assigned to a server in $S(w_0)$ at the time at which the stopping criterion is triggered and C contains at least one vertex which is labeled with w_1.

To show that OPT performs $\Omega(\varepsilon n/(\ell\log\ell))$ vertex moves, we prove that OPT performs $\Omega(\#w_1(C))$ vertex moves for each $C \in X$. Part (2) of the lemma follows since the components in X contain at least $\varepsilon n/(\ell\log\ell)$ vertices with label w_1 and thus

$$\mathrm{OPT} \geq \sum_{C\in X} \Omega(\alpha\#w_1(C)) = \Omega(\alpha\varepsilon n/(\ell\log\ell)).$$

We prove that OPT moves at least $\Omega(\#w_1(C))$ vertices for each $C \in X$ by distinguishing two cases for $C \in X$. We define g as the function which maps $C \in X$ to the server it is assigned to in the solution of OPT, i.e., OPT assigns $C \in X$ to server $S_{g(C)}$.

Case 1: $S_{g(C)} \notin S(w_1)$, i.e., in the final assignment of OPT, the vertices in C are assigned to $S_{g(C)} \notin S(w_1)$. Then OPT must perform at least $\#w_1(C)$ vertex moves because it must move all w_1-labeled vertices of C from their initial server in $S(w_1)$ to $S_{g(C)} \notin S(w_1)$.

Case 2: $S_{g(C)} \in S(w_1)$, i.e., in the final solution by OPT, the vertices in C are assigned to a server $S_{g(C)} \in S(w_1)$. We show that C contains at least $|C|/4$ vertices without label w_1. This implies the claim since OPT must move at least $\#w_1(C) \geq |C|/4$ vertices from servers not in $S(w_1)$ to $S_{g(C)} \in S(w_1)$.

Consider a doubling decomposition $(C_1, \ldots, C_k)$ of C (this decomposition exists by Lemma 6.9). After C_1 and C_2 were merged, the algorithm performed a majority voting step and placed $C_1 \cup C_2$ in a server in $S(w_0)$. Thus, $\#w_1(C_1 \cup C_2) \leq |C_1 \cup C_2|/2$ (otherwise, the majority voting step would place $C_1 \cup C_2$ in a server in $S(w_1)$). Hence, $\#w_1(C_1 \cup C_2) = |C_1 \cup C_2| - \#w_1(C_1 \cup C_2) \geq |C_1 \cup C_2|/2$. Since $|C_1 \cup C_2| \geq |C|/2$, we get $\#w_1(C) \geq |C|/4$. □

Proof of Proposition 6.22. The first statement of the proposition is implied by Lemmas 6.20 (perfect partitioning), 6.26 (total cost) and 6.19 (small augmentation). The second statement is proved in Lemma 6.27 (guarantees when stopping criterion is triggered). □

6.4.4 Small–Large–Rebalance Algorithm for Many Servers

To obtain an efficient algorithm in cases where OPT moves many vertices, we reuse the Algorithm 4 from Section 6.3.2.2. Observe that Algorithm 4 also works with ℓ servers because it did not use the fact that there are only two servers. In the setting with ℓ servers, we obtain the following result.

Proposition 6.28. *Suppose that all servers have capacity $(1+\varepsilon)n/\ell$ for $\varepsilon > 0$, i.e., the augmentation is $\varepsilon n/\ell$. Then the cost paid by the more efficient version of Algorithm 4 is $O(\alpha n \log n + (\mathrm{OPT} \cdot \ell \log n)/\varepsilon)$.*

Proof. The proof of the lemma is almost the same as the proof of Proposition 6.6. The only difference is that we need to bound the number of rebalance operations differently.

The number of vertex moves performed by the algorithm which always moves the smaller connected component to the server of the larger connected component is $O(n \log n)$ and, hence, it incurs cost $O(\alpha n \log n)$. Now, whenever a server exceeds its capacity, the algorithm must have moved at least $\varepsilon n/\ell$ vertices. This can only happen $O(\ell \log n/\varepsilon)$ times. By the same arguments as in the proof of Proposition 6.6, each rebalancing operations costs $O(\mathrm{OPT})$. Hence, the cost for all rebalancing steps is bounded by $O(\mathrm{OPT} \cdot \ell \log n/\varepsilon)$. □

We should point out that as in Lemma 6.5, we could also do the repartitioning step of Algorithm 4 by taking *any* perfectly balanced assignment respecting the connected components. In the analysis this would incur $\Theta(n)$ vertex moves for each such step and, hence, yield an algorithm with $O((n\ell \log n)/\varepsilon)$ vertex moves in total. However, unlike in the two-server case, finding a perfectly balanced assignment respecting the connected components is an NP-hard problem. Nonetheless, the problem can be solved approximately in polynomial time at the cost of a constant factor in the competitive ratio. We discuss this in further detail in Section 6.5.2.2.

6.4.5 Bringing It All Together: Theorem 6.16

Proof of Theorem 6.16. Consider the algorithm which first runs Algorithm 7 until the stopping criterion is triggered and then switches to the Algorithm 4 from Section 6.4.4.

If the stopping criterion of the Algorithm 7 is not triggered, then by Proposition 6.22 the cost of the algorithm is $O(\mathrm{OPT} \cdot \log n)$. Thus, its competitive ratio is $O(\log n)$.

If the stopping criterion is triggered, then Algorithm 7 pays $O(\alpha n \log n)$ by Proposition 6.22 and the cost of OPT is $\Omega(\varepsilon n/(\ell \log \ell))$. Furthermore, the cost of Algorithm 4 is $O(\alpha n \log n + (\mathrm{OPT} \cdot \ell \log n)/\varepsilon)$ by Proposition 6.28. Hence, we obtain the following competitive ratio:

$$\begin{aligned}\frac{O(\alpha n \log n + (\mathrm{OPT} \cdot \ell \log n)/\varepsilon)}{\mathrm{OPT}} &= \frac{O(\alpha n \log n)}{\mathrm{OPT}} + O\left(\frac{\ell \log n}{\varepsilon}\right) \\ &\leq O\left(\frac{\alpha n \log n \cdot \ell \log \ell}{\alpha \varepsilon n}\right) + O\left(\frac{\ell \log n}{\varepsilon}\right) \\ &= O\left(\frac{\ell \log n \log \ell}{\varepsilon}\right).\end{aligned}$$

□

6.5 Distributed and Fast Algorithms

In this section we show how the algorithms from Section 6.4 can be implemented in a distributed setting (Section 6.5.1) and how they need to be modified to work in polynomial time at the cost of a slightly worse competitive ratio (Section 6.5.2).

We should point out that even though we discuss the distributed and polynomial time versions of the algorithms separately, they can easily be combined to obtain a distributed algorithm with polynomial computation time.

6.5.1 Distributed Algorithm

While in Section 6.4 we presented algorithms in a centralized model of computation, we now show how Algorithms 4 and 7 can be implemented in a distributed model of computation. For realistic parameter settings, the network traffic caused by our distributed algorithms does not increase (asymptotically) compared to the traffic caused by moving around the vertices between the servers.

In our distributed model of computation we assume that all servers have access to: (1) the number of servers ℓ, (2) the ID of the root server S_0, (3) a shared clock, and (4) all-to-all communication.

When computing the network traffic, we will asymptotically count the number of messages sent by the algorithms and we further assume that each message contains $\Theta(\log n)$ bits. For the sake of simplicity we assume that moving a vertex from one server to another incurs cost $\alpha = \Theta(\log n)$.[7] Because of this simplifying assumption we do not have to distinguish between the number of messages sent by the algorithm and the number of messages used for moving algorithms.

In this distributed model of computation, we obtain the following main result for the distributed versions of the algorithms.

Theorem 6.29. *Consider a system with ℓ servers each of capacity $(1+\varepsilon)n/\ell$ (i.e., augmentation $\varepsilon n/\ell$) for $\varepsilon \in (0, 1/2)$. Let M be the number of vertex moves performed by OPT.*

Then there exists a distributed $O((\ell \log n \log \ell)/\varepsilon)$-competitive algorithm and the number of messages sent by the algorithm is at most:

1. *$O(M \log n)$ if $M = O(\varepsilon n/(\ell \log \ell))$,*
2. *$O((\ell^2 \log n)/\varepsilon + n \log n + (\mathrm{OPT} \cdot \ell \log n)/\varepsilon)$ if $M = \Omega(\varepsilon n/(\ell \log \ell))$.*

In particular, if $\ell = O(\sqrt{\varepsilon n})$, then the algorithm's communication cost does not exceed its cost for moving vertices.

We show for Algorithm 7 (Section 6.5.1.1) and for Algorithm 4 (Section 6.5.1.2) individually how they can be implemented distributedly. After that we prove Theorem 6.29 in Section 6.5.1.3.

6.5.1.1 Making Algorithm 7 Distributed

We start by considering the distributed implementation of Algorithm 7 and obtain the following result.

Lemma 6.30. *Algorithm 7 can be implemented in a distributed model of computation such that the guarantees from Proposition 6.22 still hold. Furthermore, if OPT performs M vertex moves, then we additionally have the following two properties:*

1. *If the stopping criterion is not triggered and the algorithm terminates, then the algorithm sent $O(M \log n)$ messages.*
2. *If the stopping criterion was triggered, the algorithm sent $O(n \log n)$ messages.*

Proof. We start by presenting the necessary modifications to the algorithm and analyze the number of sent messages at the end of the proof.

Let us start by observing that each server can maintain a local representation of the bipartition tree $\mathcal{T}$: Since the number of servers ℓ is known to all servers and $\mathcal{T}$ does not depend on any other quantity, each server can compute $\mathcal{T}$ locally. Next,

[7] Note that this is a realistic assumption since in order to move a vertex, a server must send the ID of a vertex to another server. Sending the ID of the vertex requires $\Theta(\log n)$ bits.

the data structure stores for each vertex its ID (requiring $O(\log n)$ bits) and the ID j of the server S_j it was initially assigned to (requiring $O(\log \ell)$ bits). Thus, the data structure uses $O(\log n)$ bits of storage for each vertex. In other words, it takes $O(1)$ messages to move a vertex between different servers.

Next, we provide the modifications for checking the stopping criterion, small-to-large steps and for majority voting steps.

Before the algorithm moves a component C from server S to server S', S and S' need to check whether the move would trigger the stopping criterion. To do so, S and S' do the following. First, S asks S' for its ID using $O(1)$ messages. Second, S distinguishes between two cases: (1) C contains at most $\lceil \log \ell \rceil$ vertices. Then for each vertex $v \in C$, S sends a message to S' containing the ID of the server v was initially assigned to. This requires $O(|C|)$ messages. (2) C contains more than $\lceil \log \ell \rceil$ vertices. Then S locally computes all internal nodes w of the bipartition tree $\mathcal{T}$ which contain S' as a leaf. For each such node w, let $\bar{w}$ be the sibling of w in $\mathcal{T}$. Now for each w, S computes the number of vertices in C which were initially assigned to a server in $S(\bar{w})$. Then S sends these values to S' using $O(\log \ell)$ messages. Note that in both cases the algorithm does not send more than $O(|C|)$ messages and these messages can be charged to the moving cost of C (which requires $\Omega(|C|)$ messages) which happens after the checking of the stopping criterion. Third, S' receives the messages from S and checks locally whether receiving C would trigger the stopping criterion. If the stopping criterion is not triggered, S' tells S to start moving C. If the stopping criterion is triggered, S sends a message to the root server S_0 about this event. Then S_0 informs all other servers about switching to Algorithm 4. This requires $O(\ell) = O(n \log n)$ messages.

Now suppose the algorithm performs a small-to-large step and the stopping criterion was previously checked and not triggered. In this case, no modifications are necessary: The component C can just be sent from one server to the other at the cost of $O(|C|)$ messages (since each vertex in C can be sent using $O(1)$ messages).

Now suppose a server S needs to perform a majority voting step for a component C. First, observe that S can locally decide whether a majority voting step is necessary for C since it must only check the size of C. Second, when a majority voting step is necessary, S can locally compute which server S' will be the recipient of C: For each vertex $v \in C$, S knows which server v was initially assigned to. Hence, for each v, S can compute the labels of v w.r.t. the bipartitioning scheme from Section 6.4.1 locally. Since S also knows $\mathcal{T}$, S can compute to which server S' the component C should be moved to. These operations do not require any communication between the servers.

To conclude the proof of the lemma, observe that the distributed algorithm performs exactly as many vertex moves as the centralized algorithm. Hence, the guarantees from Proposition 6.22 still hold. Next, we analyze the number of messages sent by the algorithm. A small-to-large step moving a component C requires $O(|C|)$ messages. Checking the stopping criterion before moving a component C requires another $O(|C|)$ messages. Checking whether a majority voting step is necessary requires no communication at all. Hence, the number of messages used by the al-

gorithm is linear in its number of vertex moves. Thus, Proposition 6.22 implies the two additional properties which are claimed in the statement of the lemma. □

6.5.1.2 Making Algorithm 4 Distributed

For the distributed implementation of Algorithm 4 we obtain the following result.

Lemma 6.31. *Algorithm 4 can be implemented in a distributed model of computation such that the guarantees from Proposition 6.28 still hold. Furthermore, if OPT performs M vertex moves, then the algorithm sends at most $O((\ell^2 \log n)/\varepsilon + n \log n + (M\ell \log n)/\varepsilon)$ messages.*

Proof. We start by stating which modifications need to be made to make Algorithm 4 distributed.

First, suppose that Algorithm 4 performs a small-to-large step moving a component C and that this move does not make any server exceed its capacity. In this case, no modifications are necessary and the number of messages sent is $O(|C|)$ as we have seen in the proof of Lemma 6.30.

Second, suppose that a small-to-large step wants to move component C to server S which would cause S to exceed its capacity. Then the algorithm performs the following operations:

1. S informs the root server S_0 that a rebuild is required.
2. S_0 asks all ℓ servers to send the edges that were inserted *and caused the merge of two connected components* since the last rebuild. The servers send of all these edges together with the timestamps when they were inserted.
3. S_0 locally simulates the whole system from the beginning and obtains knowledge about all connected components and which servers they are assigned to.
4. S_0 tells all other servers S_j which components need to be moved and all servers perform the necessary moves.

Since the distributed algorithm performs exactly the same vertex moves as the centralized algorithm, the distributed algorithm is correct and provides the same guarantees as provided in Proposition 6.28. We only need to analyze how many messages the algorithm sends. To do so, we analyze each step separately.

Every time Step 1 is performed, it requires $O(1)$ messages. As the total number of rebuilds is at most $O((\ell \log n)/\varepsilon)$, Step 1 sends $O((\ell \log n)/\varepsilon)$ messages in total.

To bound the number of messages sent in Step 2, recall that in total there are only $O(n)$ edges which merge connected components. Hence, sending these edges requires $O(n)$ messages. Furthermore, when a server did not obtain an edge merging two connected components between two rebuilds, it can inform S_0 about this in $O(1)$ messages. As this can be the case for at most ℓ servers and since there are $O((\ell \log n)/\varepsilon)$ rebuilds, at most $O((\ell^2 \log n)/\varepsilon)$ messages are sent when servers did not receive new edges.

In Step 3, S_0 locally simulates the system. This does not incur any network traffic.

Now consider Step 4. During a rebuild, the number of components which the algorithm needs to reassign is trivially bounded by the number of vertex moves performed during the rebuild. Thus, Proposition 6.28 implies that only $O(n \log n + (\mathrm{OPT} \cdot \ell \log n)/\varepsilon)$ messages are required for all invocations of Step 4.

In total, we obtain that the algorithm sends at most $O((\ell^2 \log n)/\varepsilon) + n \log n + (M\ell \log n)/\varepsilon)$ messages, where M is the number of vertices moved by OPT. □

6.5.1.3 Proof of Theorem 6.29

To prove the claim about the competitive ratio of the algorithm observe that the distributed algorithm performs exactly the same vertex moves as the centralized algorithm. Hence, the cost paid by both algorithms is the same and the distributed algorithm has the same competitive ratio as the centralized algorithm in Theorem 6.16.

The claim about the number of messages sent by the algorithm follows from Lemma 6.30 and Lemma 6.31 and summing over the number of messages.

To prove the last claim of the theorem, we distinguish two cases. If the stopping criterion was not triggered, then the claim holds by Lemma 6.30. If the stopping criterion was triggered, then if $\ell = O(\sqrt{\varepsilon n})$, we obtain that the total number of messages is

$$\begin{aligned} &O((\ell^2 \log n)/\varepsilon) + n \log n + (M\ell \log n)/\varepsilon) \\ &= O((\varepsilon n \log n)/\varepsilon + n \log n + (M\ell \log n)/\varepsilon) \\ &= O(n \log n + (M\ell \log n)/\varepsilon), \end{aligned}$$

which is exactly the number of vertices moved by Algorithm 4. □

6.5.2 Fast Algorithms

In this section, we discuss the computational challenges when computing perfectly balanced assignments. These computational problems occur when Algorithm 4 performs rebalancing steps (see Section 6.3.2 and Section 6.4.4). So far, we were only concerned with algorithms which try to minimize the vertex moves while using potentially exponential running time. We now consider polynomial time algorithms. The only step where our algorithms might use exponential time is during rebalancing. Thus we show next how to perform the rebalancing operations in polynomial time. In the case of $\ell > 2$ servers, our polynomial time algorithms perform slightly more vertex moves than the exponential time algorithms.

We discuss the two server case which can be solved optimally in polynomial time in Section 6.5.2.1. In Section 6.5.2.2, we argue that in the general case with $\ell > 2$ servers this problem is NP-hard. We resolve this issue in Section 6.5.2.3 by computing approximately balanced assignments in polynomial time.

6.5.2.1 Computing Perfectly Balanced Assignments for Two Servers

We consider computing a perfectly balanced assignment respecting the connected components for two servers. Specifically, we provide a dynamic program which can find such an assignment in polynomial time.

The dynamic program works as follows. Suppose $C_1, \dots, C_q$ are the connected components assigned to the two servers. Now let $k_i = |C_i|$ for $i = 1, \dots, q$. We create a set $\mathcal{S}$ consisting of integers with the following property: Each number $s \in \mathcal{S}$ corresponds to a set of connected components $\mathbb{C}$ such that $|\bigcup_{C \in \mathbb{C}} C| = s$. That is, whenever $s \in \mathcal{S}$, there exists a set of connected components which together contain s vertices. For each $s \in \mathcal{S}$, the algorithm maintains a set of connected components explicitly. We denote the components corresponding to value $s \in \mathcal{S}$ by $components(S)$.

At the beginning of a rebalancing step, the algorithm sets $\mathcal{S} = \{0\}$. The connected component corresponding to value 0 is simply the empty set of vertices, i.e., $components(0) = \emptyset$. For $i = 1, \dots, q$ the algorithm does the following. Iterate over all $s \in \mathcal{S}$ and over all components and add $s + k_i$ to $\mathcal{S}$ if $s + k_i \notin \mathcal{S}$ and $C_i \notin components(S)$. Whenever a new value $s + k_i$ is added to $\mathcal{S}$, set $components(s + k_i) = components(s) \cup \{C_i\}$.

As soon as the value $n/2$ is added to $\mathcal{S}$, the dynamic program stops and assigns all vertices in $components(n/2)$ to the left server and all remaining vertices to the right server.

The correctness of the above algorithm is clear by construction. We only need to show that it finishes in polynomial time.

Note that the above dynamic program runs in time $O(q|\mathcal{S}|)$. Now observe that q is bounded by n since there are at most n connected components. Furthermore, for each subset $\mathbb{C} \subseteq \{C_1, \dots, C_q\}$, we have that $\sum_{C \in \mathbb{C}} |C| \leq n$ (because the components in $\mathbb{C}$ cannot contain more than n vertices). Thus, $|\mathcal{S}| \leq n + 1$ because each value $s \in \mathcal{S}$ corresponds to a subset of components $\mathbb{C} \subseteq \{C_1, \dots, C_q\}$ and each value $s \in \{0, \dots, n\}$ is only added once to $\mathcal{S}$. Hence, the algorithm runs in time $O(q|\mathcal{S}|) = O(n^2)$.

6.5.2.2 Computing Perfectly Balanced Assignments for Many Servers

We consider computing a perfectly balanced assignment respecting the connected components for ℓ servers.

Let $C_1, \dots, C_q$ be the connected components which are assigned to the ℓ servers. To find a perfectly balanced assignment respecting the connected components, we need to find a partition of the set $\mathcal{S} = \{|C_1|, \dots, |C_q|\}$ into ℓ subsets $\mathcal{S}_1, \dots, \mathcal{S}_\ell$ such that for each subset $\mathcal{S}_i$ we have that $\sum_{s \in \mathcal{S}_i} s = n/\ell$.

Unfortunately, the above problem is known to be NP-complete, see, e.g., the result about multi-processor scheduling in Garey and Johnson [80]. However, since we prove our results in the online model of computation, which allows unlimited computational power, the algorithm can solve this NP-complete problem. We note

that this problem has also been studied in practice, see, e.g., Schreiber et al. [184] and references therein.

See Section 6.5.2.3 for how this problem can be solved approximately at the cost of a constant in the competitive ratio of the algorithm.

6.5.2.3 Computing Approximately Balanced Assignments for Many Servers

Previously we have we seen that *perfectly* balanced assignments for ℓ servers cannot be computed in polynomial time unless $\mathsf{P} = \mathsf{NP}$ (Section 6.5.2.2). Thus, we now consider computing *approximately* balanced assignments for ℓ servers which is sufficient for our purpose: Let $\varepsilon' > 0$ be a constant. An assignment is $(1+\varepsilon')$-*approximately balanced* if each server has load at most $(1+\varepsilon')n/\ell$. Using this definition, we obtain the following result.

Proposition 6.32. *Let $\varepsilon > \varepsilon' > 0$ be constants and suppose each server has capacity $(1+\varepsilon)n/\ell$. Then a $(1+\varepsilon')$-approximately balanced assignment for ℓ servers can be computed in polynomial time.*

Using the proposition (which we prove at the end of the subsection), we obtain a polynomial time algorithm with a slightly worse competitive ratio than that of Theorem 6.16.

Theorem 6.33. *Given a system with ℓ servers each of capacity $(1+\varepsilon)n/\ell$, for constant $\varepsilon \in (0, 1/2)$, then there exists an $O((\ell^2 \log n \log \ell)/\varepsilon^2)$-competitive algorithm which runs in polynomial time.*

Proof. First, observe that Algorithm 7 runs in polynomial time. Thus, the result of Proposition 6.22 also holds for polynomial time algorithms.

Second, consider a modification of Algorithm 4 where at each rebalancing step we compute a $(1+\varepsilon')$-approximately balanced assignment for $\varepsilon' = \varepsilon/2$. Such an approximately balanced assignment can be computed in polynomial time due to Proposition 6.32. Thus, the modified algorithm runs in polynomial time.

Observe that now all steps of the resulting algorithm can be computed in polynomial time. It is left to bound the competitive ratio of the modified algorithm.

We start by bounding the cost paid by the modified version of Algorithm 4. Note that each approximate rebalancing step incurs cost at most $O(\alpha n)$; recall that α denotes the cost for moving a vertex to a different server. Now we bound the number of approximate rebalancing steps. Recall from Lemma 6.4 that the number of vertex moves due to small-to-large steps is at most $O(n \log n)$. Now whenever a new approximately balanced assignment is computed, the small-to-large steps must have moved at least $\Omega((\varepsilon - \varepsilon')n/\ell)$ vertices to exceed the capacity of one of the servers. Thus, the total number of approximate rebalancing operations is bounded by $O((\ell \log n)/(\varepsilon - \varepsilon'))$ and, hence, the total cost of Algorithm 4 with approximate rebalancing steps is bounded by $O((\alpha n \ell \log n)/(\varepsilon - \varepsilon'))$.

Altogether, we obtain the following competitive ratio by following the steps from the proof of Theorem 6.16 (Section 6.4.5):

$$\frac{O(\alpha n \log n + (\alpha n \ell \log n)/(\varepsilon - \varepsilon'))}{\mathrm{OPT}} = O\left(\frac{\alpha n \log n + (\alpha n \ell \log n)/(\varepsilon - \varepsilon')}{\alpha \varepsilon n/(\ell \log \ell)}\right)$$
$$= O\left(\frac{\ell^2 \log n \log \ell}{\varepsilon^2}\right),$$

where in the last step we used that $\varepsilon - \varepsilon' = \varepsilon/2$. □

To prove Proposition 6.32, we consider the makespan minimization problem in which there are k jobs with processing times $p_1, \ldots, p_k$ which must be assigned to ℓ identical machines. Given an assignment of the jobs to the machines, the maximum running time time of any machine is called the *makespan*. The goal is to find an assignment of the jobs to the machines which minimizes the makespan.

The makespan minimization problem is known to be NP-hard but Hochbaum and Shmoys [105] presented a polynomial time approximation scheme (PTAS).

Lemma 6.34 (Hochbaum and Shmoys [105])**.** *Let $\varepsilon' > 0$ be a constant. Then there exists an algorithm which computes a $(1+\varepsilon')$-approximate solution for the makespan minimization problem in polynomial time.*

Using the result from the lemma we can prove Proposition 6.32.

Proof of Proposition 6.32. Suppose the system currently contains connected components $C_1, \ldots, C_k$. We consider these connected components as the jobs of the makespan minimization problem with processing times $p_i = |C_i|$ for $i = 1, \ldots, k$. The machines correspond to the ℓ servers.

Note that the optimal solution for the instance of the makespan minimization problem is n/ℓ: Since we have made the assumption that in the final assignment all servers have load exactly n/ℓ, there must exist a perfectly balanced assignment from the components C_i to the servers S_i. In other words, there exists an assignment of the jobs to the machines such that each machine has running time n/ℓ and, hence, the optimal makespan is n/ℓ.

By running the algorithm from Lemma 6.34, we obtain a $(1 + \varepsilon')$-approximate solution for the makespan minimization problem. Since the optimal solution for this problem is n/ℓ, each machine has load at most $(1+\varepsilon')n/\ell$ in the solution returned by the algorithm from Lemma 6.34. Assigning the components C_i to the servers in exactly the same way as the corresponding jobs are assigned to the corresponding machines, we obtain a $(1 + \varepsilon')$-approximately balanced assignment in polynomial time. □

6.6 Lower Bounds

To study the optimality of our algorithms, we derive bounds on the competitive ratios which can be achieved by *any* deterministic online algorithm.

The following theorem provides a lower bound of $\Omega(1/\varepsilon + \log n)$. The lower bound has the following two main consequences: (1) If an algorithm is only allowed to use constant augmentation (i.e., servers of capacity $n/\ell + O(1)$), then the lower bound implies that any algorithm must have a competitive ratio of $\Omega(n)$.[8] (2) The lower bound holds even in the setting in which there are only two servers. Thus, the algorithm from Section 6.3 for the two server setting is close to optimal (up to a $O(\min\{1/\varepsilon, \log n\})$ factor) and the generalized algorithm from Section 6.4 is optimal up to a $O(\ell \log \ell \min\{1/\varepsilon, \log n\})$ factor.

Theorem 6.35. *Suppose there are two servers of capacity $(1+\varepsilon)n/2$ for $\varepsilon \leq 0.98$. Then any deterministic online algorithm must have a competitive ratio of $\Omega(1/\varepsilon + \log n)$.*

To prove the theorem, we show in Section 6.6.1 that there exist input sequences such that *either* an algorithm always assigns vertices of the same connected component to the same server *or* it has prohibitively high cost. Using this fact, we prove our concrete lower bounds in Section 6.6.2.

In the next chapter, in Section 7.7.1, we will partially improve upon the above lower bound by presenting a lower bound of $\Omega(\ell \log(n/\ell))$, but this lower bound only holds for a smaller range of values for ε. Hence, we still present the full proof of the theorem in this chapter.

6.6.1 Assigning Connected Components to Servers

In this subsection, we give an important reduction which will be useful to derive the lower bounds in the next subsection (Section 6.6.2). This reduction lets us assume that every competitive algorithm will always assign vertices of the same connected component to the same server.

More concretely, we show that every sequence of edges σ can be manipulated to a new edge sequence σ' such that: (1) σ reveals the same edges as σ' and (2) on input σ', every algorithm *either* moves the vertices of the same connected components to the same server, *or* has prohibitively high cost and, hence, cannot be competitive.

We first prove the following technical lemma.

Lemma 6.36. *Consider a sequence σ which reveals the edges $\emptyset \neq E^* \subseteq E$. Let $C_1, \ldots, C_q$ be the connected components induced by E^*.*

Then for each initial assignment there exists an input sequence σ' consisting only of edges in E^* such that either (1) at some point during the input sequence the algorithm assigns all vertices from each C_i to the same S_j or (2) the cost of the algorithm is at least $\Omega(\alpha n^3)$.

Proof. We will construct an input sequence σ' provided by the adversary such that either Property (1) or Property (2) must hold.

[8] To obtain servers of capacity $n/\ell + O(1)$, we must set $\varepsilon = O(1)/n$.

Consider an arbitrary initial assignment and pick the ground truth components V_i such that they do not coincide with the initial assignment of the vertices to the servers, i.e., $V_i \neq V_{\text{init}}(S_j)$ for all i, j. Let $E^* = \{e'_1, \dots, e'_t\}$ be the edges revealed by the adversary and suppose that E^* contains at least one edge (u, v) such that u and v are assigned to different servers in the initial assignment.

Now consider the input sequence $\sigma' = (e_1, \dots, e_r)$ with $r = \lceil \alpha n^3 t \rceil$ which consists of the edges $(e'_1, \dots, e'_t)$ in E^* concatenated $\lceil \alpha n^3 \rceil$ times.

Suppose that while running the algorithm there always exists a C_i such that not all vertices from C_i are assigned to the same server S_j, i.e., Claim (1) does not apply. We show that then Claim (2) must apply.

Consider the state of the algorithm prior to a single subsequence containing the edges $(e'_1, \dots, e'_t)$. By assumption at least one edge e'_i must be between two vertices from different servers. Now the algorithm must either pay 1 for communication along this edge or it must move one of the edge's endpoints at the cost of α to avoid paying for communication along this edge. Thus, the algorithm must pay at least $\Omega(1)$ for the subsequence $(e'_1, \dots, e'_t)$.

As there are $\lceil \alpha n^3 \rceil$ such subsequences, the algorithm must pay at least $\Omega(\alpha n^3)$ in total. □

As we will see, the lemma essentially allows us to assume that every algorithm which obtains an edge between vertices on different clusters, must move their connected components to the same cluster. That is, given an input sequence σ, in our lower bound proof, we can employ Lemma 6.36 to obtain an input sequence σ' which does not reveal any additional edges and which forces every algorithm to have Property (1) or Property (2).

Now observe that if an algorithm has Property (2), since the cost of OPT are always bounded by $O(\alpha n)$ (OPT moves each vertex at most once), the algorithm cannot be competitive: the competitive ratio must be at least $\Omega(n^2)$, much higher than the competitive ratios derived in this chapter. Hence, in the following we can assume that every algorithm with a competitive ratio better than $\Omega(n^2)$ must satisfy Property (1) of Lemma 6.36.

6.6.2 Lower Bound Proofs

In this subsection, we prove Theorem 6.35 by proving two different lower bounds: The first lower bound asserts a competitive ratio of $\Omega(1/\varepsilon)$ and the second lower bounds asserts a competitive ratio of $\Omega(\log n)$.

In the lower bound constructions we heavily exploit that we provide hard instances against *deterministic* algorithms, i.e., we will rely on the fact that at each point in time the adversary knows exactly which assignment the online algorithm created.

Furthermore, we assume that after each edge which was provided by the adversary, the algorithm creates an assignment such that all vertices of the same con-

nected component are assigned to the same server. This assumption is admissible by the discussion in Section 6.6.1.

We start by proving the lower bound of $\Omega(1/\varepsilon)$.

Lemma 6.37. *Consider the setting with two servers which both have capacity* $(1+\varepsilon)n/2$ *for* $\varepsilon > 0$.

Then for each deterministic online algorithm ONL there exists an input sequence σ *such that the cost of ONL is* $\Omega(\alpha n)$ *and the cost paid by OPT is* $O(\alpha\varepsilon n)$. *Thus, the competitive ratio of every online algorithm is* $\Omega(1/\varepsilon)$.

Proof. Choose an arbitrary initial assignment of n vertices to the ℓ servers. Let $K = \varepsilon n/2$ denote the allowed augmentation of the servers. The initial assignment is as follows. In the left server, there are $q = n/(2(K+1))$ connected components $C_1, \dots, C_q$ of size $K+1$. On the right server, we build one connected component of size $K+1$ denoted C and one large connected component of size $n-K-1$ denoted C'. First, the adversary provides all edges of these connected components at no cost to the algorithm.

Then the adversary inserts an edge from a vertex in C_1 to a vertex in C. Since C_1 has size $K+1$ and the right server currently has $n/2$ vertices, the algorithm cannot move C_1 to the right server. For the same reason, the algorithm cannot move C to the left server either. Thus, the algorithm's only option to bring C_1 and C to the same server is to replace C with some C_i at the cost of $2\alpha(K+1)$.

We will refer to the merged connected component of size $2(K+1)$ as D. Note that D must be on the left server. Now let C_i be the connected component of size $K+1$ on the right server. The adversary adds an edge from a vertex in D to a vertex in C_i. By the same reasoning as before, the algorithm must now pick some C_j, $j \neq i$, of size $K+1$ from the left server and swap it with C_i. This costs another $2\alpha(K+1)$.

The adversary continues the previous procedure until only a C_i of size $K+1$ is left on the left server and then she connects C_i and C'. This gives the final partitioning of the vertices.

We observe that each vertex which is on the left server at the very end, has been on the right server exactly once during the execution of the algorithm. Thus, the costs paid by the algorithm must be $\Omega(\alpha n)$.

Note that OPT pays exactly $\alpha(K+1)$ because it can determine beforehand which C_i must be moved to the right server and only move that connected component. Before, we have seen that any deterministic algorithm must pay at least $\Omega(\alpha n)$. Thus, the competitive ratio is $\Omega(n/K)$. □

Next, we prove the $\Omega(\log n)$ lower bound for the competitive ratio of deterministic algorithms. Later, in Section 7.7.1, we will prove a stronger lower bound of $\Omega(\ell \log(n/\ell))$ in Theorem 7.28, but this stronger lower bound only applies for smaller values of ε. Furthermore, in Section 7.7.2, we will also see a randomized version of the lower bound from the lemma (see Proposition 7.37).

Lemma 6.38. *Consider the setting with two servers which both have capacity* $(1+\varepsilon)n/2$ *for* $\varepsilon \leq 0.98$.

Then for each deterministic online algorithm ONL there exists an input sequence σ *such that the cost paid by ONL is* $\Omega(\alpha n \log n)$ *and the cost paid by OPT is* $O(\alpha n)$. *Thus, the competitive ratio of every deterministic online algorithm is* $\Omega(\log n)$.

Proof. Choose an arbitrary initial assignment of n vertices to the ℓ servers. Since we want to prove a lower bound, we can assume that n is a power of 2. Thus, suppose that $n = 2^a$ for $a \geq 1000$.

In our hard instance, we are creating a sequence of edge insertions which proceeds in $\Theta(\log n)$ rounds. When round i starts, all connected components have size 2^i induced by the previously provided edges, and when round i finishes, all connected components have size 2^{i+1}. We show that ONL pays $\Omega(\alpha n)$ in each round. This implies the claimed cost of $\Omega(\alpha n \log n)$ for ONL. The cost for OPT follows immediately from Lemma 6.3 which states that OPT never pays more than $O(n)$ when there are only two servers.

When ONL starts and no edge was provided by the adversary, all connected components have size $1 = 2^0$, i.e., the connected components are isolated vertices.

Now suppose round $i = 0, \ldots, \log n$ starts. By induction, all connected components have size 2^i. We now define a sequence of edge insertions for round i which forces ONL to pay $\Omega(\alpha n)$ and after which all connected components have size 2^{i+1}.

Let z denote the current number of connected components of size 2^i. When round i starts, there are exactly $z = n/2^i = 2^{a-i}$ connected components of size 2^i each. Recall that each server has capacity $(1+\varepsilon)n/2$. Thus, at most

$$y_i = (1+\varepsilon)n/2^{i+1} \leq 1.98 \cdot 2^{a-i-1}$$

connected components of size 2^i can be assigned to each server.

Now suppose there exists an edge (u, v) such that C_u and C_v are of size 2^i and they are assigned to different servers; we call such an edge *expensive*. When the adversary inserts an expensive edge, ONL must pay $\Omega(\alpha 2^i)$ for moving C_u or C_v to a different server.

The strategy of the adversary is to insert expensive edges as long as they exist. Once no expensive edges exist anymore, the adversary connects all remaining components of size 2^i arbitrarily until all components have size 2^{i+1}.

Note that expensive edges exist as long as $z > y_i$ (because when this inequality is satisfied, not all connected components of size 2^i can be assigned to the same server). Furthermore, observe that when the adversary inserts an expensive edge, z decreases by 2.

Now we prove a lower bound on the number of expensive edges p. By the previous arguments, p must be large enough such that:

$$z = 2^{a-i} - 2p \leq y_i.$$

Solving this inequality for p, we obtain

$$\begin{aligned} p &\geq 2^{a-i-1} - 1.98 \cdot 2^{a-i-2} \\ &= 2^{a-i-1}(1 - 0.99) \\ &= 0.01 \cdot 2^{a-i-1}. \end{aligned}$$

We conclude that that adversary can perform $\Omega(2^{a-i})$ expensive edge insertions. Since for each of these edge insertions, ONL must pay $\Omega(\alpha 2^i)$, we obtain that the cost paid by ONL in round i is

$$\Omega(\alpha \cdot 2^{a-i} \cdot 2^i) = \Omega(\alpha \cdot 2^{a-2}) = \Omega(\alpha n). \qquad \square$$

6.7 Sample Applications: A Distributed Union–Find Algorithm and Online k-Way Partitioning

In this section we provide two sample applications for our model and our algorithms. First, we show that our results can be used to solve a distributed union–find problem and we give an example where a union–find data structure is used in practice. Second, we show that our algorithms imply competitive algorithms for an online version of the k-way partitioning problem.

6.7.1 Distributed Union Find

Recall that in the *static* union–find problem, there are n elements from a universe $\mathcal{U}$ and initially there are n sets containing one element each. The data structure supports two operations: $\mathrm{union}(u, v)$ and $\mathrm{find}(u)$. Given two elements $u, v \in \mathcal{U}$, the operation $\mathrm{union}(u, v)$ merges the sets containing u and v. The operation $\mathrm{find}(u)$ returns the set containing u.

In the distributed setting we consider, elements are stored across ℓ servers. Each server has enough capacity to store $(1 + \varepsilon)n/\ell$ elements and we have the natural constraint that elements from the same set must always be stored on the same server (in order to maintain locality for elements from the same set). We consider a setting in which all sets have size n/ℓ when the algorithm finishes.

Note that if the sets of $u, v \in \mathcal{U}$ are stored on different servers when the operation $\mathrm{union}(u, v)$ is performed, one of the sets containing u or v must be moved to a different server. The goal of an algorithm is to minimize the moving cost caused by union-operations.

When analyzing the moving cost, we will compare with an optimal offline algorithm which knows in advance which union-operations will be performed. Thus, the optimal algorithm can move from the initial assignment to the final assignment at the minimum possible cost. For our analysis we will compute the competitive ratio between an online algorithm solving the above problem and the optimal offline algorithm (as also detailed in Section 6.2).

Using the algorithms from Sections 6.4 and 6.5.1, we obtain the following result.

Theorem 6.39. *Consider a system with ℓ servers each of capacity $(1+\varepsilon)n/\ell$ for $\varepsilon \in (0, 1/2)$. Then there exists a distributed $O((\ell \log n \log \ell)/\varepsilon)$-competitive algorithm for the distributed union–find problem. Moreover, for $\ell = O(\sqrt{\varepsilon n})$ servers, the algorithm's communication cost does not exceed its cost for moving vertices.*

Proof. The theorem follows immediately from Theorem 6.29 by the following reduction from the model in Section 6.2. We identify vertices in the model from Section 6.2 with elements from the universe $\mathcal{U}$ in the union–find model. Furthermore, for each operation $\mathrm{union}(u, v)$ we insert an edge (u, v) into the model from Section 6.2. Since all algorithms we considered always collocate vertices from the same connected component, they satisfy the constraint that elements from the same set must be assigned to the same server. Moreover, in our analysis we were able to focus on the number of vertex moves due to Lemma 6.1. In our proofs, we showed competitive bounds for the number of vertex moves performed by the algorithm from Theorem 6.29 compared with an optimal offline algorithm. Thus, the same bounds as derived in Theorem 6.29 apply. □

For $\ell = \Omega(\sqrt{\varepsilon n})$ servers and the exact number of messages sent by the algorithm, see Theorem 6.29. The guarantees from Theorem 6.29 carry over immediately.

An examples where distributed union–find data structures are used in practice is search engines [46]. A search engine stores many different documents from the Web over multiple servers. Now union–find data structures are used to collocate duplicate documents on the same server, i.e., when documents u and v are identified as duplicates the operation $\mathrm{union}(u, v)$ is used to collocate these documents (and all previously identified duplicates) on the same server. Furthermore, union–find data structures are used to find blocks in dense linear systems and in pattern recognition tasks (see Cybenko et al. [58] and references therein).

6.7.2 Online k-Way Partitioning

The model and algorithms we study in this chapter can also be used to solve an online variant of the k-way partition problem [184]. In the static version of the k-way partition problem one is given a (multi-)set of integers $\mathcal{S}$ and the task is to partition $\mathcal{S}$ into k subsets $\mathcal{S}_1, \ldots, \mathcal{S}_k$ such that the sum of all subsets is (approximately) equal.

Our model and our algorithms can be used to solve the following online version of this fundamental problem. Initially, $\mathcal{S}$ contains n integers and all integers are 1. Each integer is assigned to one of ℓ bins and each bin has capacity $(1+\varepsilon)n/\ell$. Now in an online sequence of operations, an adversary picks two integers from $\mathcal{S}$ and these integers are added. For example, after adding integers $a, b \in \mathcal{S}$, $\mathcal{S}$ becomes $\mathcal{S} = (\mathcal{S} \cup \{a+b\}) \setminus \{a, b\}$. During this sequence of operations an online algorithm must ensure that the load of all bins is always bounded by $(1+\varepsilon)n/\ell$. We work under the assumption that after each operation there always exists an assignment

from the integers in $\mathcal{S}$ to the bins such that each bin has load exactly n/ℓ. We further assume that at the end of the sequence of operations there are ℓ integers and each integer is n/ℓ.

Note that when two integers $a, b \in \mathcal{S}$ from different bins are added, either a or b must be moved to a different bin. This might cause that bin to exceed its capacity.

We will analyze algorithms which have small moving cost. That is, the cost of an algorithm is the sum of the numbers it has moved. We consider the competitive analysis of online algorithms compared with an optimal offline algorithm which knows the sequence of additions in advance and which can move the numbers at optimal cost.

We then obtain the following result for the k-way partitioning problem.

Theorem 6.40. *Consider a system with ℓ bins each of capacity $(1+\varepsilon)n/\ell$ for $\varepsilon \in (0, 1/2)$. Then there exists a $O((\ell \log n \log \ell)/\varepsilon)$-competitive algorithm for the k-way partition problem.*

Proof. We can relate the online version of the k-way partition problem to the model we study by identifying integers and the sizes of connected components. Initially, we identify each $s \in \mathcal{S}$ with a single vertex. Note that this can be done since initially $s = 1$ and thus s and the size of its corresponding connected component are the same. After that, when two integers a and b are added, we take their corresponding connected components C_a and C_b and insert an edge between them. Note that the resulting integer $a + b$ corresponds to the connected component $C_a \cup C_b$ and their sizes agree, i.e., $a + b = |C_a \cup C_b|$. Now observe that summing the moving cost for integers is the same as counting the number of vertex reassignments for connected components. Thus, the result of the theorem follows from Theorem 6.16. □

6.8 Related Work

The design of more flexible networked systems that can adapt to their workloads has received much attention over the last years, with applications for traffic engineering [106, 112], load-balancing [65, 163], network slicing [186], server migration [33], switching [43, 72], or even adjusting the network topology [83]. The impact of distributed applications on the communication network is also well-documented in the literature [55, 76, 128, 148, 188]. Several empirical studies exploring the spatial and temporal locality in traffic patterns found evidence that these workloads are often *sparse and skewed* [16, 83, 115, 178], introducing optimization opportunities. E.g., studies of reconfigurable datacenter networks [83, 91] have shown that for certain workloads, a demand-aware datacenter network can achieve a performance similar to a demand-oblivious datacenter network at 25-40% lower cost [83, 91].

However, much less is known about the algorithmic challenges underlying such workload-adaptive networked systems, the focus of our work. From an online algorithm perspective, our problem is related to reconfiguration problems such as

online page (resp. file) migration [29, 36] as well as server migration [33] problems, k-server [71] problems, or online metrical task systems [41]. In contrast to these problems, in our model, requests do not appear somewhere in a graph or metric space but *between communication partners*. From this perspective, our problem can also be seen as a "distributed" version of online paging problems [70, 139, 189, 212] (and especially their variants *with bypassing* [8, 66]) where access costs can be avoided by moving items to a *cache*: in our model, access costs are avoided by collocating communication partners on the same *server* (a "distributed cache").

The static version of our problem, how to partition a graph, is a most fundamental and well-explored problem in computer science [198], with many applications, e.g., in community detection [1]. The balanced graph partitioning problem is related to minimum bisection problems [68], and known to be hard even to approximate [18]. The best approximation today is due to Krauthgamer [120]. In contrast, we are interested in a dynamic version of the problem where the edges of the to-be-partitioned graph are revealed over time, in an online manner. Further, the offline problem of embedding workloads in a communication-efficient manner has been studied in the context of the minimum linear arrangement problem [171] and the virtual network embedding problem [214], however, without considering the option of migrations. In this regard, our results feature an interesting connection to the *itinerant list update* model [160], a kind of "dynamic" minimum linear arrangement problem which allows for reconfigurations and, notably, considers pair-wise requests. However, communication is limited to a linear line and so far, only non-trivial *offline* solutions are known.

One of the applications of the problem we study is a distributed union–find data structure (see Section 6.7.1). Union find data structures have been initially proposed in the centralized setting and efficient algorithms were derived [78, 192]. Later, parallel versions of union–find data structures were considered in a shared memory setting in which the goal was to derive wait-free algorithms [17]; also external memory algorithms were considered [9]. To the best of our knowledge studies of union–find data structures in a distributed memory setting were only conducted experimentally, see (for example) [58, 135, 164, 165].

The second application we presented was as online k-way partitioning (Section 6.7.2). The k-way partitioning problem is known to be NP-hard as it constitutes a very simple scheduling problem [80]. The problem has also been researched in practice, see, e.g., [118, 184] and references therein. We are not aware of literature studying the online version of the problem which we have considered.

The paper most closely related to our results is by Avin et al. [20, 23] who studied a more general version of the problem considered. In their model, request patterns can change arbitrarily over time, and in particular, do not have to follow a partition and hence "cannot be learned". Indeed, as we have shown, learning algorithms can perform significantly better: in [23], it was shown that for constant ℓ any deterministic online algorithm must have a competitive ratio of at least $\Omega(n)$ unless it can collocate *all* nodes on a single server, while we have presented an $O(\log n)$-competitive online algorithm. Thus, our result is exponentially better than what

can possibly be achieved in the model of [23].

6.9 Conclusion

Motivated by the increasing resource allocation flexibilities available in modern compute infrastructures, we initiated the study of online algorithms for adjusting the embedding of workloads according to the specific communication patterns, to reduce communication and moving costs. In particular, we presented algorithms and derived upper and lower bounds on their competitive ratio.

In Chapter 7, we will improve upon the upper and lower bounds of this chapter asymptotically but this comes at the cost of a super-polynomial dependency on ε in the competitive ratio. It is an open question whether we can obtain asymptotically tight competitive ratios that only polynomially depend on ε.

The algorithm from this chapter appears to be quite implementable when one replaces the exhaustive enumeration step from Section 6.3.2.2 with suitable heuristics. Thus, it would be interesting to see how the algorithm performs in practice.

In the future, it will be natural to consider algorithms which do not collocate all communication partners (by considering more general pattern models than the one we proposed). Also, studying collocation in specific networks such as Clos networks, which are frequently encountered in datacenters, would be intriguing.

CHAPTER 7

Tight Bounds for Online Graph Partitioning

We consider the following online optimization problem. We are given a graph G and each vertex of the graph is assigned to one of ℓ servers, where servers have capacity k and we assume that the graph has $k \cdot \ell$ vertices. Initially, G does not contain any edges and then the edges of G are revealed one-by-one. The goal is to design an online algorithm ONL, which always places the connected components induced by the revealed edges on the same server and never exceeds the server capacities by more than εk for constant $\varepsilon > 0$. Whenever ONL learns about a new edge, the algorithm is allowed to move vertices from one server to another. Its objective is to minimize the number of vertex moves. More specifically, ONL should minimize the competitive ratio: the total cost ONL incurs compared to an optimal offline algorithm OPT.

The problem was recently introduced by Henzinger et al. [101, SIGMETRICS'2019 and Chapter 6] and is related to classic online problems such as online paging and scheduling. It finds applications in the context of resource allocation in the cloud and for optimizing distributed data structures such as union–find data structures.

Our main contribution is a polynomial-time randomized algorithm, that is asymptotically optimal: we derive an upper bound of $O(\log \ell + \log k)$ on its competitive ratio and show that no randomized online algorithm can achieve a competitive ratio of less than $\Omega(\log \ell + \log k)$. We also settle the open problem of the achievable competitive ratio by deterministic online algorithms, by deriving a competitive ratio of $\Theta(\ell \log k)$; to this end, we present an improved lower bound as well as a deterministic polynomial-time online algorithm.

Our algorithms rely on a novel technique which combines efficient integer programming with a combinatorial approach for maintaing ILP solutions. More precisely, we use an ILP to assign the connected components induced by the revealed

edges to the servers; this is similar to existing approximation schemes for scheduling algorithms. However, we cannot obtain our competitive ratios if we run the ILP after each edge insertion. Instead, we identify certain types of edge insertions, after which we can manually obtain an optimal ILP solution at zero cost without resolving the ILP. We believe this technique is of independent interest and will find further applications in the future.

7.1 Introduction

Distributed cloud applications generate a significant amount of network traffic [148]. To improve their efficiency and performance, the underlying infrastructure needs to become *demand-aware*: frequently communicating endpoints need to be allocated closer to each other, e.g., by collocating them on the same server or in the same rack. Such optimizations are enabled by the increasing resource allocation flexibilities available in modern virtualized infrastructures, and are further motivated by rich spatial and temporal structure featured by communication patterns of data-intensive applications [30].

This chapter studies the algorithmic problem underlying such demand-aware resource allocation in scenarios where the communication pattern is not known ahead of time. Instead, the algorithm needs to learn a communication pattern in an online manner, dynamically collocating communication partners while minimizing reconfiguration costs. It has recently (in Chapter 6) been shown that this problem can be modeled by the following *online graph partitioning problem* [101]: Let k and ℓ be known parameters. We are given a graph G which initially does not contain any edges. Each vertex of the graph is assigned to one of ℓ servers and each server has *capacity* k, i.e., stores k vertices. Next, the edges of G are revealed one-by-one in an online fashion and the algorithm has to guarantee that *every connected component is placed on the same server (cc-condition).* This is possible, as it is guaranteed that there always exists an assignment of the connected components to servers such that no connected component is split across multiple servers. Thus after each edge insertion, the online algorithm has to decide which vertices to move between servers to guarantee the cc-condition. Each vertex move incurs a cost of $1/k$. The optimal offline algorithm knows all the connected components, computes, dependent on the initial placement of the vertices, the minimum-cost assignment of these connected components to servers and moves the vertices to their final servers after the first edge insertion. It requires no further vertex moves. To measure the performance of an online algorithm ONL we use the competitive ratio: the total cost of ONL divided by the total cost of the optimal offline algorithm OPT. In this setting, no deterministic online algorithm can have a better competitive ratio than $\Omega(k)$ [101].

Thus, we relax the server capacity requirement for the online algorithm: Specifically, the online algorithm is allowed to place up to $(1+\varepsilon)k$ vertices on a server at any point in time. We call this problem the *online graph partitioning problem.*

Henzinger et al. [101] (and Chapter 6) studied this problem and showed that

the previously described demand-aware resource allocation problem reduces to the online graph partitioning problem: vertices of the graph G correspond to communication partners and edges correspond to communication requests; thus, by collocating the communication partners based on the connected components of the vertices, we minimize the network traffic (since all future communications among the revealed edges will happen locally). They also showed how to implement a distributed union–find data structure with this approach. Algorithmically, [101] presented a deterministic exponential-time algorithm with competitive ratio $O(\ell \log \ell \log k)$ and complemented their result with a lower bound of $\Omega(\log k)$ on the competitive ratio of any deterministic online algorithm. While their derived bounds are tight for $\ell = O(1)$ servers, there remains a gap of factor $O(\ell \log \ell)$ between upper and lower bound for the scenario of $\ell = \omega(1)$. Furthermore, their lower bound only applies to deterministic algorithms and thus it is a natural question to ask whether randomized algorithms can obtain better competitive ratios.

7.1.1 Our Contributions

Our main contribution is a polynomial-time randomized algorithm for online graph partitioning which achieves a polylogarithmic competitive ratio. In particular, we derive an $O(\log \ell + \log k)$ upper bound on the competitive ratio of our algorithm, where ℓ is the number of servers and k is the server capacity. We also show that no randomized online algorithm can achieve a competitive ratio of less than $\Omega(\log \ell + \log k)$. The achieved competitive ratio is hence asymptotically optimal.

We further settle the open problem of the competitive ratio achievable by deterministic online algorithms. To this end, we derive an improved lower bound of $\Omega(\ell \log k)$, and present a polynomial-time deterministic online algorithm which achieves a competitive ratio of $O(\ell \log k)$. Thus, also our deterministic algorithm is optimal up to constant factors in the competitive ratio.

These results improve upon the results of [101] and Chapter 6 in three respects: First, our deterministic online algorithm has competitive ratio $O(\ell \log k)$ and polynomial run-time, while the algorithm in [101] has competitive ratio $O(\ell \log \ell \log k)$ and requires exponential time. Second, we present a significantly higher and matching lower bound of $\Omega(\ell \log k)$ on the competitive ratio of any deterministic online algorithm. Third, we initiate the study of randomized algorithms for the online graph partitioning problem and show that it is possible to achieve a competitive ratio of $O(\log \ell + \log k)$ and we complement this result with a matching lower bound. Note that the competitive ratio obtained by our randomized algorithm provides an exponential improvement over what any deterministic algorithm can achieve in terms of the dependency on the parameter ℓ. We should note, however, that the competitive ratio of the algorithm derived in this chapter has a much worse dependency on ε than the previous work.

Technical Novelty. We will now provide a brief overview of our approach and its technical novelty. Since our deterministic and our randomized algorithms

are based on the same algorithmic framework, we will say *our algorithm* in the following.

Our algorithm keeps track of the set of connected components induced by the revealed edges. We will denote the connected components as *pieces* and when two connected components become connected due to an edge insertion, we say that the corresponding pieces are *merged*.

The algorithm maintains an assignment of the pieces onto the servers which we call a *schedule*. We will make sure that the schedule is *valid*, i.e., that every piece is assigned to some server and that the capacities of the servers are never exceeded by more than the allowed additive εk. To compute valid schedules, we solve an integer linear program (ILP) using a generic ILP solver and show how the solution of the ILP can be transformed into a valid schedule. We ensure that the ILP is of constant size and can, hence, be solved in polynomial time. Next, we show that when two pieces are merged due to an edge insertion, the schedule does not change much, i.e., we do not have to move "too many" pieces between the servers. We do this using a *sensitivity analysis* of the ILP, which guarantees that when two pieces are merged, the solution of the ILP does not change by much. Furthermore, we prove that this change in the ILP solution corresponds to only slightly adjusting the schedules, and thus only moving a few pieces.

However, the sensitivity analysis alone is not enough to obtain the desired competitive ratio. Indeed, we identify certain types of merge-operations for which the optimal offline algorithm OPT might have very small or even zero cost. In this case, adjusting the schedules based on the ILP sensitivity would be too costly: the generic ILP solver from above could potentially move to an optimal ILP solution which is very different from the current solution and, thus, incur much more cost than OPT. Hence, to keep the cost paid by our algorithm low, we make sure that for these special types of merge-operations, our algorithm sticks extremely close to the previous ILP solution, incurs zero cost for moving pieces and still obtains an optimal ILP solution. The optimality of the algorithm's solution after such merge-operations is crucial as otherwise, we could not apply the sensitivity analysis after the subsequent merge-operations. To the best of our knowledge, our algorithm is the first to *interleave ILP sensitivity analysis with manual maintenance of optimal ILP solutions.*

More specifically, we assume that each server has a unique color and consider each vertex as being colored by the color of its initial server. In our analysis we identify two different types of pieces: *monochromatic* and *non-monochromatic* ones. In the monochromatic pieces, "almost all" of the vertices have the same color, i.e., were initially assigned to the same server, while the non-monochromatic pieces contain "many" vertices which started on different servers. We show that we have to treat the monochromatic pieces very carefully because these are the pieces for which OPT might have very small or even no cost. Hence, it is desirable to always schedule monochromatic pieces on the server of the majority color. Unfortunately, we show that this is not always possible. Indeed, the hard instances in our lower bounds show that an adversary can force any deterministic algorithm to create schedules

with *extraordinary* servers. Informally, a server s is extraordinary if there exists a monochromatic piece p for which almost all of its vertices have the color of s but p is not scheduled on s (see Section 7.4 for the formal definition). All other servers are *ordinary* servers. Note that we have to deal with extraordinary servers carefully: we might have to pay much more than OPT for their monochromatic pieces that are scheduled on other servers.

Thus, to obtain a competitive algorithm, we need to minimize the number of extraordinary servers. We achieve this with the following idea: We equip our ILP with an objective function that minimizes the number of extraordinary servers and we show that the number of extraordinary servers created by the ILP gives a lower bound on the cost paid by OPT. We use this fact to argue that we can charge the algorithm's cost when creating extraordinary servers to OPT to obtain competitive results.

The previously described ideas provide a deterministic algorithm with competitive ratio $O(\ell \log k)$. We also provide a matching lower bound of $\Omega(\ell \log k)$. The lower bound provides a hard instance which essentially shows that an adversary can force any deterministic algorithm to make each of the ℓ servers extraordinary at some point in time. More generally, the adversary can cause such a high competitive ratio whenever it knows *which servers are extraordinary.* Hence, to obtain an algorithm with competitive ratio $O(\log \ell + \log k)$ we use randomization to keep the adversary from knowing the extraordinary servers.

Our strategy for picking the extraordinary servers randomly is as follows. First, we show that our algorithm experiences low cost (compared to OPT) when two pieces assigned to an ordinary server are merged, while the cost for merging pieces that are assigned to extraordinary servers is large (compared to OPT). Next, we reduce the problem of picking extraordinary servers to a paging problem, where the pages correspond to servers such that pages in the cache correspond to ordinary servers and pages outside the cache correspond to extraordinary servers. Now when two pieces are merged, we issue the corresponding page requests: A merge of pieces assigned to an ordinary server corresponds to a page request of a page which is inside the cache, while merging two pieces with at least one assigned to an extraordinary server corresponds to a page request of a page which is not stored in the cache. This leads the paging algorithm to insert and evict pages into and from the cache and our algorithm always changes the types of the servers accordingly. For example, when a page corresponding to an ordinary server is evicted from the cache, we make the corresponding server extraordinary. We conclude by showing that since the randomized paging algorithm of Blum et al. [37] allows for a polylogarithmic competitive ratio, we also obtain a polylogarithmic competitive ratio for our problem.

The chapter is organized as follows. Section 7.2 introduces our notation and Section 7.3 gives an overview of the algorithmic framework. We explain the deterministic algorithm in detail, including the ILP, in Section 7.4 and analyze it in Section 7.5. Section 7.6 presents the randomized algorithm. Our lower bounds are presented in Section 7.7 and Section 7.8 contains two omitted proofs.

7.1.2 Related Work

The online graph partitioning problem considered in this chapter is generally related to classic online problems such as competitive paging and caching [8, 66, 70, 139, 189, 212], k-server [71], or metrical task systems [41]. However, unlike these existing problems where requests are typically related to specific items (e.g., in paging) or locations in a graph or metric space (e.g., the k-server problem and metrical task systems), in our model, requests are related to *pairs of vertices*. The problem can hence also be seen as a *symmetric* version of online paging, where each of the two items (i.e., vertices in our model) involved in a request can be moved to either of the servers currently hosting one of the items (or even to a third server). The static problem variant is essentially a k-way partitioning or graph partitioning problem [1, 198]. The balanced graph partitioning problem is related to minimum bisection [68], and known to be hard to approximate [18, 120]. Balanced clustering problems have also been studied in streaming settings [15, 190].

Our model is also related to dynamic bin packing problems which allow for limited *repacking* [69]: this model can be seen as a variant of our problem where pieces (resp. items) can both be dynamically inserted and deleted, and it is also possible to open new servers (i.e., bins); the goal is to use only an (almost) minimal number of servers, and to minimize the number of piece (resp. item) moves. However, the techniques of [69] do not extend to our problem.

Another related problem arises in the context of generalized online scheduling, where the current server assignment can be changed whenever a new job arrives, subject to the constraint that the total size of moved jobs is bounded by some constant times the size of the arriving job. While the reconfiguration cost in this model is fairly different from ours, the sensitivity analysis of our ILP is inspired by the techniques used in Hochbaum and Shmoys [105] and Sanders et al. [182].

Our work is specifically motivated by the online balanced (re-)partitioning problem introduced by Avin et al. in [20, 23]. In their model, the connected components of the graph G can contain more than k vertices and, hence, might have to be split across multiple servers. They presented a lower bound of $\Omega(k)$ for deterministic algorithms. They complemented this result by a deterministic algorithm with competitive ratio of $O(k \log k)$. This problem was also studied when the graph G follows certain random graphs models [21, 22].

7.2 Preliminaries

Let us first re-introduce our model together with some definitions. We are given a graph $G = (V, E)$ with $|V| = \ell \cdot k$. In the beginning, $E = \emptyset$ and then edges are inserted in an online manner. Initially, every vertex v is assigned to one of ℓ servers such that each server is assigned exactly k vertices. We call this server the *source server* of v. For a server s we call the vertices which are initially assigned to s the *source vertices* of s. For normalization purposes, we consider each vertex to have

a *volume* $\mathrm{vol}(v)$ of $1/k$, so that the total volume of vertices initially assigned to a server is exactly 1.

After each edge insertion, the online algorithm must re-assign vertices to fulfill the *cc-condition*, i.e., so that all vertices of the same connected component of G are assigned to the same server. To this end, it can move vertices between servers at a cost of $1/k$ per vertex move. As described in the introduction, the optimum offline algorithm OPT is only allowed to place vertices with total volume up to 1 onto each server, while the online algorithm ONL is allowed to place total volume of total volume of $1+\epsilon$ on each server, where $\epsilon > 0$ is a small constant. For notational convenience, our algorithms will place a total volume of $1+c\cdot\epsilon$ for some constant c, which does not affect our asymptotic results as the algorithm can be started with $\epsilon' = \epsilon/c$.

Formally, the objective is to devise an online algorithm ONL which minimizes the *(strict) competitive ratio* ρ defined as $\rho = \mathit{cost}(\mathrm{ONL})/\mathit{cost}(\mathrm{OPT})$, where $\mathit{cost}(\cdot)$ denotes the total volume of pieces moved by the corresponding algorithm. For deterministic online algorithms, the edge insertion order is adversarial; for randomized online algorithms, we assume an oblivious adversary that fixes an adversarial request sequence without knowing the random choices of the online algorithm.

The following definitions and concepts are used in the remainder of this chapter.

Pieces. Our online algorithm proceeds by tracking the *pieces*, the connected components of G induced by the revealed edges. The *volume* of a piece p, denoted by $|p|$, is the total volume of all its vertices. For convenience, every server has a unique color from the set $\{1, \dots, \ell\}$ and every vertex has the *color* of the server it was *initially* assigned to. For a piece p, we define the *majority color* of p as the color that appears most frequently among vertices of p and, in case of ties, that is the smallest in the order of colors. We also refer to the corresponding server as the *majority server* of p. Similarly, we define the *majority color* for a vertex v to be the majority color of the piece of v. Note that the latter changes dynamically as the connected components of G change due to edge insertions.

Size Classes and Committed Volume. To minimize frequent and expensive moves, our approach groups the pieces into small and large pieces, and for the ILP also partitions them into a constant number of size classes. The basic idea is to "round down" the volume of a piece to a suitable multiple of $1/k$ and to call all pieces of zero rounded volume small. However, pieces can grow and, thus, change their size class, which in turn might create cost for the online algorithm. Thus, we need to use a more "refined" rounding, that gives us some control over when such a class change occurs.

More formally, let us assume that $1/4 > \epsilon \geq (10/k)^{1/4}$. We choose δ such that $\frac{1}{2}\epsilon^2 \leq \delta \leq \epsilon^2$ and $\delta = j\frac{1}{k}$ for some $j \in \mathbb{N}$. In addition, we assume $\lceil 1 \rceil_\delta - 1 \leq \delta/2$, where $\lceil\cdot\rceil_\delta$ is the operation of rounding up to the closest multiple of δ. Claim 7.38 in Section 7.8.1 shows that we can always find such a δ provided that $k \geq 10/\epsilon^4$. We will also use a constant $\gamma = 2\delta < 1$.

We partition the volume of a piece into *committed* and *uncommitted* volume. The committed volume will always be a multiple of δ, while the uncommitted volume

will be rather small (see below). We refer to the sum of committed and uncommitted volume as the *real volume* of the piece. We extend this definition to vertices: Each vertex is either committed or uncommitted. Now the committed volume of a piece is the volume of its committed vertices. For a piece p, we write $|p|_c$ to denote its committed volume and $|p|_u$ to denote its uncommitted volume. Hence, $|p| = |p|_c + |p|_u$.

We introduce size classes for the pieces. We say that a piece is in *class* $i \in \mathbb{N}$ if its *committed* volume is $i \cdot \delta$ (recall that committed volume is always a multiple of δ). Since the volume of a piece is never larger than 1, we have that $i \leq 1/\delta$ and thus there are only $O(1/\delta) = O(1/\varepsilon^2) = O(1)$ size classes in total.

Large and Small Pieces. Intuitively, we want to refer to pieces with total volume at least ϵ as *large* and to the remaining pieces as *small*. For technical reasons, we change this as follows. We say a piece is *large* if its committed volume is non-zero and *small* otherwise. This means that the small pieces are exactly the pieces in class 0. As the algorithm decides when to commit volume, the algorithm controls the transition from small to large. Note that committed volume never becomes uncommitted and, thus, a piece transitions only once from small to large.

Monochromatic Pieces. Pieces that overwhelmingly contain vertices of a single color have to be handled very carefully by an online algorithm because OPT may not have to move many vertices of such a piece and thus experience very little cost. Therefore we introduce the following notion, which needs to be different for small and large pieces since we use different scheduling techniques for them: A large piece is called *monochromatic* for its majority server s if the volume of its vertices that did not originate at s is at most δ. A small piece is called *monochromatic* if an ϵ-fraction of its volume did not originate at the majority server of the piece. We refer to pieces that are not monochromatic as *non-monochromatic*.

7.3 Algorithmic Framework

In this section we present our general algorithmic framework. Some further details follow in Section 7.4.

(1) The algorithm always maintains the current set of pieces $\mathcal{P}$, where each piece is annotated by its size class. If an edge insertion merges two pieces p_s and p_ℓ, into a new merged piece p_m, it holds that $|p_m|_u = |p_s|_u + |p_\ell|_u$ and $|p_m|_c = |p_s|_c + |p_\ell|_c$. We say that a merge is *monochromatic* if both p_s and p_ℓ are monochromatic for the same server s. Note that p_m is then also monochromatic for s. Throughout the rest of the chapter, we let p_m denote the piece that resulted from the *last merge-operation*.

Invariants for Piece Volumes. Whenever the algorithm has completed its vertex moves after an edge insertion, the following invariants for piece volumes are maintained.

1. A piece p is small (i.e. has $|p|_c = 0$) iff $|p| < \epsilon$.

2. A large piece has committed volume $i \cdot \delta$ for some $i \in \mathbb{N}, i > 0$. If it is monochromatic, all committed volume must be from its majority color.
3. The uncommitted volume of a large piece is at most 2δ, while the uncommitted volume of a small piece is at most ϵ.

Now suppose that before a merge-operation, all pieces fulfill the invariants. Then after the merge, the new piece p_m might fulfill only the relaxed constraint $|p_m| < 2\epsilon$ if p_m is small (no committed volume) and the relaxed constraint $|p_m|_u \leq \epsilon + 2\delta$ if p_m is large (with committed volume). Before the next merge-operation, our algorithm will perform *commit-operations* on the piece p_m until p_m fulfills the above invariants. More concretely, if $|p_m| \geq \epsilon$ then a commit-operation is executed as long as $|p_m|_u > 2\delta$. It selects uncommitted vertices inside p_m of volume δ and sets their state to committed (which makes p_m large). If p_m is monochromatic, the commit-operation only selects vertices of the majority color, of which there is a sufficient number since for large monochromatic pieces, the volume of vertices of non-majority color is at most δ.

(2) The algorithm further maintains a *schedule* S, which is an assignment of the pieces in $\mathcal{P}$ to servers. The algorithm guarantees that this schedule fulfills certain invariants—the most important being the fact that the total volume of pieces assigned to a server does not exceed the server's capacity by much.

Adjusting Schedules. To reestablish these invariants after a change to $\mathcal{P}$, we run the *adjust schedule subroutine*. We provide the details of this subroutine in Section 7.4 and now give a very short summary. When the set $\mathcal{P}$ changes, this is due to one of two reasons: a *merge operation* or a *commit-operation*. Both types of changes might force us to change the old schedule S to a new schedule S'. To do so, we first solve an ILP (to be defined in Section 7.4) that computes the rough structure of the new schedule S'. The ILP solution defines, among other things, the number of extraordinary servers in the new schedule S'. Then we determine a concrete schedule S' that conforms to the structure provided by the ILP solution. Crucially, we have to determine an S' that is not too different from S, in order to keep the cost for switching from S to S' small.

We note that the subroutine for adjusting the schedules only moves pieces between the servers and hence does not affect the invariants for piece volumes.

Handling an Edge Insertion. We now give a high-level overview of the algorithm when an edge (u, v) is inserted. If u and v are part of the same piece, we do nothing since the set of pieces $\mathcal{P}$ did not change. Otherwise, assume that u is in piece p_s and v is in p_ℓ with $p_s \neq p_\ell$ and $|p_s| \leq |p_\ell|$. Then we proceed as follows.

Step I *Move small to large piece:* Move the smaller piece p_s to the server of the larger piece p_ℓ.

Step II *Merge pieces:* Merge p_s and p_ℓ into p_m. Run the adjust schedule subroutine.

Step III *Commit volume:* If $|p_m| \geq \varepsilon$, then

while $|p_m|_u > 2\delta$:
Commit volume δ for p_m.
Run the adjust schedule subroutine.

7.4 Adjusting Schedules

Now we describe the subroutine for adjusting schedules in full detail. In the following, we define an ILP that helps in finding a good assignment of the pieces to servers. We ensure that when the set of pieces only changes slightly, then also the ILP solution only changes slightly. We also show how the ILP solution can be mapped to concrete schedules.

Before we describe the ILP in detail, we introduce reservation and source vectors, as well as configurations. In a nutshell, a server's reservation vector encodes how many pieces of each size class can be assigned to that server at most. A server's source vector, on the other hand, describes the structure of the monochromatic pieces for that server. A configuration is a pair of a reservation and a configuration vector and solving the ILP will inform us which configurations should be used for the servers in our schedule.

A *reservation vector* r_s for a server s has the following properties. For a size class $i > 0$, the entry r_{si} describes the total volume reserved on s for the (committed) volume of pieces in class i (regardless of their majority color). The entry r_{s0} describes the total volume that is reserved for uncommitted vertices (again, regardless of color); note that these uncommitted vertices could belong to small or large pieces. An entry $r_{si}, i > 0$, must be a multiple of $i\delta$ while the entry r_{s0} is a multiple of δ. Note that r_s does *not* describe which concrete pieces are scheduled on s and not even the exact number of pieces of a certain class, as it only "reserves" space.

A *source vector* m_s for server s has the following properties. For a size class $i > 0$, the entry m_{si} specifies the total committed volume of pieces in class i *that are monochromatic for* s. Again recall that a monochromatic piece only has committed volume of its majority color. The entry m_{s0} describes the total uncommitted volume of color s rounded up to a multiple of δ. Observe that similarly to the reservation vectors, (a) the entries m_{si} in the source vector are multiples of $i\delta$ and (b) the entry m_{s0} is a multiple of δ. In addition, (c) the entries in m_s sum up to at most $\lceil 1 \rceil_\delta$ as only vertices of color s contribute. Observe that the source vector of a server s just depends on the sizes of the s-monochromatic pieces and on which of their vertices are committed; *it does not depend on how an algorithm assigns the pieces to servers.*

A vector m is *a potential source vector* if it fulfills properties (a)-(c) without necessarily being the source vector for a particular server. Similarly, a *potential reservation vector* r is a vector where the i-th entry is a multiple of $i\delta$, the 0-th entry a multiple of δ, and r is γ-valid. Here, we say that r is γ-*valid* if $\|r\|_1 \leq 1 + \gamma$. Note that there are only $O(1)$ potential reservation or source vectors since they have only $O(1/\delta) = O(1)$ entries (one per size class) and for each entry there are only $O(1/\delta) = O(1)$ choices.

A *configuration* (r, m) is a pair consisting of a potential reservation vector r and a potential source vector m. We further call a configuration (r, m) *ordinary* if $r \geq m$ (i.e., $r_i \geq m_i$ for each i) and otherwise we call it *extraordinary*. The intuition is that servers with ordinary configurations have enough reserved space such that they can be assigned all of their monochromatic pieces. Next, note that

as there are only $O(1)$ potential source and reservation vectors, there are only $O(1)$ configurations in total.

Claim 7.1. *There exist only $O(1)$ different configurations (r, m).*

Proof. Note that both r and m are vectors with one entry for each size class. As argued in Section 7.2, there are only $O(1)$ different size classes. Furthermore, by definition of r and m each entry is a multiple of $\delta = O(1/\varepsilon^2)$ between 0 and $1 + \gamma$. Hence, there are only $O(1)$ choices for each entry of r and m. This proves the claim. □

In the following, we assign configurations to servers and we will call a server *ordinary* if its assigned configuration is ordinary and *extraordinary* if its assigned configuration is extraordinary.

We now define the ILP. Remember that the goal in this step is to obtain a set of configurations, which we will then assign to the servers and which will guide the assignment of the pieces to the servers. Thus, we introduce a variable $x_{(r,m)} \in \mathbb{N}_0$ for each (ordinary or extraordinary) configuration (r, m). After solving the ILP, our schedules will use exactly $x_{(r,m)}$ servers with configuration (r, m). Furthermore, the objective function of the ILP is set such that the number of extraordinary configurations is minimized.

The constraints of the ILP are picked as follows. First, let V_i, $i > 0$, denote the total committed volume of all pieces in class i and let V_0 denote the total uncommitted volume of all pieces. Note that the V_i do not depend on the schedule of the algorithm. Now we add a set of constraints, which ensures that the configurations picked by the ILP reserve enough space such that all pieces of class i can be assigned to one of the servers. Second, let Z_m denote the *number* of servers with the potential source vector m at this point in time. (Recall that the source vectors of the servers only depend on the current graph and the commitment decisions of the algorithm and *not* on the algorithm's schedule.) We add a second set of constraints which ensures that for each m, the ILP solution contains exactly Z_m configurations with source vector m. Now the ILP is as follows.

$$
\begin{aligned}
\min \quad & \textstyle\sum_{(r,m):\, r \not\geq m} x_{(r,m)} & & \\
\text{s.t.} \quad & \textstyle\sum_{(r,m)} x_{(r,m)} r_i/\delta & \geq V_i/\delta & \quad \text{for all } i \\
& \textstyle\sum_{r} x_{(r,m)} & = Z_m & \quad \text{for all } m
\end{aligned}
$$

In the ILP we wrote r_i/δ and V_i/δ to ensure that ILP only contains integral values. Further observe that the ILP has constant size and can, hence, be solved in constant time: As there are only $O(1)$ different configurations (Claim 7.1), the ILP only has $O(1)$ variables. Also, since there are only $O(1)$ size classes i and $O(1)$ source vectors m, there are only $O(1)$ constraints.

Next, we show that an optimal ILP solution serves as a lower bound on the cost paid by OPT.

Lemma 7.2. *Suppose the objective function value of the ILP is h, then we have that cost*$(\mathrm{OPT}) \geq (\gamma - \delta)h = \Omega(h)$.

Proof. Consider the solution of OPT, i.e., an assignment of the pieces to servers at minimum moving cost. We show that this assignment implies an ILP solution with small cost. In particular, for each server s for which OPT assigns at least a $(1-\gamma+\delta)$-fraction of its source vertices on s itself, we show that we can construct an ordinary configuration. For all other servers (for which OPT assigns at least a $(\gamma - \delta)$-fraction of their source vertices on other servers), we assign an extraordinary configuration. Note for each server which has obtained an extraordinary configuration in the previous construction, OPT has cost at least $\gamma - \delta$. Since the ILP minimizes the total number of extraordinary configurations, we get that *cost*$(\mathrm{OPT}) \geq (\gamma - \delta)h$.

We now construct the configurations for ordinary servers. Fix a server s for which OPT assigns at least a $(1 - \gamma + \delta)$-fraction of its source vertices on s itself. Let v be a vector such that v_i is the real volume of pieces in the i-th class, that OPT assigned to s. Note that the v_i are *not* rounded to multiples of δ and that $\|v\| = 1$. We construct a γ-valid reservation vector r for server s as follows.

For every piece p in class i that is assigned on s, we decrease v_i by the *uncommitted* volume of the piece and add this value to v_0. For a piece p of class i which is not assigned on s, we just consider the source vertices of s in p. We increase v_i by the committed volume of these vertices and v_0 by the uncommitted volume.

The above modifications did not increase $\|v\|_1$ if the piece p is assigned on s. The total increase of $\|v\|_1$ due to pieces not assigned on s can be at most $\gamma - \delta$ because each such increase is caused by an element with source-server s that is not assigned on s (and the total volume of such vertices is only $\gamma - \delta$). Now we round v_0 up to the nearest multiple of δ. This increases $\|v\|_1$ by at most δ. Therefore, $\|v\|_1 \leq 1 + \gamma$.

We use the final vector v as our reservation vector r. By construction r_0 is at least as large as the total uncommitted volume with source server s, i.e., $r_0 \geq m_{s0}$. Further, r_i is at least as large as the committed volume of monochromatic pieces with source s, i.e., $r_i \geq m_{si}$. Finally, the r_i value is still as large as the total committed volume of class i elements assigned on s. This means that the first set of constraints in the ILP still holds. □

7.4.1 Schedules That Respect an ILP Solution

Next, we describe the relationship of schedules and configurations. A *schedule* S is an assignment of pieces to servers. The set of pieces assigned to a particular server s is called the *schedule for* s. A schedule for a server s with source vector m_s *respects a reservation* r if the following holds:

1. The committed volume of class i pieces scheduled on s is at most r_i.
2. The total uncommitted volume scheduled on s is at most $r_0 + 14\epsilon$.
3. If $r \geq m_s$ then all pieces that are monochromatic for s are placed on s.

A schedule *respects an ILP solution* x if there exists an assignment of configurations to servers such that:

- A server s with source vector m_s is assigned a configuration (r, m) with $m = m_s$.
- A configuration (r, m) is used exactly $x_{(r,m)}$ times.
- The schedule of each server respects the reservation of its assigned configuration.

The next lemma shows that servers respecting a reservation only slightly exceed their capacities.

Lemma 7.3. *If the schedule for a server s respects a γ-valid reservation r, then the total volume of all pieces assigned to s is at most $1 + \gamma + 14\epsilon = 1 + O(\varepsilon)$.*

Proof. The committed volume in large pieces can be at most $1 + \gamma - r_0$ because $\|r\|_1 \leq 1 + \gamma$. The uncommitted volume scheduled on s can be at most $r_0 + 14\epsilon$ due to Property 2. □

7.4.2 How to Find Schedules

In this section, we describe how to resolve the ILP and adjust the existing schedule after a merge or commit-operation so that it respects the ILP solution, in particular, that the schedule of every server respects the reservation of its assigned configuration, i.e., Properties 1-3 above. It is crucial that this step can be performed at a small cost. We present different variants: In most situations, the algorithm uses a generic variant that is based on sensitivity analysis of ILPs. However, in some special cases (cases in which OPT might pay very little) using the generic variant might be too expensive. Therefore, we develop special variants for these cases that resolve the ILP and adjust the schedule at zero cost.

Before we describe our variants in detail, note that it is not clear how to assign small pieces to servers based on the ILP solution. Hence, we define our variants such that in the first phase they move some pieces around to construct a respecting schedule but they ignore Property 2 while doing so, i.e., they only guarantee Property 1 and Property 3. After this (preliminary) schedule has been constructed, we run a *balancing procedure* (described below), which ensures that Property 2 holds. The balancing procedure only moves small pieces and we show that its cost is at most the cost paid for the first phase. As it is relatively short, we describe it first.

Balancing Procedure for Small Pieces. We now describe our balancing procedure, which moves only small pieces and for which we show that Property 2 of respecting schedules is satisfied after it finished. The balancing procedure is run after one of the variants of the ILP solving is finished.

For a server s, let $v_u(s)$ denote the total uncommitted volume scheduled at s. We define the slack of a server s by $\mathrm{slack}(s) := r_{s0} - v_u(s)$. Note that because of the first constraint in the ILP with $i = 0$, there is always a server with non-negative slack. Next, we equip every server s with an *eviction budget* $budget(s)$ that is initially 0. Now, any operation outside of the balancing procedure that decreases

the slack must increase the eviction budget by the same amount. Such an operation could, e.g., be a piece p that is moved to s (which decreases the slack by $|p|_u$) or a decrease in r_{s0} when a new configuration is assigned to s. (Note that increasing the eviction budget increases the cost for the operation performing the increase; we will describe how we charge this cost later.) Intuitively it should follow that $budget(s)$ roughly equals $-\operatorname{slack}(s)$ and indeed we can show through a careful case analysis that $budget(s) \geq -\operatorname{slack}(s) - 2\epsilon$ (Claim 7.6).

This eviction budget is used to pay for the cost of moving small pieces away from s when the balancing procedure is called. We say a small piece p is *movable* if either its majority color has at most a $(1-2\epsilon)$-fraction of the volume of p (the piece is far from monochromatic), or its majority color corresponds to an extraordinary server. The balancing procedure does the following for each server s:

while there is a movable piece p on s with $|p| < budget(s)$:
move p to a server with currently non-negative slack
$budget(s) = budget(s) - |p|$

We show in the following lemma that after balancing procedure finished, we have that $\operatorname{slack}(s) \geq -14\varepsilon$. This implies that when the balancing procedure finished, Property 2 holds since $v_u(s) = r_{s0} - \operatorname{slack}(s) \leq r_{s0} + 14\varepsilon$.

Lemma 7.4. *After the balancing procedure for a server s finished, we have that* $\operatorname{slack}(s) \geq -14\epsilon$.

We prove the lemma in Subsection 7.4.2.5.

7.4.2.1 Overview of the Variants

Next, we give the full details of the main algorithm for adjusting the schedules in different cases.

Step I *Move small to large piece:* Move the smaller piece p_s to the server of the larger piece p_ℓ.

Step II *Merge pieces:* Merge p_s and p_ℓ into p_m. Then adjust the schedule as follows:
- If p_s is small, then the ILP does not change and no adjustment is necessary.
- Else: if the merge is non-monochromatic *or* s is extraordinary, use the Generic Variant, otherwise, use Special Variant A.

Step III *Commit volume:* If $|p_m| \geq \varepsilon$, then
while $|p_m|_u > 2\delta$:
Commit volume δ for p_m.
If p_m is non-monochromatic *or* s is extraordinary, use the Generic Variant, otherwise, use Special Variant B.

7.4.2.2 The Generic Variant

We now describe the Generic Variant of the schedule adjustment. Suppose the set of pieces $\mathcal{P}$ changed into $\mathcal{P}'$ due to a merge or a commit-operation.

The algorithm always maintains for the current set $\mathcal{P}$ an optimum ILP solution. Let x be the ILP solution for $\mathcal{P}$. When $\mathcal{P}$ changes, the algorithm runs the ILP to obtain the optimum ILP solution x' for $\mathcal{P}'$.

In the following, we first argue how to assign the configurations from x' to the servers and then we argue how we can transform a schedule S (respecting x) into a schedule S' (respecting x') with little cost.

We first assign the configurations given by x' to servers by the following greedy process. A configuration (r, m) is *free* if it has not yet been assigned to $x'_{(r,m)}$ servers. As long as there is a free configuration (r, m) and a server s that had been assigned (r, m) in schedule S, we assign (r, m) to s. The remaining configurations are assigned arbitrarily subject to the constraint that a server s with source vector m_s obtains a configuration of the form (r, m_s) for some r.

Now that we have assigned the configurations to the servers, we still have to ensure that the new schedule respects these new server configurations. We start with some definitions.

First, let $\mathcal{A}$ be the set of servers for which the set of scheduled pieces changed due to the merge or commit-operation. For a merge-operation, these are the servers that host one of the pieces p_s, p_ℓ, or p_m, and for a commit-operation, this is the server that hosts the piece p_m that executes the commit. Note that $|\mathcal{A}| \leq 3$. Second, let $\mathcal{B}$ be the set of servers that changed their source vector due to the merge or commit-operation. Note that for a commit-operation $|\mathcal{B}|$ could be large, because the committed volume could contain many different colors and for each corresponding server, the source vector could change by a reduction of m_0. Third, we let $\mathcal{C}$ be the set of servers that changed their assigned configuration between S and the current schedule S'. Note that $|\mathcal{C}| \leq |\mathcal{B}| + \|x - x'\|_1$.

Observe that for servers $s \in \overline{\mathcal{A} \cup \mathcal{C}}$ neither their assigned configuration (since $s \notin \mathcal{C}$) nor their set of scheduled pieces (since $s \notin \mathcal{A}$) has changed. Thus, these servers already respect their configuration and, hence, we do not move any pieces for these servers now. For the servers in $\mathcal{A} \cup \mathcal{C}$ we do the following:

1. We mark all pieces currently scheduled on servers in $\mathcal{A} \cup \mathcal{C}$ as *unassigned*.
2. Every ordinary server in $\mathcal{A} \cup \mathcal{C}$ moves all of its monochromatic pieces to itself. This guarantees Property 3 of a respecting schedule. Note that this step may move pieces away from servers in $\overline{\mathcal{A} \cup \mathcal{C}}$.
3. The remaining pieces are assigned in a first fit fashion. We say a server is *free* for class $i > 0$ if the *committed volume* of class i pieces already scheduled on it is (strictly) less than r_i. It is *free* for class 0 if the uncommitted volume scheduled on it is less than r_0.

   To schedule an unassigned piece p of class i, we determine a free server for class i and schedule p there. The first set of constraints in the ILP guarantees that we always find a free server.

This scheduling will guarantee Property 1 of a respecting schedule, i.e., for all $i > 0$ the volume of class i pieces scheduled on a server s is at most r_{si}. This holds because r_{si} is a multiple of $i\delta$. If we decide to schedule a class i piece on s because a server is free for class i then it actually has space at least $i\delta$ remaining for this class. Hence, we never overload class i, $i > 0$.

In the following, we develop a bound on the cost of the above scheme. For the analysis of our overall algorithm we use an involved amortization scheme. Therefore, the cost that we analyze here is not the real cost that is incurred by just moving pieces around but it is inflated in two ways:

(A) If we move a piece p to a server s, we increase the eviction budget of s by $|p|_u$.

(B) Whenever we change the configuration of a server from an ordinary configuration to an extraordinary configuration, we experience an *extra cost* of $4(1+\gamma)/\delta$. This will be required later in Case IIa of the analysis.

Observe that Cost Inflation (A) clearly only increases the cost by a constant factor. Cost Inflation (B) will also only increase the cost by a constant factor as the analysis below assumes constant cost for every server that changes its configuration. Note that the Generic Variant is the only variant for adjusting the schedule for which Inflation (B) has an affect; the other variants do not move pieces around and do not generate any new extraordinary configurations.

The following lemma provides the sensitivity analysis for the ILP. Its first point essentially states that for adjusting the schedules, we need to pay cost proportional to the number of servers that change their source configuration from $\mathcal{P}$ to $\mathcal{P}'$ plus the change in the ILP solutions. The second point then bounds the change in the ILP solutions by the number of servers that change their source vectors from $\mathcal{P}$ to $\mathcal{P}'$.

Lemma 7.5. *Suppose we are given a schedule S that respects an ILP solution x for a set of pieces $\mathcal{P}$. Let $\mathcal{P}'$ denote a set of pieces obtained from $\mathcal{P}$ by either a merge or a commit-operation, and let D denote the number of servers that have a different source vector in $\mathcal{P}$ and $\mathcal{P}'$. Then:*

1. *If x' is an ILP solution for $\mathcal{P}'$, then then we can transform S into S' with cost $O(1 + D + \|x - x'\|_1)$.*
2. *Then we can find an ILP solution x' for $\mathcal{P}'$ with $\|x - x'\|_1 = O(1 + D)$.*
3. *If the operation was a merge-operation, then $D \leq 3$.*

We prove the lemma in Subsection 7.4.2.6.

7.4.2.3 Special Variant A: Monochromatic Merge

Special Variant A is used if we performed a monochromatic merge-operation of two large pieces p_s, p_ℓ and if the server s that holds the piece p_ℓ is ordinary. Then OPT may not experience any cost. Therefore, we also want to resolve the ILP and adjust the schedule S with zero cost.

Since s has an ordinary configuration and p_ℓ is monochromatic for s, we know that p_ℓ was scheduled at s. Hence, our new piece p_m (which is generated at p_ℓ's server) is already located at the right server s.

We obtain our schedule S' by deleting the assignments for p_s and p_ℓ from S and adding the location s for the new piece p_m. Now let i_s, i_ℓ, and i_m denote the classes of pieces p_s, p_ℓ, and p_m, respectively (note that these classes are at least 1 as all pieces are large). Then the new ILP can be obtained by only changing the configuration vector m_s and setting

$$\begin{array}{rcl} m'_{si_s} &:=& m_{si_s} - |p_s|_c \\ m'_{si_\ell} &:=& m_{si_\ell} - |p_\ell|_c \\ m'_{si_m} &:=& m_{si_m} + |p_m|_c \end{array} \quad , \quad \begin{array}{rcl} Z'_{m_s} &:=& Z_{m_s} - 1 \\ Z'_{m'_s} &:=& Z_{m'_s} + 1 \end{array}$$

and

$$\begin{array}{rcl} V'_{i_s} &:=& V_{i_s} - |p_s|_c \\ V'_{i_\ell} &:=& V_{i_\ell} - |p_\ell|_c \\ V'_{i_m} &:=& V_{i_m} + |p_m|_c \end{array} .$$

To obtain a solution x' to this new ILP, we change the reservation vector for the server s as follows.

$$\begin{array}{rcl} r'_{si_s} &:=& r_{si_s} - |p_s|_c \\ r'_{si_\ell} &:=& r_{si_\ell} - |p_\ell|_c \\ r'_{si_m} &:=& r_{si_m} + |p_m|_c \end{array} .$$

This does not change the $\|\cdot\|_1$-norm of the vector r because $r_{i_s} \geq m_{i_s} \geq |p_s|_c$ (this follows from the definition of m_{i_s} and the fact that $r_s \geq m_s$ holds) and because $|p_s|_c + |p_\ell|_c = |p_m|_c$. We obtain the ILP solution x' by setting

$$x'_{(r_s, m_s)} := x_{(r_s, m_s)} - 1 \quad \text{and} \quad x'_{(r'_s, m'_s)} := x_{(r'_s, m'_s)} + 1.$$

Note that $r_s \geq m_s$ implies $r'_s \geq m'_s$. Hence, our new ILP solution does not increase the objective function value of the ILP (i.e., the number of extraordinary configurations). In Lemma 7.39 in Section 7.8.2 we show that merging two large monochromatic pieces of a server cannot decrease the objective function value of the ILP. Therefore, the new ILP solution x', which has the same objective function value as x, is optimal.

Finally, observe that we only changed the configuration of server s and that we did not move any pieces. Hence, we can transform $\mathcal{P}$, x and S into $\mathcal{P}'$, x' and S' with zero cost.

7.4.2.4 Special Variant B: Monochromatic Commit

Suppose we perform a commit-operation for a monochromatic piece p_m that is located at an ordinary server s. Then OPT may not experience any cost. Therefore,

we present a special variant for adjusting the schedule that also induces no cost. We perform a routine similar to Special Variant A. Recall that in a commit for a monochromatic piece, we commit volume exactly δ and all of the committed volume has color s.

Let i and i' denote the old and the new class of the piece, respectively. Then the source vector vector m_s of s changes as follows:

$$\begin{array}{lcll} m'_{si} & := & m_{si} & - & |p_m|_c \\ m'_{si'} & := & m_{si'} & + & |p_m|_c + \delta \\ m'_{s0} & := & m_{s0} & - & \delta \end{array} .$$

Note that the above is also correct for the case that $i = 0$ (p_m small) because then $|p_m|_c = 0$. The new ILP is obtained by setting

$$\begin{array}{lcll} Z'_{m_s} & := & Z_{m_s} & - & 1 \\ Z'_{m'_s} & := & Z_{m'_s} & + & 1 \end{array} \quad \text{and} \quad \begin{array}{lcll} V'_{si} & := & V_{si} & - & |p_m|_c \\ V'_{si'} & := & V_{si'} & + & |p_m|_c + \delta \\ V'_{s0} & := & V_{s0} & - & \delta \end{array} .$$

We obtain a solution to the new ILP by adjusting the reservation vector r at the server s where p_m is scheduled (recall that s is ordinary, i.e., the monochromatic piece p_m must be located at s):

$$\begin{array}{lcll} r'_{si} & := & r_{si} & - & |p_m|_c \\ r'_{si'} & := & r_{si'} & + & |p_m|_c + \delta \\ r'_{s0} & := & r_{s0} & - & \delta \end{array} .$$

Observe that we reduce r_{s0} by δ. Usually, if we reduce r_{s0} we increase the eviction budget of s, so that s can evict small pieces in case the uncommitted volume scheduled on s is larger than $r_{s0} + 14\epsilon$. However, here this is not necessary because the commit also decreases the uncommitted volume that is scheduled on s by δ.

Observe that $\|r_s\| \leq 1 + \gamma$ implies $\|r'_s\| \leq 1 + \gamma$, i.e., r' is γ-valid. The new solution x' is

$$\begin{array}{lcll} x'_{(r_s, m_s)} & := & x_{(r_s, m_s)} & - & 1 \\ x'_{(r'_s, m'_s)} & := & x_{(r'_s, m'_s)} & + & 1 \end{array} .$$

Note that the configuration (r'_s, m'_s) is ordinary because (r_s, m_s) is ordinary. Overall only a single server changed its configuration and this server keeps an ordinary configuration. Therefore the number of extraordinary configurations did not increase and we still have an optimum solution to the ILP. See Lemma 7.40 in Section 7.8.2 for a formal proof that the new solution is indeed optimal.

Also the old schedule still respects this new ILP solution. Therefore we do not experience any cost.

7.4.2.5 Proof of Lemma 7.4

We want to show that $\mathrm{slack}(s) = r_{s0} - v_u(s) \geq -14\epsilon$. We prove the lemma by contradiction, i.e., assume that $\mathrm{slack}(s) < -14\epsilon$. We will use the following claim, which we prove at the end of the subsection.

Claim 7.6. $budget(s) \geq v_u(s) - r_{s0} - 2\epsilon = -\operatorname{slack}(s) - 2\varepsilon$.

First, we argue that the total volume of small pieces is $r_{s0} + 4\varepsilon$. Let $\bar{m}_{s0}$ denote the total uncommitted volume for s *not* rounded up, i.e., $m_{s0} = \lceil \bar{m}_{s0} \rceil_\delta$. Since $\operatorname{slack}(s) < -14\varepsilon$, the above claim gives that the budget is at least $12\epsilon \geq 2\epsilon$, i.e., it is larger than the volume of small pieces. Thus, the only reason to not perform an eviction is that all small pieces scheduled on s are close to being monochromatic for s and s is ordinary. But the total volume of such pieces can be at most

$$\bar{m}_{s0}/(1-2\epsilon) \leq (1+4\epsilon)\bar{m}_{s0} \leq \bar{m}_{s0} + 4\epsilon \leq r_{s0} + 4\epsilon$$

, where used $\epsilon \leq 1/4$ and that s has an ordinary configuration.

Second, the following claim gives a bound of 10ϵ on the total uncommitted volume in large pieces; we prove the claim at the end of this subsection.

Claim 7.7. *If $\gamma \leq 1$, then the total uncommitted volume in large pieces at a server s is at most 10ϵ.*

We conclude that Property 2 holds, i.e., $v_u(s) \leq r_{s0} + 14\epsilon$. However, this is a contradiction to our assumption that $r_{s0} - v_u(s) = \operatorname{slack}(s) < -14\varepsilon$ since this inequality implies $v_u(s) > r_{s0} + 14\varepsilon$.

Proof of Claim 7.6. In the initial state of the algorithm $budget(s) = 0$, $v_u(s) = 1$, $m_{s0} = r_{s0} = \lceil 1 \rceil_\delta$. Thus, the statement holds. The following operations affect the slack:

- A piece p is moved to s outside of the balancing procedure. This increases both sides of the equation by $|p_u|$.
- A small piece p is moved to s inside of the balancing procedure. This is only performed if the slack on s is non-negative. Note that the move causes the slack to increase by at most $|p| \leq 2\epsilon$. Hence the equation holds afterwards.
- A piece is moved away from s outside of the balancing procedure. This only decreases $v_u(s)$.
- A small piece is moved away from s inside the balancing rouine. This decreases both sides of the equation by the volume of the piece.
- When adjusting the schedule by the generic routine (see Secction 7.4.2.2) and a new configuration with a smaller r_{s0}-value is assigned to s, then the eviction budget is increased by the change in r_{s0}. Hence, both sides of the equation increase by the same amount.

As no operation can make the equation invalid it holds throughout the algorithm.

Note that when we use Special Variant A for adjusting the schedule (see Section 7.4.2.3) r_{s0} will be decreased but the committed volume scheduled on s will be decreased by the same amount. This means the slack does not change in this case. □

Proof of Claim 7.7. Consider the set L of large pieces that exclude the piece p_m that resulted from the last merge-operation. Let $\bar{v}_c$ and $\bar{v}_u$ denote the total committed

and uncommitted volume for pieces in L that are scheduled on s. Pieces in L have volume at least ϵ and uncommitted volume at most 2δ (Invariants 1 and 3). Therefore, the factor $f := \bar{v}_u/(\bar{v}_c+\bar{v}_u)$ between uncommitted and real volume of these pieces is at most $2\delta/\epsilon$. We can derive a bound on the total uncommitted volume for pieces in L as follows:

$$\bar{v}_c + \bar{v}_u = (1 + \tfrac{f}{1-f})\bar{v}_c \leq (1+2f)\bar{v}_c \ ,$$

where we use $f \leq 2\delta/\epsilon \leq 1/2$, which holds because $\delta \leq \epsilon^2$ and $\epsilon \leq 1/4$. To obtain a bound on the total uncommitted volume in large pieces, we have to also consider the piece p_m. For this piece we have $|p_m|_u \leq \epsilon + 2\delta$ according to Invariant 3. We get

$$v_u = \bar{v}_u + |p_m|_u \leq 2f\bar{v}_c + (\epsilon + 2\delta) \leq \tfrac{4\delta}{\epsilon}(1+\gamma) + 2\epsilon \leq 10\epsilon \ ,$$

where the second step uses that $|p_m|_u \leq \epsilon + 2\delta$ due to Invariant 3, the third step uses that the committed volume at s is at most $\|r\|_1 \leq 1+\gamma$, and the final step uses $\gamma \leq 1$. □

7.4.2.6 Proof of Lemma 7.5

The third claim holds since only the servers for which one of the pieces p_s, p_ℓ, or p_m is monochromatic can change their source vector.

Next, consider the first claim. The cost for the second step in the above scheme is at most $O(|\mathcal{A} \cup \mathcal{C}|)$ because only pieces that are monochromatic for a server in $\mathcal{A} \cup \mathcal{C}$ move. The cost for the third step is also at most $O(|\mathcal{A} \cup \mathcal{C}|)$ because only pieces that were scheduled on servers from $\mathcal{A} \cup \mathcal{C}$ in S move. This gives the first claim since

$$|\mathcal{A} \cup \mathcal{C}| \leq 3 + |\mathcal{B}| + \|x - x'\|_1 = O(1 + D + \|x - x'\|_1).$$

The rest of the proof is devoted to the proof of the second claim. We use the following general result about the sensitivity of optimal ILP solutions. It states that a small change in the constraint vector of the ILP implies only a small change in the optimal solution x.

Theorem 7.8 ([185, Corollary 17.2a]). *Let A be an integral $n_r \times n_c$ matrix, such that each subdeterminant of A is at most Δ in absolute value; let b' and b'' be column n_r-vectors, and let c be a row n_c-vector. Suppose $\max\{cx \mid Ax \leq b'; x \text{ integral}\}$ and $\max\{cx \mid Ax \leq b''; x \text{ integral}\}$ are finite. Then for each optimum solution z' of the first maximum there exists an optimum solution z'' of the second maximum such that $\|z' - z''\|_\infty \leq n_c\Delta(\|b' - b''\|_\infty + 2)$.*

To apply the theorem, we bound how much the constraint vector in our ILP changes. Every change in the value of Z_m represents a change in the source configuration of some server. Hence, $\|Z - Z'\| \leq D$. Every change in a value V_i represents a change in the committed volume of a piece. For a merge operation

there are at most 3 pieces that change their committed volume (pieces p_s, p_ℓ, pm in a merge-operation). For a commit-operation only the piece p_m executing the commit changes its committed volume. Hence, $\|V - V'\|_1 \leq 3/\delta$. Overall the RHS vector in the ILP changes by $O(1 + D)$.

Let A denote the matrix that defines the ILP. Then the number of columns n_c is the number of configurations (r, m). This is constant due to Claim 7.1.

An entry in the matrix A is either 0, 1 or $r_i/\delta \leq \|r\|_1/\delta \leq (1 + \gamma)/\delta$. Hence, $a_{\max} := (1 + \gamma)/\delta$ is an upper bound for the absolute value of entries in A. As the number of columns is n_c, we can use Hadamard's inequality to get a bound of $\Delta \leq n_c^{n_c/2} a_{\max}^{n_c} = O(1)$ on the value of any subdeterminant.

Now, Theorem 7.8 gives that we can find an optimum ILP solution x' for $\mathcal{P}'$ with $\|x - x'\|_\infty \leq O(1 + \|b - b'\|_\infty)$. As the vectors x and x' have a constant number of entries, we also get $\|x - x'\|_1 \leq O(1 + \|b - b'\|_1) = O(1 + D)$, as desired.

7.5 Analysis

We first give a high level overview of the analysis. Let $\mathcal{P}^*$ denote the final set of pieces. A simple lower bound on the cost of OPT is as follows. Let NM denote the set of vertices that do not have the majority color within their piece in $\mathcal{P}^*$. Then *cost*$(\mathrm{OPT}) \geq \frac{1}{k}|\mathrm{NM}|$, because each vertex has volume $\frac{1}{k}$ and for each piece in $\mathcal{P}^*$, OPT has to move all vertices apart from vertices of a single color. Hence, the total volume of pieces moved by OPT is at least $\frac{1}{k}|\mathrm{NM}|$.

We want to exploit this lower bound by a charging argument. The general idea is that whenever our online algorithm experiences some cost C, we *charge* this cost to vertices whose color does not match the majority color of their piece. If the total charge made to each such vertex v is at most $\alpha \cdot \mathrm{vol}(v)$, then the cost of the online algorithm is at most $\alpha \cdot$ *cost*(OPT). When we charge cost to vertices, we will refer to this as *vertex charges*.

The difficulty with this approach is that at the time of the charge, we do not know whether a vertex will have the majority color of its piece in the end. Therefore, we proceed as follows. Suppose we have a subset S of vertices in a piece p and a subset $Q \subseteq S$ does not have the current majority color of S . Then *regardless of the final majority color of* p, a total volume of $\mathrm{vol}(Q)$ of vertices in S will not have this color in the end. Hence, when we distribute a charge of C evenly among the vertices of S, a charge of $\mathrm{vol}(Q) \cdot C/\mathrm{vol}(S)$ goes to vertices that do not have the final majority color. We call this portion of the charge *successful*.

The following lemma shows that to obtain algorithms competitive to OPT, it suffices if we bound the successful and the total vertex charges.

Lemma 7.9. *Suppose the total successful charge is at least* $charge_{\mathrm{succ}}$ *while the maximum (successful and unsuccessful) charge to a vertex is at most* $charge_{\max}$*. Then* *cost*$(\mathrm{OPT}) \geq \frac{1}{k} charge_{\mathrm{succ}}/charge_{\max}$.

Proof. Note that successful charge only goes to vertices in NM. Hence, $|\mathrm{NM}| \geq \mathit{charge}_{\mathrm{succ}}/\mathit{charge}_{\max}$, and, therefore, we obtain that $\mathit{cost}(\mathrm{OPT}) \geq \frac{1}{k}|\mathrm{NM}| \geq \frac{1}{k}\mathit{charge}_{\mathrm{succ}}/\mathit{charge}_{\max}$. □

Another lower bound that we use is due to Lemma 7.2. Let $h_{\max}$ denote the maximum objective value obtained when solving different ILP instances during the algorithm. From time to time, when vertex charges are not appropriate, we perform *extraordinary charges* or just *extra charges*. In the end, we compare the total extra charge to $h_{\max}$. We stress that we only perform extra charges when extraordinary configurations are involved. This means if $h_{\max} = 0$ we never perform extra charges, as otherwise, it would be difficult to obtain a good competitive ratio.

In the following analysis, we go through the different steps of the algorithm. For every step, we charge the cost either by a vertex charge or by an extra charge. If we apply a vertex charge, we argue that (1) enough of the applied charge is successful and (2) the charge can accumulate to not too much at every vertex. For extra charges, we require a more global argument and we will derive a bound on the total extra charge in terms of $h_{\max}$ in Section 7.5.1.1.

7.5.1 Analysis Details

When merging a piece p_ℓ and p_s with $|p_s| \leq |p_\ell|$ we proceed in several steps.

Step I: Small to Large. In this first step, we move the vertices of p_s to the server of p_ℓ. If p_s and p_ℓ are on different servers we experience a cost of $|p_s|$. Also, we have to increase the eviction budget of the server that holds piece p_ℓ (if p_s is a small piece). The cost for this step is 0 if p_s and p_ℓ are on the same server and, otherwise, it is at most $2|p_s|$. We charge the cost as follows.

Case (Ia) Merge is monochromatic. If p_s, p_ℓ, and p_m are monochromatic for the same server s we only experience cost if s is extraordinary because otherwise p_s and p_ℓ are located at s. We make an extra charge for this cost.

Case (Ib) Merge is not monochromatic. We make the following vertex charges:

- Type I charge: We charge $\frac{2}{\delta} \cdot \frac{|p_s|}{|p_m|} \cdot \mathrm{vol}(v)$ to every vertex in p_m.
- Type II charge: We charge $\frac{2}{\delta} \cdot \mathrm{vol}(v)$ to every vertex in p_s.

Claim 7.10 below shows that the Type I and Type II charge at a vertex can accumulate to at most $O(\log k)$. In the following, we argue that at least a charge of $2|p_s|$ is successful. We distinguish several cases.

- If either p_ℓ or p_m is not monochromatic, we know that at least a volume of δ (if the non-monochromatic piece is large) or a volume of $\epsilon|p_\ell|$ of vertices does not have the majority color. Hence, we get that at least $\min\{\delta, \epsilon|p_\ell|\}\frac{2|p_s|}{\delta|p_m|} \geq 2|p_s|$ of the Type I charge is successful. The inequality uses $|p_m| \leq 1$, $|p_\ell| \geq \frac{1}{2}|p_m|$, and $\delta \leq \epsilon^2 \leq \epsilon/2$.
- If p_s is not monochromatic then at least $\delta|p_s|$ volume in p_s has not the majority color. This gives a successful charge of at least $\delta|p_s| \cdot \frac{2}{\delta} \geq 2|p_s|$.

- Finally suppose that p_s and p_ℓ are monochromatic for different colors C_s and C_ℓ, respectively. If in the end C_s is not the majority color of the final piece then we have a successful charge of at least $(1-\epsilon)|p_s| \cdot 2/\delta \geq 2|p_s|$ from the Type II charge. Otherwise, C_ℓ is not the majority color and we obtain a successful charge of $(1-\epsilon)|p_\ell| \cdot \frac{2|p_s|}{\delta|p_m|} \geq 2|p_s|$.

Claim 7.10. *The combined Type I and Type II charge that can accumulate at a vertex v is at most $O(\log k \cdot \mathrm{vol}(v)/\delta)$.*

Proof. A vertex v only experiences a Type II charge if the piece that it is contained in just increased its volume by at least a factor of 2. This can happen at most $\log k$ times and therefore the total Type II charge at a vertex is at most $O(\log k \cdot \mathrm{vol}(v)/\delta)$.

Fix a vertex v and define $a_i := |p_s|$ and $b_i := |p_m|$ at the time of the i-th Type I charge to vertex v. Then the total Type I charge to v is

$$\frac{2}{\delta}\,\mathrm{vol}(v)\sum_{i\geq 1}\frac{a_i}{b_i}\ .$$

To estimate $\sum_i \frac{a_i}{b_i}$, we use the fact that $b_i \geq b_{i-1}+a_i$ and that each a_i is a multiple of $1/k$. We define $A_i := \sum_{j=1}^{i} a_j$. This gives

$$\sum_{i\geq 1}\frac{a_i}{b_i} \leq \sum_{i\geq 1}\frac{a_i}{A_i} = \sum_{i\geq 1}\sum_{j=1}^{ka_i}\frac{1}{kA_i} \leq \sum_{i\geq 1}\sum_{j=1}^{ka_i}\frac{1}{kA_i-j} = \sum_{j=1}^{kA_t-1}\frac{1}{j}\ ,$$

where t denotes the total number of charges to vertex v. Since, A_t is at most 1 we get that the sum is $O(\log k)$. This gives a total Type II charge of at most $O(\log k \cdot \mathrm{vol}(v)/\delta)$. □

Step II: Resolve ILP and Adjust Schedule. In this step, we merge the pieces p_s and p_ℓ into p_m and run the subprocedure for adjusting the schedule, which finds a new optimum solution to the ILP and finds a schedule respecting the ILP solution. Due to Lemma 7.5 this incurs at most constant cost. In the following, we distinguish several cases. For some cases, the bound of Lemma 7.5 is sufficient and we only have to show how to properly charge the cost. For other cases, we give a better bound than the general statement of Lemma 7.5. In the following, s denotes the server where the merged piece p_m is located now (and where p_ℓ was located before).

Case (IIa) p_s small. In this case, the input to the ILP did not change. This holds because no volume was committed and no uncommitted volume changed between classes. Therefore we do not experience any cost for resolving the ILP.

However, it may happen that p_ℓ was not monochromatic but the merged piece p_m is. Suppose p_m is monochromatic for a server $s' \neq s$, and this server has an ordinary configuration. Then we have to move p_m to s' for the new schedule to respect the configuration of s'. We incur a cost of $|p_m| + |p_m|_u \leq 2|p_m|$, where

$|p_m|_u$ is required to increase the eviction budget at s'. We charge $4/\delta \cdot \mathrm{vol}(v)$ to every vertex in p_m. We call this charge a Type III charge.

How much of the charge is successful? Observe that p_ℓ was not monochromatic for s' before the merge as otherwise it would have been located at s'. This means vertices with volume at least $\delta|p_\ell| \geq \delta|p_m|/2$ in p_m have a color different from s' (the majority color in p_m). This means we get a successful charge of at least $\delta|p_m|/2 \cdot 4/\delta = 2|p_m|$, as desired.

To obtain a good bound on the total Type III charge accumulating at a vertex v we have to add a little tweak. Whenever a server s switches its configuration from ordinary to extraordinary, we cancel the most recent Type III charge operation for all vertices currently scheduled on s.

This negative charge is accounted for in the *extra cost* that we pay when switching the configuration of a server from ordinary to extraordinary. Recall that in Cost Inflation (B), we said that we experience an extra cost of $4(1+\gamma)/\delta$ whenever we switch the configuration of a server s from ordinary to extraordinary. This cost is used to cancel the most recent Type III charge for all pieces currently scheduled on s.

Lemma 7.11. *Suppose a vertex v experiences a positive Type III charge at time t that is not canceled. Let t' denote the time step of the next Type III charge for vertex v, and let p and p' denote the pieces that contain v at times t and t', respectively. Then $|p'| \geq (1+\epsilon)|p|$.*

Proof. Let p be monochromatic for s and p' be monochromatic for s', i.e., the Type III charges occur because we have to move p to s and p' to s'. We distinguish two cases.

First assume that $s \neq s'$ and let $v(s)$ denote the volume of vertices of color s in p'. Then

$$(1-\epsilon)|p| \leq v(s) \leq \epsilon|p'| \ ,$$

where the first inequality follows because p is monochromatic for s and the second because p' is monochromatic for s'. We get that $|p'| \geq (1-\epsilon)/\epsilon \cdot |p| \geq (1+\epsilon)|p|$, where the last inequality holds for $\epsilon \leq \sqrt{2}-1$, which holds as $\epsilon \leq 1/4$.

Now, assume that $s = s'$. This means that p was moved to server s, subsequently a piece $p'' \supset p$ was moved away from s, and in the end $p' \supset p''$ was moved back to s.

The server s cannot be extraordinary at time t as then there would be no need to move p to s and no Type III charge would occur. Also, s cannot become extraordinary between time t and t' because then the Type III charge at t would be canceled. Hence, s is ordinary.

The only reason for moving p'' away from s is one of the following:

- The eviction routine does it. As s is ordinary, this routine only moves p'' if at most a $(1-2\epsilon)$-fraction of its vertices have color s. Let $\bar{v}(s)$ denote the

volume of vertices in p'' that have a color different from s. Then

$$2\epsilon|p''| \leq \bar{v}(s) \leq \epsilon|p| + |p''| - |p| \ ,$$

because at most a volume of $\epsilon|p|$ did not have color s at time t and after that at most a volume of $|p''| - |p|$ has been added. This gives $|p'| \geq |p''| \geq (1-\epsilon)/(1-2\epsilon) \cdot |p| \geq (1+\epsilon)|p|$.

- p'' is moved just before being merged with a (larger) piece p_ℓ. Let p_m denote the piece obtained by merging p'' with p_ℓ. Then $|p| \leq |p''| \leq |p_m|/2 \leq |p'|/2$, which gives $|p'| \geq 2|p|$.
- p'' is a large piece and we determine a new location for it when resolving the ILP and adjusting the schedule. However, then we only move it away if it is not monochromatic for s. But this is a contradiction to the fact that p' is monochromatic for s at time t' because large pieces cannot become monochromatic by merging them with other pieces.

□

Corollary 7.12. *The total Type III charge that can accumulate at a vertex is only* $O(\log k \cdot \mathrm{vol}(v))$.

Proof. Since, between any two uncanceled Type III charges to a vertex v the volume of the piece that v is contained in must grow by a factor of $1+\epsilon$, there can be at most $O(\log k)$ such charges, each charging $4/\delta$. □

Case (IIb) p_s large, merge not monochromatic. We resolve the ILP and adjust the schedule S. According to Item 1 and Item 3 of Lemma 7.5 this incurs constant cost. Let C_{IV} denote the bound on this cost. We perform a vertex charge of $C_{\mathrm{IV}}/\delta \cdot \mathrm{vol}(v)$ for every vertex in p_m. We call this charge a Type IV charge. In the following we argue that at least a charge of C_{IV} is successful. We distinguish two cases.

If one of the pieces p_s, p_ℓ, or p_m is not monochromatic we know that at least vertices of volume δ in the piece do not have the majority color. Hence, we get that at least $C_{\mathrm{IV}}/\delta \cdot \delta \geq C_{\mathrm{IV}}$ of the Type IV charge is successful.

Now, suppose that p_s is monochromatic for server s and p_m is monochromatic for a different server s'. Regardless of which color is the majority color in the end, there will be vertices of volume at least $(1-\epsilon)|p_s|$ that will not have this majority color. Hence, we obtain a successful charge of at least $(1-\epsilon)|p_s| \cdot C_{\mathrm{IV}}/\delta \geq (1-\epsilon)\epsilon \cdot C_{\mathrm{IV}}/\delta \geq C_{\mathrm{IV}}$, where the first step uses that p_s is large and the second that $\delta \leq \epsilon^2 \leq (1-\epsilon)\epsilon$, which holds because $\epsilon \leq 1/4$.

Claim 7.13. *A vertex v can accumulate a total Type IV charge of at most* $C_{\mathrm{IV}}/\delta \cdot \mathrm{vol}(v)$.

Proof. Whenever we perform a Type IV charge for a vertex v, the piece that v is contained in just increased its volume by $|p_s| \geq \delta$. This can happen at most $1/\delta$ times. □

Case (IIc) p_s large, merge monochromatic, s extraordinary. In this case, we also resolve the ILP and adjust the schedule, which according to Item 1 and Item 3 of Lemma 7.5 incurs constant cost. Let C denote this cost. We make an extra charge of C. Observe that $C = O(|p_s|)$ because p_s is a large piece. This will be important when we derive a bound on the total extra charge.

Case (IId) p_s large, merge monochromatic, s ordinary. Suppose that the server s has an ordinary configuration. In this case we do not want to have any cost, because we cannot perform an extra charge as no extraordinary configurations are involved and we cannot charge against the vertices of p_m as the piece is monochromatic. We use Special Variant A for adjusting the schedule. This induces zero cost.

Step III: Commit-operation. We analyze the commit-operation. We will call a commit-operation monochromatic if it is performed on a monochromatic piece and, otherwise, we call it non-monochromatic.

Case (IIIa) p_m not monochromatic, s ordinary. The commit-operation may change the source vector of several servers. Let D denote the number of servers that changed their source vector. The cost for handling the commit-operation is at most $O(1 + D)$ according to Lemma 7.5. Let C_{V} denote the hidden constant, i.e., the cost is at most $C_{\mathrm{V}}(1 + D)$. We split this cost into two parts: C_{V} is the *fixed cost* and $C_{\mathrm{V}}D$ is the *variable cost* of the commit.

We charge $3C_{\mathrm{V}}/\delta \cdot \mathrm{vol}(v)$ to every vertex v in p_m. We call this charge a Type V charge. In p_m at least vertices of volume δ have not the majority color because p_m is not monochromatic. Therefore we get a successful charge of $3C_{\mathrm{V}}/\delta \cdot \delta = 3C_{\mathrm{V}}$.

Clearly, the charge is sufficient for the fixed cost. However, the remaining successful charge of $2C_{\mathrm{V}}$ may not be sufficient for the variable cost. In the following, we argue that the total remaining successful charge that is performed *for all* non-monochromatic commits is enough to cover the variable cost for these commits.

Lemma 7.14. *Let $X_{\mathrm{nm}}(s)$ denote the number of times that a non-monochromatic commit causes a change in the source vector of s. Then the variable cost for all non-monochromatic commits is at most $\sum_s C_{\mathrm{V}} X_{\mathrm{nm}}(s) \leq 2C_{\mathrm{V}}N$, where N denotes the total number of non-monochromatic commits.*

Proof. We analyze $X_{\mathrm{nm}}(s)$ for a fixed server s. Observe that a non-monochromatic commit can only change the entry m_{s0} in a source vector as the other entries concern volume of monochromatic pieces, which does not change due to the commit. Let v_s denote the uncommitted volume of server s, i.e., $m_{s0} = \lceil v_s \rceil_\delta$ where $\lceil \cdot \rceil_\delta$ denotes the operation of rounding up to a multiple of δ. Let $\xi_i(s)$ denote the reduction in v_s caused by the i-th commit-operation (monochromatic or non-monochromatic).

Then the total reduction in v_s throughout the algorithm is exactly $\sum_i \xi_i(s)$. Observe that in the beginning of the algorithm $v_s = 1$. Furthermore, by choice of δ we have that $\lceil 1 \rceil_\delta - 1 \leq \delta/2$ which is equivalent to $1 - \lceil 1 - \delta \rceil_\delta \geq \delta/2$. This ensures

that the first change in m_{s0} can occur only after v_s decreased by at least $\delta/2$. Every other change occurs after v_s decreased by an additional value of δ. Hence, if at least one change in m_{s0} occurs, we have

$$\delta(X_{\mathrm{m}}(s) + X_{\mathrm{nm}}(s) - 1) + \delta/2 \leq \sum_i \xi_i(s) = \sum_{i \in I_{\mathrm{m}}} \xi_i(s) + \sum_{i \in I_{\mathrm{nm}}} \xi_i(s) \ ,$$

where I_{m} and I_{nm} denote the index set of monochromatic and non-monochromatic commits, respectively, and $X_{\mathrm{m}}(s)$ denotes the number of times that a monochromatic commit causes a change in the source vector of s. For a monochromatic commit $\xi_i(s)$ is either 0 or δ. This gives that

$$\delta X_{\mathrm{nm}}(s) - \delta/2 \leq \sum_{i \in I_{\mathrm{nm}}} \xi_i(s) \ . \tag{7.1}$$

For $X_{\mathrm{nm}}(s) \geq 1$ we obtain

$$X_{\mathrm{nm}}(s) \leq 2X_{\mathrm{nm}}(s) - 1 \leq 2 \sum_{i \in I_{\mathrm{nm}}} \xi_i(s)/\delta \ ,$$

by multiplying Equation 7.1 with $2/\delta$. Next, observe that $\sum_s \xi_i(s) = \delta$ since every non-monochromatic commit switches a volume of exactly δ from uncommitted to committed. Hence, summing over all servers gives

$$\sum_{s : X_{\mathrm{nm}}(s) \geq 1} X_{\mathrm{nm}}(s) \leq 2 \sum_s \sum_{i \in I_{\mathrm{nm}}} \xi_i(s)/\delta = 2 \sum_{i \in I_{\mathrm{nm}}} \sum_s \xi_i(s)/\delta = 2N \ . \quad \square$$

Observe that the total remaining charge for the non-monochromatic commits is $2C_{\mathrm{V}}N$ (a charge of $2C_{\mathrm{V}}$ for every commit). Hence, the previous lemma implies that this remaining charge is sufficient for the variable cost of all non-monochromatic commits.

Claim 7.15. *The Type V charge at a vertex v can accumulate to at most $3C_{\mathrm{V}}/\delta^2 \cdot \mathrm{vol}(v)$.*

Proof. A vertex v can participate in at most $1/\delta$ commit-operations as each such operation increases the committed volume of the piece that v is contained in by δ and the committed volume of a piece can be at most 1. For each monochromatic commit-operation the vertex is charged $3C_{\mathrm{V}}/\delta \cdot \mathrm{vol}(v)$. This gives the lemma. $\square$

Case (IIIb) p_m monochromatic, s ordinary. Suppose we perform a commit-operation for the piece p_m. Here we use Special Variant B for resolving the ILP and adjusting the schedule. This incurs zero cost.

Case (IIIc) p_m monochromatic, s extraordinary. We resolve the ILP and adjust the schedule. The cost for this is $O(1)$, since we can use Item 1 of Lemma 7.5 with $D = 1$, because p_m is monochromatic and thus we only commit volume of color s. Let C_1 denote the upper bound for this cost. We perform an extra charge of C_1.

Since the committed volume has only color s, the total number of monochromatic commits for a specific server s is at most $1/\delta = O(1)$ because each commit increases the committed volume of color s by δ. Consequently, the total extra charge that we perform for monochromatic commits of a specific sever s is at most C_1/δ. To simplify the analysis of the total extra charge in Section 7.5.1.1 we combine all these extra charges into one extra charge of C_1/δ that is performed whenever the server s switches its state from ordinary to extraordinary *for the first time*.

7.5.1.1 Analysis of Extra Charges

In this section we derive a bound on the total extra charge generated by our charging scheme. Let us first recap when we perform extra charges:

(I) During the merge-operation we perform an extra charge of $O(|p_s|)$ in Case Ia and Case IIc, when the merge-operation is monochromatic for server s and s has an extraordinary configuration.
We stress the fact that whether a merge is monochromatic only depends on the sequence of merges and not on the way that pieces are scheduled by our algorithm.

(II) Whenever a server changes its configuration from ordinary to extraordinary *for the first time*, we generate an extra charge of $C_1/\delta = O(1)$ to take care of the cost of monochromatic commits (Case IIIc).

Now let $h_{\max}$ denote the maximum number of extraordinary configurations that are used throughout the algorithm. Clearly, if $h_{\max} = 0$ there is never any extraordinary configuration and the extra charge will be zero. If $h_{\max} \geq 1$, we show that the previously described deterministic online algorithm guarantees an extra charge of at most $O(\ell \log k)$.

Lemma 7.16. *If $h_{\max} = 0$, there is no extra charge. If $h_{\max} \geq 1$, the total extra charge is $O(\ell \log k)$.*

Before we prove Lemma 7.16, we prove the following claim which bounds the total extra charge for a single server.

Claim 7.17. *The total extra charge for a server s is at most $O(\log k)$.*

Proof. Suppose whenever we perform an extra charge of Type (I) for server s, we place this charge on the vertices in the smaller piece p_s such that each such vertex v receives a charge of $O(1) \cdot \mathrm{vol}(v)$. Then the charge that can accumulate at a vertex can be at most $O(\log k) \cdot \mathrm{vol}(v)$, since a vertex can be on the smaller side of a merge-operation at most $O(\log k)$ times. Since the volume of all vertices originating on s is 1 we obtain $O(\log k)$ for the extra charge of Type (I).

Clearly the extra charge of Type (II) can be at most $O(1)$ for any server. □

Proof of Lemma 7.16. For $h_{\max} = 0$, there are no extraordinary servers and thus we do not make any extra charges. For $h_{\max} \geq 1$, the lemma follows from Claim 7.17 since there are only ℓ servers. □

Next, we show that the maximum vertex charge (successful or unsuccessful) is $O(\log k \cdot \mathrm{vol}(v))$.

Lemma 7.18. *The maximum vertex charge $charge_{\max}$ (successful or unsuccessful) that a vertex v can receive is at most $O(\log k \cdot \mathrm{vol}(v))$.*

Proof. We have the following vertex charges:

- Type I charge and Type II charge:
  Accumulates to at most $O(\log(k)/\delta) \cdot \mathrm{vol}(v)$ according to Claim 7.10.
- Type III charge:
  Accumulates to at most $O(\log k) \cdot \mathrm{vol}(v)$ according to Corollary 7.12.
- Type IV charge:
  Accumulates to at most $C_{\mathrm{IV}}/\delta \cdot \mathrm{vol}(v)$ according to Claim 7.13.
- Type V charge:
  Accumulates to at most $3C_{\mathrm{V}}/\delta^2 \cdot \mathrm{vol}(v)$ according to Claim 7.15.

This gives the lemma. □

Combining Lemma 7.16 and Lemma 7.2 for extra charges and our arguments about vertex charges with Lemma 7.9, we obtain the following theorem.

Theorem 7.19. *There exists a deterministic online algorithm with competitive ratio $O(\ell \log k)$.*

Proof. All the cost is either charged by successful vertex charges or by extra charges. Therefore, the cost of the online algorithm is at most the total successful vertex charge plus the extra charge. Lemma 7.16 together with the observation that no extra charge is performed if $h_{\max} = 0$ gives that the total extra charge is at most $O(\ell \log k \cdot h_{\max}) = O(\ell \log k) \cdot \mathit{cost}(\mathrm{OPT})$.

Lemma 7.18 shows that the maximum vertex charge $charge_{\max}$ (successful or unsuccessful) made to a vertex v is at most $O(\log k) \cdot \mathrm{vol}(v) = O(\log k) \cdot 1/k$. From Lemma 7.9 we obtain

$$charge_{\mathrm{succ}} \leq (k \cdot charge_{\max}) \cdot \mathit{cost}(\mathrm{OPT}) = O(\log k) \cdot \mathit{cost}(\mathrm{OPT}).$$

Hence, we can bound the total extra charge and the total successful vertex charge by $O(\ell \log k) \cdot \mathit{cost}(\mathrm{OPT})$. This gives a competitive ratio of $O(\ell \log k)$. □

Note that we obtain an even stronger result if $h_{\max} = 0$: the cost is at most $O(\log k) \cdot \mathit{cost}(\mathrm{OPT})$ because of the bound on the total vertex charge (and the fact that we do not have extra charges). Otherwise ($h_{\max} > 0$), the total extra charge is at most $O(\ell \log k)$, which means that we are $O((\ell \log k)/h_{\max})$-competitive. So the worst-case competitive ratio occurs when $h_{\max} = 1$.

7.6 Randomized Algorithm

How can randomization help to improve on the competitive ratio? For this observe that the cost that we charge to vertices is at most $O(\log k \cdot \mathit{cost}(\mathrm{OPT}))$. Hence, the critical part is the cost for which we perform extra charges, which can be as large as $\Omega(\ell \log k)$ according to Theorem 7.28. A rough sketch of a (simplified) lower bound is as follows. We generate a scenario where initially all servers have the same source vector but some server needs to schedule its source-pieces on different servers (as, otherwise, we could not fulfill all constraints).

In this situation, an adversary can issue merge requests for all vertices that originated at the server s that currently has its source-pieces distributed among several servers. Then the online algorithm incurs constant cost to reassemble these pieces on one server, and, in addition, has to split the source-pieces of another server between at least two servers. Repeating this for $\ell - 1$ steps gives a cost of $\Omega(\ell)$ to the online algorithm while an optimum algorithm just incurs constant cost.

The key insight for randomized algorithms is that the above scenario cannot happen if we randomize the decision of which server distributes its source-pieces among several servers. The online problem then turns into a paging problem and we use results from online paging to derive our bounds.

7.6.1 Augmented ILP

Let M denote the set of all potential source vectors. We introduce a partial ordering on M as follows. We say $m \geq_p m'$ if any prefix-sum of m is at least as large as the corresponding prefix-sum for m'. Formally,

$$m \geq_p m' \iff \forall i\colon \sum_{j=0}^{i} m_j \geq \sum_{j=0}^{i} m'_j \ .$$

Observe that $m \geq m'$ implies $m \geq_p m'$. We adapt the ILP by adding a cost-vector c that favors large source vectors w.r.t. $\geq_p$. This means as a first objective the ILP tries to minimize the number of extraordinary configurations as before but as a tie-breaker it favors extraordinary configurations with large source vectors. For this we assign unique ids from $1, \dots, |M|$ to the source vectors s.t. $m_1 \geq_p m_2 \implies \mathrm{id}(m_1) \leq \mathrm{id}(m_2)$. Then we define the cost-vector c by setting

$$c_{(r,m)} := \begin{cases} 0 & r \geq m \\ 1 + \lambda\,\mathrm{id}(m) & \text{otherwise} \end{cases} , \tag{7.2}$$

for $\lambda = 1/(|M|^2 \cdot \ell)$. Given the cost-vector c, we set the objective function of our new ILP to $\sum_{(r,m)} c_{(r,m)} x_{(r,m)}$. The choice of λ together with $\|x\|_1 = \ell$ imply that $\sum_{(r,m):r\not\geq m} \lambda\,\mathrm{id}(m) x_{(r,m)} \leq \lambda \cdot |M| \ell = 1/|M| < 1$. Thus, the ILP still minimizes the number of extraordinary servers.

Note that the sensitivity analysis for the ILP still holds (Theorem 7.8 is independent of the cost vector and also Lemmas 7.39 and 7.40 hold for the cost vector defined above). This means if we have a constant change in the RHS vector of the

ILP, we can adjust the ILP solution and the schedule at the cost stated in Lemma 7.5. Similarly, when we manually adjust the ILP solution (Case IId and Case IIIb), we do not increase the cost because only the configuration of a single server s changes and this server keeps its ordinary configuration, i.e., it does not contribute to the objective function of the ILP.

A crucial property of the partial order $\geq_p$ is that source vectors of servers are monotonically decreasing w.r.t. $\geq_p$ as time progresses and as more merge-operations are processed.

Observation 7.20. *Let $m_s(t)$ denote the source vector of some server s after some timestep t of the algorithm. Then $t_1 \leq t_2$ implies $m_s(t_1) \geq_p m_s(t_2)$, i.e., the source vector of a particular server is monotonically decreasing w.r.t. $\geq_p$.*

Proof. m_s changes because of the following operations:

- A monochromatic commit on a piece p in class i executes $m'_{s0} := m_{s0} - \delta$, $m'_{si} := m_{si} - |p|_u$, and $m'_{s,i+1} := m_{s,i+1} + |p|_u + \delta$.
- A non-monochromatic commit may decrease m_{s0} but does not increase any entry.
- A monochromatic merge executes $m'_{si_s} := m_{si_s} - |p_s|_u$, $m'_{si_\ell} := m_{si_\ell} - |p_\ell|_u$, and $m'_{si_m} := m_{si_m} + |p_m|_u$.
- If the piece p_m is not monochromatic after a merge (and p_m is large) then entries in m_s are only reduced.
- A merge between small pieces does not change m_s (only the following commit may do so).

This means operations either reduce entries in m_s or they shift the mass of m_s to higher entries while not changing $\|m_s\|$. This gives the observation. □

7.6.2 Marking Scheme

The total extra charge that is generated by our algorithm is determined by how we assign extraordinary configurations to servers. We use a marking scheme to decide which servers *may* receive an extraordinary configuration. Formally, a (randomized) *marking scheme* dynamically partitions the servers into *marked* and *unmarked* servers and satisfies the following properties:

- Initially, i.e., before the start of the algorithm, all servers are unmarked.
- Let h_m denote the number of servers with source vector m that are assigned an extraordinary configuration by the ILP, i.e., $h_m = \sum_{(r,m):r\not\geq m} x_{(r,m)}$. The marking scheme has to mark at least h_m servers with source vector m.

The cost $\mathrm{cost}(\mathcal{M})$ of a marking scheme $\mathcal{M}$ is defined as follows:

- Switching the state of a server from marked to unmarked or vice versa induces a cost of 1.
- If a marked server experiences a monochromatic merge, the cost increases by $|p_s|$, where p_s is the smaller piece involved in the merge-operation.

Suppose for a moment that the marked servers always are exactly the servers that are assigned an extraordinary configuration. Then the above cost is clearly an upper

bound on the total extra charge as define in Section 7.5.1.1 (up to constant factors). This is because the marking scheme pays whenever switching between marked and unmarked, while in our analysis we only make one extra charge of constant cost when a server switches to an extraordinary configuration for the first time.

In the following, we enforce the condition that a server only has an extraordinary configuration if it is marked by the marking scheme. However, the marking scheme could mark additional servers that are not extraordinary. Thus, by enforcing this condition our algorithm incurs additional cost. Suppose, e.g., that the marking scheme decides to unmark a server s that is currently marked and has been assigned an extraordinary configuration. Then we have to switch the (extraordinary) configuration (r, m_s) assigned to s with an ordinary configuration (r', m_s) that currently is assigned to a different marked server s'. Note that we always find such a server because there exist at least h_{m_s} marked servers with source vector m_s. The switch can then be performed at constant cost. We make an additional extra charge for this increased cost of our algorithm. Note that the marking scheme accounts for this additional cost as it incurs cost whenever the state of a server changes. Therefore, the cost of the marking scheme can indeed serve as an upper bound on the total extra charge (including the additional extra charge). This gives the following observation.

Observation 7.21. *Let $\mathcal{M}$ be a marking scheme. The total extra charge is at most $O(\mathrm{cost}(\mathcal{M}))$.*

Next, we construct a marking scheme with small cost. For simplicity of exposition we assume that we know $h_{\max}$, the maximum number of extraordinary configurations that will be used throughout the algorithm, in advance. We describe in Section 7.6.3 how to adjust the scheme to work without this assumption by using a simple doubling trick (i.e., make a guess for $h_{\max}$ and increase the guess by a factor of 2 if it turns out to be wrong).

We will use results from online paging. In the online paging problem[1] [37] a sequence of page requests has to be served with a cache of size z. A request (p, w) consists of a page p from a set of $\ell \geq z$ pages together with a weight $w \leq 1$. If the requested page is in the cache, the cost for an algorithm serving the request sequence is 0. Otherwise, an online algorithm experiences a cost of w. It can then decide to put the page into the cache (usually triggering the eviction of another page) at an additional cost of 1.

The cost metric for the optimal offline algorithm is different and provides an advantage to the offline algorithm. If the offline algorithm does not have p in its cache, it pays a cost of w/r, where $r \geq 1$ being a parameter of the model, and then it can decide to put p into its cache at an additional cost of 1. In [37], the authors show how to obtain a competitive ratio of $O(r + \log z)$ in this model.

[1]Note that our problem definition slightly differs from the model analyzed by Blum et al. [37], which has $w = 1$ for every request. However, it is straightforward to show that the results of [37] carry over to our model.

The Paging Problems. Let M denote the set of potential source vectors and recall that $|M| = O(1)$. We introduce $|M|$ different paging problems, one for every potential source vector $m \in M$.

Fix a potential source vector m. Let S_m denote the set of servers that have a source vector $m' \geq_p m$. Essentially, we simulate a paging algorithm on the set S_m (i.e., servers correspond to pages) with a cache of size $|S_m| - h_{\max}$ and parameter $r = \log k$.

Note that a server may leave the set S_m, but it is not possible for a server to enter this set because the source vector m_s of a server is non-increasing w.r.t. $\geq_p$ (Observation 7.20). The fact that servers may leave S_m is problematic for setting up our paging problem because this would correspond to decreasing the cache size, which is usually not possible. Therefore, we define the paging problem on the set of *all servers* and we set the cache size to $\ell - h_{\max}$, but we make sure that servers/pages not in S_m are always in the cache. This effectively reduces the set of pages to S_m and the cache size to $|S_m| - h_{\max}$.

We construct the request sequence of the paging problem for S_m as follows. A monochromatic merge for a server $s \in S_m$ is translated into a page request for page s with weight $|p_s|$, where p_s is the smaller piece that participates in the merge-operation. Following such a merge request, we issue a page request (with weight 1) for every page/server not in S_m. This makes sure that an optimum solution keeps all these pages in the cache at all times, thus reducing the effective cache-size to $|S_m| - h_{\max}$. The request sequence stops when $|S_m| = h_{\max}$.

The Marking Scheme. We obtain a marking scheme from all the different paging algorithms as follows. A server with source vector m is marked if it is *not* in the cache for the paging problem on set S_m, or if $|S_m| \leq h_{\max}$. The following lemma shows that this gives a valid marking scheme and we prove it in Section 7.6.4.

Lemma 7.22. *The marking scheme marks at least h_m servers with source vector m.*

Let $\mathrm{cost}(S_m)$ denote the cost of the solution to the paging problem for S_m. The following two claims give an upper bound on the cost of the marking scheme.

Claim 7.23. *We have that*

$$\begin{aligned}\mathrm{cost}(\mathcal{M}) &\leq \sum_m \left(\mathrm{cost}(S_m) + h_{\max} + O(\log k) \cdot h_{\max} \right) \\ &= O\left(\sum_m \mathrm{cost}(S_m) + \log k \cdot h_{\max} \right).\end{aligned}$$

Proof. Initially, we have to choose for every paging problem $h_{\max}$ servers/pages that are not in the cache. The marking scheme marks these servers and experiences a cost of 1 for each marking. This gives a total cost of $|M| h_{\max}$ for the initialization.

As long as $|S_m| > h_{\max}$, a monochromatic merge-operation on a server with source vector m introduces a request of weight $|p_s|$. If the marking scheme has to pay for the merge (because the corresponding server s is extraordinary) then

the page s is outside of the cache in the paging problem for S_m and the paging problem has to pay for the request. The total extra charge for monochromatic merge-operations that occur after $|S_m| \leq h_{\max}$ can be at most $O(\log k) \cdot h_{\max}$ because the total extra charge for a specific server s is at most $O(\log k)$ due to Claim 7.17. □

Claim 7.24. *There is a randomized online algorithm for the paging problem on S_m with (expected) cost $\operatorname{cost}(S_m) \leq O((\log k + \log \ell) \cdot h_{\max})$.*

Proof. Note that an offline paging algorithm for the constructed request sequence on S_m can simply determine the $h_{\max}$ elements that leave the set S_m last and put all other servers into the cache. It then experiences a cost of at most $h_{\max} + O(\log k)/r \cdot h_{\max} = O(h_{\max})$, where the first $h_{\max}$-term is due to the initialization. The second term comes from the fact that the total weight of all requests to a specific page is equal (up to constant factors) to the total extra charge for a specific server. The latter is at most $O(\log k)$ due to Claim 7.17. Hence, by not moving any page after the initialization OPT pays $O(\log k)/r \cdot h_{\max}$ for serving the page requests.

Since $r = \log k$ and since the online algorithm of Blum et al. is $O(r + \log \ell)$-competitive, we obtain $\operatorname{cost}(S_m) \leq O(\log k + \log \ell) \cdot h_{\max}$. □

Now combining the two claims above with Lemma 7.2 and the analysis of vertex charges from Section 7.5, we obtain our main theorem.

Theorem 7.25. *There is a randomized algorithm with competitive ratio $O(\log \ell + \log k)$.*

Proof. Analyzing the cost for the vertex charges is identical to the deterministic case. Combining Observation 7.21 with Claim 7.23 and Claim 7.24 gives that the expected total extra charge is only $O(\log \ell + \log k) \cdot h_{\max}$. As $\operatorname{cost}(\mathrm{OPT}) = \Omega(h_{\max})$ the theorem follows. □

7.6.3 Marking Scheme Without Knowledge of $h_{\max}$

In Section 7.6.2 we showed how we can construct a marking scheme when $h_{\max}$ is known in advance. We now argue how this assumption can be dropped using a simple doubling trick.

In particular, when the algorithm starts and no edges were revealed, we set $h_{\max} = 0$. After that, when the object function of ILP is at least 1 for the first time, we set $h_{\max} = 1$ and run the marking scheme with fixed $h_{\max}$ from Section 7.6.2. After that, whenever the objective function of the ILP is larger than $h_{\max}$, we double the value of $h_{\max}$ and restart the marking scheme from Section 7.6.2.

It remains to analyze the cost of this scheme. First, let $h_{\max}^{\text{final}}$ denote the final value of $h_{\max}$ used by the above procedure when the algorithm stops and let $h_{\max}^{*}$ denote the highest objective function value of the ILP at any point in time. Now observe that $h_{\max}^{\text{final}} \leq 2h_{\max}^{*}$. Second, note that for fixed $h_{\max}$, Claim 7.23 and

Claim 7.24 imply that $\mathrm{cost}(\mathcal{M}) \leq O((\log k + \log \ell)h_{\max})$. Third, observe that the first two points imply that the total cost paid by the above procedure is

$$\begin{aligned}\sum_{i=0}^{\log h_{\max}^{\text{final}}} O((\log k + \log \ell)2^i) &= O(\log k + \log \ell) \sum_{i=0}^{\log h_{\max}^{\text{final}}} 2^i \\ &= O((\log k + \log \ell) \cdot h_{\max}^{\text{final}}) \\ &= O((\log k + \log \ell) \cdot h_{\max}^{*}).\end{aligned}$$

Hence, the cost paid by the above procedure which does not know $h_{\max}^*$ in advance is asymptotically the same as that of the procedure which knows $h_{\max}^*$ in advance.

7.6.4 Proof of Lemma 7.22

Before we prove the lemma, we first need to prove another claim and another lemma.

Claim 7.26. *Suppose we have an ILP solution x with $x_{(r,m)} > 0$ for a configuration (r, m) with $r \not\geq m$ but $r \geq_p m$. Then we can reduce the cost of x by at least $1 + \lambda\,\mathrm{id}(m)$.*

Proof. We change the extraordinary configuration (r, m) into an ordinary configuration without creating additional extraordinary configurations (i.e., for all $m' \neq m$: $\sum_{r:r\not\geq m'} x_{(r,m')}$ will stay the same). We change entries of r in a step by step process starting with the highest coordinate. Suppose that we already have $r_i \geq m_i$ for $i > j$ and that still $r \geq_p m$. Assume that $r_j = m_j - \xi$. Then we set $r_j^{\text{new}} := m_j$ and $r_{j-1}^{\text{new}} := r_{j-1} - \xi$ as the new entries for coordinate j and $j-1$, respectively. Note that $\|r\|_1$ does not change. However, the ILP now has

$$\sum\nolimits_{(\bar{r},\bar{m})} x_{(\bar{r},\bar{m})}\bar{r}_{j-1} \geq V_{j-1} - \xi \quad \text{and} \quad \sum\nolimits_{(\bar{r},\bar{m})} x_{(\bar{r},\bar{m})}\bar{r}_{j} \geq V_{j} + \xi \ . \tag{7.3}$$

Since $\sum_{(\bar{r},\bar{m})} x_{(\bar{r},\bar{m})}\bar{m}_j \leq V_j$ there must exist a configuration (r', m') such that $x_{(r',m')} > 0$ and $r'_j > m'_j$. We choose such a configuration, decrease r'_j and increase r'_{j-1} by the same amount. This does not change $\|r'\|_1$ and we can choose the increment so that $r'_j \geq m'_j$ still holds. Repeating this process can fix the constraints in Eq. (7.3) without generating new extraordinary configurations.

Fixing all coordinates in r results in an ordinary configuration for the source vector m and this will reduce the cost of the ILP by $1 + \lambda\,\mathrm{id}(m)$. □

Lemma 7.27. *Suppose the optimal ILP solution uses an extraordinary configuration for some source vector m, i.e., $x_{(r,m)} > 0$ with $r \not\geq m$. Then it does not use any ordinary configurations of the form (r', m'), $m' \geq_p m$, i.e., $\sum_{r\geq m'} x_{(r,m')} = 0$.*

Proof. Assume for contradiction that the lemma does not hold. Let $m' \geq_p m$, where m and m' are source vectors with reservation r and r', respectively. Further assume that $r \not\geq m$ and $r' \geq m'$.

We switch the reservation vector between the configurations, i.e., we increase $x_{(r',m)}$ and $x_{(r,m')}$ and decrease $x_{(r,m)}$ and $x_{(r',m')}$.

If $r' \geq m$ then this step decreased the cost of the ILP because we decreased the number of extraordinary configurations of the form $(\cdot, m)$ and we (may) have increased the number of extraordinary configurations of the form $(\cdot, m')$. This decreases the overall cost and contradicts that x is an optimal ILP solution.

Now assume $r' \not\geq m$. We may have increased the cost of the ILP solution by $1 + \lambda \operatorname{id}(m')$. However, $r' \geq m' \geq_p m$ implies $r' \geq_p m$. This means that we can reduce the cost of this solution by $1 + \lambda \operatorname{id}(m)$ due to Claim 7.26. Altogether the cost decreases because $\operatorname{id}(m) > \operatorname{id}(m')$. This contradicts the fact that x is an optimal ILP solution. □

Now we prove Lemma 7.22. First, suppose that $|S_m| \leq h_{\max}$. Then all servers with source vector m are marked and the lemma clearly holds. Otherwise, let $X_m := \{s \mid s \in S_m; m_s \neq m\}$. Since the paging problem for S_m must leave at least $h_{\max}$ pages from S_m outside the cache, there are at least $h_{\max} - |X_m|$ of these that have source vector m. The ILP has at most $h_{\max}$ extraordinary configurations. If at least one server with source vector m is extraordinary (i.e., $h_m > 0$) then all servers in X_m are assigned an extraordinary configuration due to Lemma 7.27. Hence, $h_m \leq h_{\max} - |X_m|$ and the lemma follows.

7.7 Lower Bounds

In this section, we derive lower bounds on the competitive ratios for deterministic and randomized algorithms. In particular, we show that any deterministic algorithm must have a competitive ratio of $\Omega(\ell \log k)$ and any randomized algorithm must have a competitive ratio of $\Omega(\log \ell + \log k)$.

We note that the lower bounds derived in this section also apply to the model studied by Henzinger et al. [101]. Their model is slightly more restrictive than ours in that eventually, every server must have exactly one piece of volume 1 (resp. k in their terminology); in contrast, in our model, servers may eventually host multiple pieces smaller than 1. However, our lower bounds are designed such that they also fulfill the definition of the model by Henzinger et al.

7.7.1 Lower Bounds for Deterministic Algorithms

Theorem 7.28. *For any $k \geq 32$ and any constant $1/k \leq \varepsilon \leq 1/32$ such that εk is a power of 2, any deterministic algorithm must have a competitive ratio of $\Omega(\ell \log k)$.*

We devote the rest of this subsection to prove the theorem.

Set m be a positive integer such that $\varepsilon k = 2^m$. As $\varepsilon \leq 1/32$ it follows that $k \geq 2^{m+5}$. Fix any deterministic algorithm ONL. We will show that there exists a sequence σ_{ONL} of edge insertions such that the cost of the optimum offline algorithm is $O(\varepsilon)$, while the cost of ONL is $\Omega(\varepsilon \ell \log(\varepsilon k))$. The sequence σ_{ONL} depends

on ONL, i.e., edge insertions will depend on which servers ONL decides to place the pieces.

Definitions. We assume that the servers are numbered sequentially. As before, each server has a color and every vertex is colored with the color of its initial server. For simplicity, we assume server i has color i. The *main server* of a color c is the server that, out of all servers, currently contains the largest volume of color-c vertices *and* whose index number of all such servers is the smallest[2].

A piece is called *single-colored* if all vertices of the piece have the same color. If a single-colored piece with color c is not assigned to the main server for c, it is called c-*away* or simply *away*. Any piece of volume at least 2ε is called a *large* piece, all other pieces are called *small*. We say *two pieces are merged* if there is an edge insertion connecting the two pieces.

Initial Configuration. Initially each of the ℓ servers contains one large single-colored piece of volume 2ε and $(1-2\varepsilon)k$ isolated vertices, each of volume $1/k$. The large pieces of on servers 1, 2, and 3 are called *special*. A color c is *deficient* if the total volume of all *small* c-away pieces is at least ε.

Sequence σ_{ONL}. The first two edge insertions merge the three special pieces into one (multi-colored) special piece of volume 6ε. As we will show *any* algorithm now has at least one deficient color. Note that all small pieces are single-colored and have volume $2^0/k = 1/k$.

Now σ_{ONL} proceeds in *rounds*. We will show that there is a deficient color at the end and, thus, also at the beginning of every round. In each round only small pieces of the same (deficient) color are merged such that their volume doubles. As a result, all small pieces continue to be single-colored and, at the end of each round, all small pieces of the same color have the same volume, namely $2^i/k$ for some integer i, *except* for potentially one piece of smaller volume, which we call the *leftover piece*. A leftover piece is created if the number of small items of color c and volume $2^i/k$ at the beginning of a round is an odd number. If this happens, it is merged with the leftover piece of color c of the previous rounds (if it exists) to guarantee that there is always just one leftover piece of color c. To simplify the notation we will use the term *almost all small pieces of color* c to denote all pieces of color c except the leftover piece of color c.

A round of σ_{ONL} consists of the following sequence of requests among the small pieces: If there exists a deficient color c such that the volume of almost all small pieces of color c is $2^i/k$ for some integer i and $2^i/k < \varepsilon$, then σ_{ONL} contains the following steps.
If there are small pieces of color c and volume $2^i/k$ that are currently on different servers, they are connected by an edge, otherwise two such pieces on the same server are connected by an edge. Repeat this until there is at most one small piece of color c of volume $2^i/k$ left. Once this happens and if such a piece exists, it becomes a *leftover* piece of color c and if another leftover piece of color c exists from earlier

[2]The difference between *majority* and *main* color is that we added the second condition to guarantee that the main server of a color is unique.)

rounds, the two are merged. Note that almost all pieces of color c have now volume $2^{i+1}/k$ and the leftover piece has smaller volume. If $2^{i+1}/k \geq \varepsilon$, merge all *non-special* (i.e. the small and the non-special large) pieces of color c and call color c *finished*. As long as there are at least two unfinished deficient colors, start a new round.

Once there are no more rounds we will show that there is exactly one unfinished deficient color c^* left and there are at least 2^{j+3} small pieces of color c^* and volume $\varepsilon/2^j$ for some integer $j \geq 1$. Furthermore there exists the special piece of volume 6ε (which is not single-colored) and for every other color there exists one piece of volume 1 (if it does not belong to $\{1,2,3\}$) or of volume $1-2\varepsilon$ (if it belongs to $\{1,2,3\}$).

Final Merging Steps: To guarantee that each piece has volume exactly 1 at the end, the remaining pieces of volume less than 1 are now suitably merged. First 2^{j+3} of the pieces of color c^* and volume $\varepsilon/2^j$ are merged into 3 pieces of volume 2ε each, the rest is merged into one piece. Then consider two cases: If $c^* \in \{1,2,3\}$, let c' and c'' be the other two colors of $\{1,2,3\}$. In this case σ_{ONL} merges the first small piece of volume 2ε of c^* with the non-special piece of c' and then merges the second small piece of volume 2ε of c^* with the non-special piece of c''. Then all the remaining pieces of color c^* are merged with each other and with the special piece.

If $c^* \notin \{1,2,3\}$, then σ_{ONL} merges the small pieces of volume $\varepsilon/2$ of color c^* with the non-special piece of color 1 and then does the same with color 2 and 3. Then all the remaining (small and large) pieces of color c^* are merged with each other and with the special piece.

Note that as a consequence all piece now have volume 1.

We show first that all the assumptions made in the description of σ_{ONL} hold. Specifically the next three lemmata will show that (1) after initialization and after each round there exists a deficient color for any algorithm, that (2) for each color c almost all small pieces of color c have volume $2^i/k$ and the leftover piece of color c has volume less than $2^i/k$, and that (3) at the beginning of the final merging steps there is exactly one unfinished deficient color left and there are at least 2^{j+3} small pieces of this color that have volume $\varepsilon/2^j$ for some integer $j \geq 1$. Then we will show that algorithm ONL has cost at least $\Omega(\varepsilon\ell\log(\varepsilon k))$ to process the sequence.

Lemma 7.29. *At the beginning of each round there exists an unfinished deficient color for algorithm* ONL.

Proof. After initialization and after each round there exists (1) the special piece of volume 6ε that is not single-colored and (2) for each color there exist small single-colored pieces of total volume at least $1-2\varepsilon$. *Now suppose by contradiction that no color is deficient.* Then for each color c the total volume of small c-away pieces is less than ε, i.e. the volume of the small pieces on the main server for c is at least $1-3\varepsilon$. As no server can have pieces of total volume more than $1+\varepsilon$ assigned to it and $\varepsilon \leq 1/8$, it follows that the non-special pieces on the main server of c require volume more than $(1+\varepsilon)/2$, and, thus, each server can be the main server for at

most one color. As there are as many colors as there are server, each server is the main server for exactly one color and each color has exactly one main server.

Now consider the server s^* on which the special piece is placed and let it be the main server for some color c. Then the total volume of the pieces on s^* is 6ε for the special piece. If c is not deficient, s^* has load at least $1 - 3\varepsilon$ for the non-special pieces of color c. Thus, the server's load is at least $1 + 3\varepsilon$ which is not possible. Hence, there must exist a deficient color.

Next we show that there is always an unfinished deficient color. This is trivially true after initialization as all colors are unfinished. Let us now consider the end of a round. Note that every color c that is finished has a non-special piece of volume at least $1 - 2\varepsilon$ and, thus, *the special piece cannot be placed on the main server of a finished color c*. Recall that every non-deficient color has pieces of total volume at least $1 - \varepsilon$ on its main server. Thus, *the special piece cannot be placed the main server of any non-deficient color*. Thus, the special piece can only be placed on a server that is not the main server of a finished deficient or a non-finished color. If every deficient color is finished, every color has a main server and the special piece cannot be placed on any of them. As, however, there are as many servers as there are colors, it would follow that the special piece is not placed on any server, which is not possible. Thus, there must exist a deficient unfinished color. □

Lemma 7.30. *For each color c it holds at the beginning and end of each round that almost all small pieces of color c have volume $2^i/k$ for some integer i and the other small piece has even smaller volume.*

Proof. By induction on the number of rounds. The claim holds after initialization for $i = 1$ for every color. During each round for some color c the pieces of color c and volume $2^i/k$ are merged pairwise, and the possible left-over piece of volume $2^i/k$ is merged with the leftover piece of earlier rounds, if it exists. From the induction claim it follows that the leftover piece of earlier rounds has volume less than $2^i/k$. Thus, the resulting leftover piece has volume less than $2^{i+1}/k$. Furthermore, the pieces of the other colors remain unchanged. Thus, the claim follows. □

Lemma 7.31. *At the beginning of the final merging steps there is exactly one unfinished deficient color left and there are at least 2^{j+3} small pieces have volume $\varepsilon/2^j$ for some integer $j \geq 1$.*

Proof. Lemma 7.29 holds after each round, thus, also after the final round. It shows that there is still at least one deficient unfinished color. As there are no more rounds, there at most one deficient unfinished color, which implies that there is exactly one deficient unfinished color. As it is unfinished, all its small pieces have volume less than ε. For the rest of the proof we only consider small pieces of this color.

Recall that $\varepsilon k = 2^m$ and $k \geq 2^{m+5}$. Initially there are $k - 2\varepsilon k \geq 2^{m+5} - 2^{m+1}$ small pieces of volume $1/k$ each. Let k' be the largest power of 2 that is at most $k - 2\varepsilon k$. It follows that $k - 2\varepsilon k \geq k' > k/2 - \varepsilon k \geq 2^{m+4} - 2^m$. Thus, initially there are at least k' small pieces of volume $1/k = \varepsilon/2^m$ each. Let j be any integer

with $0 \leq j \leq m$ such that exactly $m - j$ rounds were executed for this color. Thus, there are at least $k'/2^{m-j}$ pieces of volume $2^{m-j}\varepsilon/2^m = \varepsilon/2^j$ at the beginning of the final merging steps. Note that $k'/2^{m-j} \geq (2^{m+4} - 2^m)/2^{m-j} = 2^{j+4} - 2^j \geq 2^{j+3}$. Thus, there are at least 2^{j+3} pieces of volume $\varepsilon/2^j$. As the color is unfinished, each small piece has volume less than ε, i.e. $j \geq 1$. Thus the lemma holds. □

Next we analyze how many rounds are performed for a given color until it is finished. Consider any color c. The number of rounds necessary to increase the volume of almost all small pieces of color c from $1/k$ to ε is $\log(\varepsilon k)$ as εk is a power of 2. Each round roughly halves the number of small pieces. Thus, we only have to show that there are enough small pieces available initially so that $\log(\varepsilon k)$ many rounds are possible for color c.

Lemma 7.32. *For each finished color* $\log(\varepsilon k)$ *many rounds are executed.*

Proof. Fix a color c and consider in this proof only pieces of color c. As $\varepsilon \leq 1/8$ and each initial small piece is a single vertex, there are $k - 2\varepsilon k \geq 3k/4$ many such small pieces initially. Let k' be the largest power of 2 that is at most $3k/4$. Note that $k' > 3k/8$. Thus the number of small pieces of volume at least ε is at least $k'/2^{\log(\varepsilon k)} > 3/(8\varepsilon) \geq 3$. Hence for each finished color $\log(\varepsilon k)$ rounds will be executed. □

As there are ℓ different colors, it suffices to show that in almost every round algorithm ONL moves pieces with total volume $\Omega(\varepsilon)$ to achieve the desired lower bound of $\Omega(\varepsilon\ell \log(\varepsilon k))$ for the cost of ONL.

Lemma 7.33. *In one round of the above process, except in the last round for each color,* ONL *moves vertices with volume* $\Omega(\varepsilon)$*. In total, the algorithm moves vertices with volume* $\Omega(\varepsilon\ell \log(\varepsilon k))$

Proof. Fix a color c and only consider pieces of color c in this proof. Note that when two pieces of different servers, of volume $2^i/k$ each, are merged, at least one of them has to change its server, resulting in a cost of $2^i/k$ for the algorithm. We proved in Lemma 7.29 that at the beginning of each round a deficient color exists. A deficient color has away pieces of total volume at least ε, i.e., there are small pieces of total volume at least ε not on the main server. During a round, as shown by Lemma 7.30, almost all of these pieces have volume $2^i/k$ for some integer i and their total contribution to the total volume of all away pieces of color c is larger than $\varepsilon - 2^i/k$ (subtracting out the volume of the potentially existing leftover piece of even smaller volume). Thus, as long as $\varepsilon - 2^i/k \geq \varepsilon/2$, i.e., in all but the last round, the total volume of all the away pieces excluding the leftover piece is larger then $\varepsilon/2$. In the following when we talk about a small piece, we mean a small piece that is *not* the leftover piece and we fix a round that is not the last round. We will show that at least $\varepsilon/2$ volume is merged by pieces on different servers in this round, resulting in at least $\varepsilon/4$ cost for the algorithm.

Now consider two cases: (1) If the main server s^* contains small pieces of total volume at least $\varepsilon/2$, then every away piece can be merged with a small piece either on s^* or on a different server. Thus at least $\varepsilon/2$ volume is merged by pieces on different servers. (2) If, however, the main server contains small pieces of total volume less than $\varepsilon/2$, then *every* server contains small pieces of total volume less than $\varepsilon/2$. Small pieces of different servers are merged until all remaining small pieces are on the same server. However, this server has less than $\varepsilon/2$ volume of small pieces, i.e., more than $\varepsilon/2$ volume must have been merged between different servers. Thus in both cases the algorithm has cost at least $\varepsilon/4$. The second claim follows immediately from the discussion preceding the lemma. □

Next, we prove an upper bound on the volume of vertices moved by OPT.

Lemma 7.34. *In total,* OPT *moves vertices with volume* $O(\varepsilon)$.

Proof. Right at the beginning OPT places the special piece of volume 6ε on server s^* and moves the small pieces of color c^* that are merged in the final merging step with a different color to the main server for the corresponding color. Thus, none of the other steps cause any cost for OPT. Thus, OPT only has cost $O(\varepsilon)$. □

The previous two lemmas imply a lower bound on the competitive ratio of $\Omega(\ell \log(\varepsilon k))$ for deterministic algorithms. This finishes the proof of the theorem.

7.7.2 Lower Bounds for Randomized Algorithms

Theorem 7.35. *Any randomized online algorithm must have a competitive ratio of* $\Omega(\log \ell + \log k)$.

Proposition 7.36. *If* $\varepsilon < 1/6$*, then any randomized online algorithm must have a competitive ratio of* $\Omega(\log \ell)$.

Proof. We use Yao's principle [210] to derive our lower bound and provide a randomized hard instance against a deterministic algorithm. The hard instance starts by merging the vertices of each server into monochromatic pieces of volume 2ε each. Now the hard instance arbitrarily picks three pieces with different majority colors and merges them into a piece of volume 6ε and we call this piece *special*. Next, the hard instance proceeds in $\ell - 1$ rounds. Before the first round all servers are *unfinished*. In round i, the hard instance picks an unfinished server s uniformly at random. Now the hard instance uniformly at random picks monochromatic pieces with color s of total volume $1 - 2\varepsilon$ and merges them in arbitrary order; after that we call s *finished*. When all $\ell - 1$ rounds are over, a final configuration in which all pieces have volume 1 is obtained as follows. First, observe that there is a unique unfinished server s^*. Now the hard instance merges the special piece and monochromatic pieces of color s^* of total volume $1 - 6\varepsilon$. The remaining monochromatic pieces of color s^* are merged with the components of the finished servers from

which the vertices of the special piece originated. All other monochromatic components of volume 2ε are merged with the large monochromatic components with the same color as the piece itself.

For a given schedule, we say that an unfinished server s is *split* if monochromatic pieces with color s and of volume at least ε are not scheduled on s. Now observe that after each round there exists a server which is split: First, observe that none of the finished servers can store its monochromatic piece of volume $1 - 2\varepsilon$ together with the special piece of volume 6ε. Now if none of the (unfinished) servers was split, one of them would contain all of its monochromatic pieces of total volume at least $1-2\varepsilon$ together with the special piece of volume 6ε. Thus, the total load of the server is $1 + 4\varepsilon$ which is not a valid schedule.

Next, we show that if a server s is split, then the algorithm has moved monochromatic pieces with color s of volume at least ε: First, suppose the algorithm has scheduled all monochromatic pieces of color s on some server $s' \neq s$. Then the algorithm has paid at least $1-2\varepsilon \geq \varepsilon$ to move the monochromatic pieces of color s to s'. Second, suppose the monochromatic pieces of color s are scheduled on at least two different servers. Then the algorithm must have moved at least one monochromatic piece of color s away from s. Since s is unfinished and all monochromatic pieces of s have volume 2ε, the algorithm has paid at least ε for moving monochromatic pieces of color s.

Now we analyze the cost paid by the algorithm. Observe that before round i there are $\ell-i+1$ unfinished servers and at least one of them is split. Let s be a split server. Thus with probability $1/(\ell-i+1)$ the hard instance picks the split server s. It follows from the previous claims that the algorithm paid at least ε to move pieces of color s. Since the above arguments hold for each round, the total expected cost of the algorithm is

$$\sum_{i=1}^{\ell-1} \varepsilon \frac{1}{\ell-i+1} = \sum_{i=2}^{\ell} \varepsilon \frac{1}{i} = \Omega(\varepsilon \log \ell).$$

Next, observe that OPT never moves more than $O(\varepsilon)$ volume: Indeed, the hard instance only merges pieces in which all vertices have the same color except when (1) creating the special piece of volume $O(\varepsilon)$, (2) merging the special piece with the vertices from s^* and (3) merging the small pieces from s^* with the large pieces of the servers from which the special piece originated. All of these steps can be performed by only moving volume $O(\varepsilon)$.

Thus, the competitive ratio of the algorithm is $\Omega(\log \ell)$. □

Proposition 7.37. *Any randomized algorithm must have a competitive ratio of at least* $\Omega(\log k)$.

Proof. We use Yao's principle [210] to derive our lower bound and provide a random instance against a deterministic algorithm. In the instance all pieces initially have volume $1/k$, i.e., the pieces consist of single vertices. The lower bounds proceeds

in $\log k$ rounds. In each round, we pick a perfect matching between all pieces uniformly at random. Thus, after i rounds, all pieces have volume $2^i/k$. Note that after $\log k$ rounds all pieces have volume 1 and we have obtained a valid final configuration.

We claim that in each round the algorithm has to move volume $\Omega(\ell)$. Suppose we are currently in round i. Now consider two pieces p_1 and p_2 which are merged during a single round. Then the probability that p_1 and p_2 are assigned to different servers is $\Omega((\ell-1)/\ell) = \Omega(1)$. Furthermore, observe that each piece has volume $2^i/k$ and in total there are $n/2^i$ pieces. Now by linearity of expectation we obtain that the expected volume moved by the algorithm in round i is $\Omega(2^i/k \cdot n/2^i) = \Omega(\ell)$.

Next, observe that the total cost paid by the algorithm is $\Omega(\ell \cdot \log k)$ since there are $\log k$ rounds. Furthermore, OPT never moves volume more than $O(\ell)$ because it moves each vertex at most once. Thus, the competitive ratio is $\Omega(\log k)$. □

7.8 Omitted Proofs

7.8.1 Claim 7.38

Claim 7.38. *Let $\varepsilon \in (0, \frac{1}{4})$ and $k \geq 10/\varepsilon^4$. Then there exists δ such that (1) $\frac{1}{2}\epsilon^2 \leq \delta \leq \epsilon^2$, (2) $\delta = i\frac{1}{k}$ with $i \in \mathbb{N}$ and (3) $\lceil 1 \rceil_\delta - 1 \leq \delta/2$, where $\lceil \cdot \rceil_\delta$ is the operation of rounding up to a multiple of δ.*

Proof. Set $\delta^* = \max\{i\frac{1}{k} : i\frac{1}{k} \leq \varepsilon^2\}$ and observe that $\varepsilon^2 - \frac{1}{k} < \delta^* \leq \varepsilon^2$. Thus, δ^* satisfies (1) and (2). Next, if (3) is satisfied, we have found a value $\delta = \delta^*$ with the desired properties. Otherwise, we set $\delta = \delta^*$ and then we keep on decreasing δ by $\frac{1}{k}$ until (3) holds. Clearly, when this procedure stops, (2) and (3) are satisfied. It remains to show that (1) holds.

Observe that for every δ, we have that $\lceil 1 \rceil_\delta = \lceil \frac{1}{\delta} \rceil \delta$, where $\lceil \cdot \rceil$ denotes rounding up to the next integer. Note that when the above procedure decreases δ by $\frac{1}{k}$, we still have $\lceil \frac{1}{\delta} \rceil = \lceil \frac{1}{\delta^*} \rceil$: After the first decrease, the value of $\lceil 1 \rceil_{\delta^*}$ drops by $\lceil \frac{1}{\delta^*} \rceil \cdot \frac{1}{k} \leq \frac{2}{\varepsilon^2} \cdot \frac{\varepsilon^4}{10} \leq \frac{\varepsilon^2}{5}$. Thus, for $\delta = \delta^* - \frac{1}{k}$ we still have $\lceil 1 \rceil_\delta - 1 \geq 0$. Using that $\delta \leq \delta^*$ implies $\lceil \frac{1}{\delta} \rceil \geq \lceil \frac{1}{\delta^*} \rceil$, gives the claim for $\delta = \delta^* - \frac{1}{k}$. Now the argument can be extended using induction.

Now we show that (1) holds when the procedure finishes. Observe that every time the procedure decreases δ by $\frac{1}{k}$, the RHS of (3) decreases by $\frac{1}{2k} \leq \frac{\varepsilon^4}{20}$ and the LHS of (3) decreases by $\lceil \frac{1}{\delta} \rceil \cdot \frac{1}{k} = \lceil \frac{1}{\delta^*} \rceil \cdot \frac{1}{k} \geq \frac{1}{2\varepsilon^2} \cdot \frac{1}{k} \gg \frac{1}{k}$. Since we have that $\lceil 1 \rceil_{\delta^*} - 1 - \delta^*/2 \leq \varepsilon^2/2$ and we just saw that the procedure decreases the LHS much faster than δ, the above procedure finishes with $\delta \geq \frac{1}{2}\varepsilon^2$. □

7.8.2 Optimality of Monochromatic Merges and Commits on Ordinary Servers

Let c be a cost vector for the ILP from Section 7.4. Note that c has an entry $c_{(r,m)}$ for each γ-feasible configuration (r,m). For the deterministic algorithm we simply have

$$c_{(r,m)} := \begin{cases} 0, & r \geq m, \\ 1, & r \ngeq m, \end{cases} \tag{7.4}$$

and for the randomized algorithm c is defined as in Equation 7.2.

Recall the definition of the partial order $\leq_p$ from Section 7.6.1. We say that a cost vector c is $\leq_p$*-respecting* if it satisfies (1) $c_{(r,m)} = 0$ for all ordinary configurations (r,m) and (2) $m_1 \geq_p m_2 \implies c_{(r_1,m_1)} \leq c_{(r_2,m_2)}$ for all extraordinary configurations (r_1,m_1) and (r_2,m_2). It is easy to see that the cost vectors for the deterministic and the randomized algorithm both satisfy this condition.

Lemma 7.39. *Suppose the ILP is equipped with a $\leq_p$-respecting cost vector. Let s be a server and let p_s and p_ℓ be two large pieces monochromatic for s. Further, let x be an optimal ILP solution before the pieces p_s and p_ℓ get merged and let x' be an optimal ILP solution directly after the merge. Then the objective function value of the ILP for x' is not smaller than the objective function value of the ILP for x.*

Proof. We prove the lemma by contradiction. Suppose the new solution x' has a smaller objective function value than x. We show that this implies that the solution x before merging pieces p_s and p_ℓ was not optimal.

Let i_s, i_ℓ and i_m denote the size classes of p_s, p_ℓ and p_m. Now consider a schedule S' respecting the ILP solution x'. Let s' be the server on which p_m is scheduled (note that in S' it might be the case that $s' \neq s$) and let (r'_s, m'_s), $(r'_{s'}, m'_{s'})$ denote the configurations of s and s' in S', respectively. Now we set S^* to the same schedule as S' except that piece p_m is replaced by p_s and p_ℓ. Note that in S^* all server configurations are the same as in S' except that s' has reservation $r^*_{s'}$ and s has source vector m^*_s with

$$\begin{array}{lcll} r^*_{s'i_s} & := & r'_{s'i_s} & + \ |p_s|_c \\ r^*_{s'i_\ell} & := & r'_{s'i_\ell} & + \ |p_\ell|_c \\ r^*_{s'i_m} & := & r'_{s'i_m} & - \ |p_m|_c \end{array}$$

and

$$\begin{array}{lcll} m^*_{si_s} & := & m'_{si_s} & + \ |p_s|_c \\ m^*_{si_\ell} & := & m'_{si_\ell} & + \ |p_\ell|_c \\ m^*_{si_m} & := & m'_{si_m} & - \ |p_m|_c \end{array} .$$

It follows that S^* and S' have exactly the same extraordinary servers.

Now based on S^* we derive an ILP solution x^* by setting $x^*_{(r,m)}$ to the number of servers in S^* with configuration (r,m) for all (r,m). It is easy to see that x^* is a feasible solution for the ILP before p_s and p_ℓ were merged. Now we distinguish two

cases. First, suppose s is not extraordinary in S^*. Then x^* has the same objective function value for the ILP as x'. This contradicts the optimality of x before p_s and p_ℓ were merged. Second, suppose that s is extraordinary in S^*. In this case note that we have $m'_s \leq_p m^*_s$ and hence $c_{(r'_s,m'_s)} \geq c_{(r^*_s,m^*_s)}$ since the cost vector c is $\leq_p$-presering. Thus, the objective function value of x^* is upper bounded by the objective function of x'. This again contradicts the optimality of x. □

Lemma 7.40. *Suppose the ILP is equipped with a $\leq_p$-respecting cost vector. Let s be a server and let p_m be a large monochromatic pieces for s. Let x be an optimal ILP solution before volume is committed for p_m and let x' be an optimal ILP solution directly after the commit. Then the objective function value of the ILP for x' is not smaller than the objective function value of the ILP for x.*

Proof. We proceed similar to the proof of the Lemma 7.40. We prove the lemma by contradiction. Suppose the new solution x' has a smaller objective function value than x. We show that this implies that the solution x before committing the volume for p_m was not optimal.

Let i_m and i'_m denote the size classe of p_m before and after the commit, respectively. Now consider a schedule S' respecting the ILP solution x'. Let s' be the server on which p_m is scheduled (note that in S' it might be the case that $s' \neq s$) and let (r'_s, m'_s), $(r'_{s'}, m'_{s'})$ denote the configurations of s and s' in S', respectively. Now we set S^* to the same schedule as S' except that we undo the commit for piece p_m. Note that in S^* all server configurations are the same as in S' except that s' has reservation $r^*_{s'}$ and s has source configuration m^*_s with

$$\begin{array}{lcll} r^*_{s'0} & := & r'_{s'0} & + \ \delta \\ r^*_{s'i_m} & := & r'_{s'i_m} & + \ |p_m|_c - \delta \\ r^*_{s'i'_m} & := & r'_{s'i'_m} & - \ |p_m|_c \end{array}$$

and

$$\begin{array}{lcll} m^*_{s0} & := & m'_{s0} & + \ \delta \\ m^*_{si_m} & := & m'_{si_m} & + \ |p_m|_c - \delta \\ m^*_{si'_m} & := & m'_{si'_m} & - \ |p_m|_c \end{array} .$$

It follows that S^* and S' have exactly the same extraordinary servers.

Now based on S^* we derive an ILP solution x^* by setting $x^*_{(r,m)}$ to the number of servers in S^* with configuration (r, m) for all (r, m). It is easy to see that x^* is a feasible solution for the ILP before the commit was performed. Now we distinguish two cases. First, suppose s is not extraordinary in S^*. Then x^* has the same objective function value for the ILP as x'. This contradicts the optimality of x before the commit. Second, suppose that s is extraordinary in S^*. In this case note that we have $m'_s \leq_p m^*_s$ and hence $c_{(r'_s,m'_s)} \geq c_{(r^*_s,m^*_s)}$ since the cost vector c is $\leq_p$-presering. Thus, the objective function value of x^* is upper bounded by the objective function of x'. This again contradicts the optimality of x. □

7.9 Conclusion

We studied an online graph partitioning problem which has applications in resource allocation problems in the cloud. We improved upon the results of Chapter 6 by obtaining new randomized and deterministic algorithms together with matching lower bounds. Thus, our derived bounds and algorithms are asymptotically optimal.

Two open research questions are as follows. First, the dependency of our algorithms on ε is superpolynomial and thus the algorithm is of rather theoretical nature. It is an interesting question whether we can obtain a polynomial dependency on ε (similar to the algorithm from Chapter 6) or whether this cannot be done when obtaining asymptotically tight competitive ratios. Second, it would be interesting to shed light on the competitive ratios achievable by randomized online algorithms in scenarios where request patterns can change arbitrarily over time (i.e., when dropping the cc-condition). We currently do not know whether this problem variant allows for polylogarithmic competitive ratios, or whether the competitive ratio is inherently linear, as in the deterministic case [20, 23].

Bibliography

[1] Emmanuel Abbe. "Community Detection and Stochastic Block Models: Recent Developments". In: *J. Mach. Learn. Res.* 18.177 (2018), pp. 1–86.

[2] Emmanuel Abbe and Colin Sandon. "Community Detection in General Stochastic Block models: Fundamental Limits and Efficient Algorithms for Recovery". In: *FOCS*. 2015, pp. 670–688.

[3] Emmanuel Abbe and Colin Sandon. "Recovering Communities in the General Stochastic Block Model Without Knowing the Parameters". In: *NIPS*. 2015, pp. 676–684.

[4] Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. "Dynamic set cover: improved algorithms and lower bounds". In: *STOC*. 2019, pp. 114–125.

[5] Amir Abboud, Aviad Rubinstein, and R. Ryan Williams. "Distributed PCP Theorems for Hardness of Approximation in P". In: *FOCS*. 2017, pp. 25–36.

[6] Amir Abboud, Richard R. Williams, and Huacheng Yu. "More Applications of the Polynomial Method to Algorithm Design". In: *SODA*. 2015, pp. 218–230.

[7] Amir Abboud and Virginia Vassilevska Williams. "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems". In: *FOCS*. 2014, pp. 434–443.

[8] Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. "An $O(\log k)$-competitive algorithm for generalized caching". In: *SODA*. 2012, pp. 1681–1689.

[9] Pankaj K. Agarwal, Lars Arge, and Ke Yi. "I/O-efficient batched union-find and its applications to terrain analysis". In: *ACM Trans. Algorithms* 7.1 (2010), 11:1–11:21.

[10] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. "Mergeable summaries". In: *ACM Trans. Database Syst.* 38.4 (2013), 26:1–26:28.

[11] Charu C. Aggarwal. *Data Mining - The Textbook*. Springer, 2015.

[12] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. "Fast Discovery of Association Rules". In: *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996, pp. 307–328.

[13] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". In: *VLDB*. 1994, pp. 487–499.

[14] Rakesh Agrawal and Ramakrishnan Srikant. "Mining Sequential Patterns". In: *ICDE*. 1995, pp. 3–14.

[15] Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. "Streaming Min-max Hypergraph Partitioning". In: *NeurIPS*. 2015, pp. 1900–1908.

[16] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data center TCP (DCTCP)". In: *SIGCOMM*. Vol. 40. 4. 2010, pp. 63–74.

[17] Richard J. Anderson and Heather Woll. "Wait-free Parallel Algorithms for the Union-Find Problem". In: *STOC*. 1991, pp. 370–380.

[18] Konstantin Andreev and Harald Räcke. "Balanced graph partitioning". In: *Theory of Comput. Syst.* 39.6 (2006), pp. 929–939.

[19] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. "Local Search Heuristics for k-Median and Facility Location Problems". In: *SIAM J. Comput.* 33.3 (2004), pp. 544–562.

[20] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. "Dynamic Balanced Graph Partitioning". In: *SIAM J. Discrete Math.* 2019.

[21] Chen Avin, Louis Cohen, Mahmoud Parham, and Stefan Schmid. "Competitive Clustering of Stochastic Communication Patterns on a Ring". In: 101.9 (2019), pp. 1369–1390.

[22] Chen Avin, Louis Cohen, and Stefan Schmid. "Competitive Clustering of Stochastic Communication Patterns on the Ring". In: *NETYS*. 2017, pp. 231–247.

[23] Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. "Online Balanced Repartitioning". In: *DISC*. 2016, pp. 243–256.

[24] Chen Avin and Stefan Schmid. "Toward Demand-Aware Networking: A Theory for Self-Adjusting Networks". In: *Comput. Commun. Rev.* 48.5 (2018), pp. 31–40.

[25] Arturs Backurs and Piotr Indyk. "Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False)". In: *SIAM J. Comput.* 47.3 (2018), pp. 1087–1097.

[26] Arturs Backurs, Liam Roditty, Gilad Segal, Virginia Vassilevska Williams, and Nicole Wein. "Towards tight approximation bounds for graph diameter and eccentricities". In: *STOC*. 2018, pp. 267–280.

[27] Thomas Baignères, Pascal Junod, and Serge Vaudenay. "How Far Can We Go Beyond Linear Cryptanalysis?" In: *ASIACRYPT*. 2004, pp. 432–450.

[28] Frank Ban, Vijay Bhattiprolu, Karl Bringmann, Pavel Kolev, Euiwoong Lee, and David P. Woodruff. "A PTAS for ℓ_p-Low Rank Approximation". In: *SODA*. 2019, pp. 747–766.

[29] Yair Bartal, Moses Charikar, and Piotr Indyk. "On Page Migration and Other Relaxed Task Systems". In: *Theor. Comput. Sci.* 268.1 (2001), pp. 43–66. Conference version in *SODA'97*.

[30] Theophilus Benson, Aditya Akella, and David A. Maltz. "Network Traffic Characteristics of Data Centers in the Wild". In: *IMC*. 2010, pp. 267–280.

[31] Anup Bhattacharya, Dishant Goyal, Ragesh Jaiswal, and Amit Kumar. "Streaming PTAS for Binary ℓ_0–Low Rank Approximation". In: *CoRR* abs/1909.11744 (2019).

[32] Sayan Bhattacharya, Monika Henzinger, and Stefan Neumann. "New Amortized Cell-Probe Lower Bounds for Dynamic Problems". In: *Theor. Comput. Sci.* 779 (2019), pp. 72–87.

[33] Marcin Bienkowski, Anja Feldmann, Johannes Grassler, Gregor Schaffrath, and Stefan Schmid. "The Wide-Area Virtual Service Migration Problem: A Competitive Analysis Approach". In: *IEEE/ACM Trans. Netw.* 22.1 (2014), pp. 165–178.

[34] Yonatan Bilu and Nathan Linial. "Are Stable Instances Easy?" In: *ICS*. 2010, pp. 332–341.

[35] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. "Listing Triangles". In: *ICALP*. 2014, pp. 223–234.

[36] David L. Black and Daniel D. Sleator. *Competitive algorithms for replication and migration problems*. Carnegie Mellon University, Tech. Rep. CMU-CS-89-201, 1989.

[37] Avrim Blum, Carl Burch, and Adam Kalai. "Finely-Competitive Paging". In: *FOCS*. 1999, pp. 450–458.

[38] Francesco Bonchi, Fosca Giannotti, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Roberto Trasarti. "A constraint-based querying system for exploratory pattern discovery". In: *Inf. Syst.* 34.1 (2009), pp. 3–27.

[39] Francesco Bonchi, Fosca Giannotti, Alessio Mazzanti, and Dino Pedreschi. "ExAnte: Anticipated Data Reduction in Constrained Pattern Mining". In: *PKDD*. 2003, pp. 59–70.

[40] Francesco Bonchi and Claudio Lucchese. "On Closed Constrained Frequent Pattern Mining". In: *ICDM*. 2004, pp. 35–42.

[41] Alan Borodin, Nati Linial, and Michael E. Saks. "An optimal on-line algorithm for metrical task system". In: *J. ACM* 39.4 (1992), pp. 745–763. Conference version in *STOC'87*.

[42] Endre Boros, Vladimir Gurvich, Leonid Khachiyan, and Kazuhisa Makino. "On Maximal Frequent and Minimal Infrequent Sets in Binary Matrices". In: *Ann. Math. Artif. Intell.* 39.3 (2003), pp. 211–221.

[43] Pat Bosshart et al. "P4: Programming protocol-independent packet processors". In: *Comput. Commun. Rev.* 44.3 (2014), pp. 87–95.

[44] Vladimir Braverman, Adam Meyerson, Rafail Ostrovsky, Alan Roytman, Michael Shindler, and Brian Tagiku. "Streaming k-means on Well-Clusterable Data". In: *SODA*. 2011, pp. 26–40.

[45] Karl Bringmann and Marvin Künnemann. "Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping". In: *FOCS*. 2015, pp. 79–97.

[46] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. "Syntactic Clustering of the Web". In: *Comput. Networks* 29.8-13 (1997), pp. 1157–1166.

[47] Douglas Burdick, Manuel Calimlim, Jason Flannick, Johannes Gehrke, and Tomi Yiu. "MAFIA: A Maximal Frequent Itemset Algorithm". In: *IEEE Trans. Knowl. Data Eng.* 17.11 (2005), pp. 1490–1504.

[48] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. "Counting triangles in data streams". In: *PODS*. 2006, pp. 253–262.

[49] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. "A Duality between Clause Width and Clause Density for SAT". In: *CCC*. 2006, pp. 252–260.

[50] Toon Calders, Nele Dexters, Joris J. M. Gillis, and Bart Goethals. "Mining frequent itemsets in a stream". In: *Inf. Syst.* 39 (2014), pp. 233–255.

[51] Timothy M. Chan and Ryan Williams. "Deterministic APSP, Orthogonal Vectors, and More: Quickly Derandomizing Razborov-Smolensky". In: *SODA*. 2016, pp. 1246–1255.

[52] L. Sunil Chandran, Davis Issac, and Andreas Karrenbauer. "On the Parameterized Complexity of Biclique Cover and Partition". In: *IPEC*. 2016, 11:1–11:13.

[53] Moses Charikar, Piotr Indyk, and Rina Panigrahy. "New Algorithms for Subset Query, Partial Match, Orthogonal Range Searching, and Related Problems". In: *ICALP*. 2002, pp. 451–462.

[54] Lijie Chen. "On The Hardness of Approximate and Exact (Bichromatic) Maximum Inner Product". In: *CCC*. 2018, 14:1–14:45.

[55] Cisco. "Cisco Global Cloud Index: Forecast and Methodology, 2015-2020". In: *White Paper* (2015).

[56] Vincent Cohen-Addad and Chris Schwiegelshohn. "On the Local Structure of Stable Clustering Instances". In: *FOCS*. 2017, pp. 49–60.

[57] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *J. Algorithms* 55.1 (2005), pp. 58–75.

[58] George Cybenko, T. G. Allen, and J. E. Polito. "Practical parallel Union-Find algorithms for transitive closure and clustering". In: *Int. J. Parallel Program.* 17.5 (1988), pp. 403–423.

[59] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Morten Stöckel. "Finding even cycles faster via capped k-walks". In: *STOC*. 2017, pp. 112–120.

[60] Anirban Dasgupta, John E. Hopcroft, Ravi Kannan, and Pradipta P. Mitra. "Spectral clustering with limited independence". In: *SODA*. 2007, pp. 1036–1045.

[61] Timothy A. Davis and Yifan Hu. "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25.

[62] Holger Dell, Wolfgang Dvořák, and Stefan Neumann. *Conditional Hardness for Approximate Counting Problems*. Manuscript. 2020. Authors ordered alphabetically.

[63] Inderjit S. Dhillon. "Co-clustering documents and words using bipartite spectral graph partitioning". In: *KDD*. 2001, pp. 269–274.

[64] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

[65] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *NSDI*. 2016, pp. 523–535.

[66] Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. "Online File Caching with Rejection Penalties". In: *Algorithmica* 71.2 (2015), pp. 279–306.

[67] Kemal Eren, Mehmet Deveci, Onur Küçüktunç, and Ümit V. Çatalyürek. "A comparative analysis of biclustering algorithms for gene expression data". In: *Briefings in Bioinformatics* 14.3 (2013), pp. 279–292.

[68] Uriel Feige and Robert Krauthgamer. "A polylogarithmic approximation of the minimum bisection". In: *SIAM J. Comput.* 31.4 (2002), pp. 1090–1118.

[69] Björn Feldkord, Matthias Feldotto, Anupam Gupta, Guru Guruganesh, Amit Kumar, Sören Riechers, and David Wajc. "Fully-Dynamic Bin Packing with Little Repacking". In: *ICALP*. 2018, 51:1–51:24.

[70] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. "Competitive paging algorithms". In: *J. Algorithms* 12.4 (1991), pp. 685–699.

[71] Amos Fiat, Yuval Rabani, and Yiftach Ravid. "Competitive k-server algorithms". In: *J. Comput. Syst. Sci.* 48.3 (1994), pp. 410–428.

[72] Daniel Firestone. *SmartNIC: FPGA Innovation in OCS Servers for Microsoft Azure*. Online. https://ocpussummit2016.sched.com/event/68u4/. 2016. Accessed: 2020-05-12.

[73] Laura Florescu and Will Perkins. "Spectral thresholds in the bipartite stochastic block model". In: *COLT*. 2016, pp. 943–959.

[74] Fedor V. Fomin, Petr A. Golovach, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. "Approximation Schemes for Low-Rank Binary Matrix Approximation Problems". In: *ACM Trans. Algorithms* 16.1 (2020), 12:1–12:39.

[75] M. Fortelius (coordinator). *New and Old Worlds Database of Fossil Mammals (NOW)*. Online. http://www.helsinki.fi/science/now/. 2003. Accessed: 2015-09-23.

[76] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. "Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters". In: *INFOCOM*. 2016, pp. 1–9.

[77] François Le Gall. "Powers of tensors and fast matrix multiplication". In: *ISSAC*. 2014, pp. 296–303.

[78] Bernard A. Galler and Michael J. Fischer. "An improved equivalence algorithm". In: *Commun. ACM* 7.5 (1964), pp. 301–303.

[79] Chao Gao, Zongming Ma, Anderson Y. Zhang, and Harrison H. Zhou. "Achieving Optimal Misclassification Proportion in Stochastic Block Models". In: *J. Mach. Learn. Res.* 18 (2017), 60:1–60:45.

[80] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[81] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. "SPIRIT: Sequential Pattern Mining with Regular Expression Constraints". In: *VLDB*. 1999, pp. 223–234.

[82] Floris Geerts, Bart Goethals, and Taneli Mielikäinen. "Tiling Databases". In: *DS*. 2004, pp. 278–289.

[83] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil R. Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel C. Kilper. "Projector: Agile reconfigurable data center interconnect". In: *SIGCOMM*. 2016, pp. 216–229.

[84] Amit Goyal, Hal Daumé III, and Suresh Venkatasubramanian. "Streaming for large scale NLP: Language Modeling". In: *HLT-NAACL*. 2009, pp. 512–520.

[85] Gösta Grahne, Laks V. S. Lakshmanan, and Xiaohong Wang. "Efficient Mining of Constrained Correlated Sets". In: *ICDE*. 2000, pp. 512–521.

[86] Gianluigi Greco, Antonella Guzzo, and Luigi Pontieri. "Mining taxonomies of process models". In: *Data Knowl. Eng.* 67.1 (2008), pp. 74–102.

[87] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharm. "Discovering all most specific sentences". In: *ACM Trans. Database Syst.* 28.2 (2003), pp. 140–174.

[88] Dimitrios Gunopulos, Heikki Mannila, Roni Khardon, and Hannu Toivonen. "Data mining, hypergraph transversals, and machine learning". In: *PODS*. 1997, pp. 209–216.

[89] Tias Guns, Siegfried Nijssen, and Luc De Raedt. "k-Pattern Set Mining under Constraints". In: *IEEE Trans. Knowl. Data Eng.* 25.2 (2013), pp. 402–418.

[90] Bruce E. Hajek, Yihong Wu, and Jiaming Xu. "Computational Lower Bounds for Community Detection on Random Graphs". In: *COLT*. 2015, pp. 899–928.

[91] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. "Firefly: A reconfigurable wireless data center fabric using free-space optics". In: *SIGCOMM*. Vol. 44. 4. 2014, pp. 319–330.

[92] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. "Frequent pattern mining: current status and future directions". In: *Data Min. Knowl. Discov.* 15.1 (2007), pp. 55–86.

[93] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach". In: *Data Min. Knowl. Discov.* 8.1 (2004), pp. 53–87.

[94] Frank Harary and Helene J Kommel. "Matrix measures for transitivity and balance". In: *Journal of Mathematical Sociology* 6.2 (1979), pp. 199–210.

[95] F. Maxwell Harper and Joseph A. Konstan. "The MovieLens Datasets: History and Context". In: *TiiS* 5.4 (2016), 19:1–19:19.

[96] John A. Hartigan. "Direct Clustering of a Data Matrix". In: *J. Am. Stat. Assoc.* 67.337 (1972), pp. 123–129.

[97] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. "Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture". In: *STOC*. 2015, pp. 21–30.

[98] Monika Henzinger, Andrea Lincoln, Stefan Neumann, and Virginia Vassilevska Williams. "Conditional Hardness for Sensitivity Problems". In: *ITCS*. 2017, 26:1–26:31.

[99] Monika Henzinger and Stefan Neumann. "Incremental and Fully Dynamic Subgraph Connectivity for Emergency Planning". In: *ESA*. 2016, 48:1–48:11.

[100] Monika Henzinger, Stefan Neumann, Harald Räcke, and Stefan Schmid. *Tight Bounds for Online Graph Partitioning*. Manuscript. 2020. Authors ordered alphabetically.

[101] Monika Henzinger, Stefan Neumann, and Stefan Schmid. "Efficient Distributed Workload (Re-)Embedding". In: *POMACS* 3.1 (2019), 13:1–13:38. Authors ordered alphabetically. Conference version in *SIGMETRICS'19*.

[102] Monika Henzinger, Stefan Neumann, and Andreas Wiese. "Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles". In: *SoCG*. 2020. To appear.

[103] Monika Henzinger, Stefan Neumann, and Andreas Wiese. "Explicit and Implicit Dynamic Coloring of Graphs with Bounded Arboricity". In: *CoRR* abs/2002.10142 (2020).

[104] Sibylle Hess, Nico Piatkowski, and Katharina Morik. "The Trustworthy Pal: Controlling the False Discovery Rate in Boolean Matrix Factorization". In: *SDM*. 2018, pp. 405–413.

[105] Dorit S. Hochbaum and David B. Shmoys. "Using dual approximation algorithms for scheduling problems theoretical and practical results". In: *J. ACM* 34.1 (1987), pp. 144–162.

[106] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving high utilization with software-driven WAN". In: *SIGCOMM*. Vol. 43. 4. 2013, pp. 15–26.

[107] Qinghua Huang, Ting Wang, Dacheng Tao, and Xuelong Li. "Biclustering Learning of Trading Rules". In: *IEEE Trans. Cybernetics* 45.10 (2015), pp. 2287–2298.

[108] Harry B. Hunt III, Madhav V. Marathe, Venkatesh Radhakrishnan, and Richard Edwin Stearns. "The Complexity of Planar Counting Problems". In: *SIAM J. Comput.* 27.4 (1998), pp. 1142–1167.

[109] Russell Impagliazzo and Ramamohan Paturi. "On the Complexity of k-SAT". In: *J. Comput. Syst. Sci.* 62.2 (2001), pp. 367–375.

[110] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. "Which Problems Have Strongly Exponential Complexity?" In: *J. Comput. Syst. Sci.* 63.4 (2001), pp. 512–530.

[111] Alon Itai and Michael Rodeh. "Finding a Minimum Circuit in a Graph". In: *SIAM J. Comput.* 7.4 (1978), pp. 413–423.

[112] Sushant Jain et al. "B4: Experience with a globally-deployed software defined WAN". In: *SIGCOMM* 43.4 (2013), pp. 3–14.

[113] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. "On Generating All Maximal Independent Sets". In: *Inf. Process. Lett.* 27.3 (1988), pp. 119–123.

[114] Roberto J. Bayardo Jr. "Efficiently Mining Long Patterns from Databases". In: *SIGMOD*. 1998, pp. 85–93.

[115] Glenn Judd. "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter." In: *NSDI*. 2015, pp. 145–157.

[116] Matti Karppa and Petteri Kaski. "Engineering Boolean Matrix Multiplication for Multiple-Accelerator Shared-Memory Architectures". In: *CoRR* abs/1909.01554 (2019).

[117] Benny Kimelfeld and Phokion G. Kolaitis. "The Complexity of Mining Maximal Frequent Subgraphs". In: *ACM Trans. Database Syst.* 39.4 (2014), 32:1–32:33. Conference version in *PODS'13*.

[118] Richard E. Korf. "Multi-Way Number Partitioning". In: *IJCAI*. 2009, pp. 538–543.

[119] Petr Krajca and Martin Trnecka. "Parallelization of the GreConD Algorithm for Boolean Matrix Factorization". In: *ICFCA*. 2019, pp. 208–222.

[120] Robert Krauthgamer and Uriel Feige. "A Polylogarithmic Approximation of the Minimum Bisection". In: *SIAM Rev.* 48.1 (2006), pp. 99–130.

[121] Amit Kumar and Ravindran Kannan. "Clustering with Spectral Norm and the k-Means Algorithm". In: *FOCS*. 2010, pp. 299–308.

[122] Ravi Kumar, Rina Panigrahy, Ali Rahimi, and David P. Woodruff. "Faster Algorithms for Binary Matrix Factorization". In: *ICML*. 2019, pp. 3551–3559.

[123] Jérôme Kunegis. "KONECT: the Koblenz network collection". In: *WWW*. 2013, pp. 1343–1350.

[124] Michihiro Kuramochi and George Karypis. "Frequent Subgraph Discovery". In: *ICDM*. 2001, pp. 313–320.

[125] Branislav Kveton, S. Muthukrishnan, Hoa T. Vu, and Yikun Xian. "Finding Subcube Heavy Hitters in Analytics Data Streams". In: *WWW*. 2018, pp. 1705–1714.

[126] Matthieu Latapy. "Main-memory triangle computations for very large (sparse (power-law)) graphs". In: *Theor. Comput. Sci.* 407.1-3 (2008), pp. 458–473.

[127] Lillian Lee. "Fast context-free grammar parsing requires fast boolean matrix multiplication". In: *J. ACM* 49.1 (2002), pp. 1–15.

[128] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. "Scaling Distributed Machine Learning with the Parameter Server." In: *OSDI*. Vol. 14. 2014, pp. 583–598.

[129] Lifan Liang and Songjian Lu. "Noisy and Incomplete Boolean Matrix Factorization via Expectation Maximization". In: *CoRR* abs/1905.12766 (2019).

[130] Edo Liberty, Michael Mitzenmacher, Justin Thaler, and Jonathan Ullman. "Space Lower Bounds for Itemset Frequency Sketches". In: *PODS*. 2016, pp. 441–454.

[131] Shiau Hong Lim, Yudong Chen, and Huan Xu. "A Convex Optimization Framework for Bi-Clustering". In: *ICML*. 2015, pp. 1679–1688.

[132] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. "Tight Hardness for Shortest Cycles and Paths in Sparse Graphs". In: *SODA*. 2018, pp. 1236–1252.

[133] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. "A Unifying Framework for Mining Approximate Top-k Binary Patterns". In: *IEEE Trans. Knowl. Data Eng.* 26.12 (2014), pp. 2900–2913.

[134] Sara C. Madeira and Arlindo L. Oliveira. "Biclustering Algorithms for Biological Data Analysis: A Survey". In: *IEEE/ACM Trans. Comput. Biology Bioinform.* 1.1 (2004), pp. 24–45.

[135] Fredrik Manne and Md. Mostofa Ali Patwary. "A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers". In: *PPAM*. 2009, pp. 186–195.

[136] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. "Efficient Algorithms for Discovering Association Rules". In: *AAAI Workshops. Technical Report WS-94-03*. 1994, pp. 181–192.

[137] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. "Better Algorithms for Counting Triangles in Data Streams". In: *PODS*. 2016, pp. 401–411.

[138] Frank McSherry. "Spectral Partitioning of Random Graphs". In: *FOCS*. 2001, pp. 529–537.

[139] Manor Mendel and Steven S. Seiden. "Online companion caching". In: *Theor. Comput. Sci.* 324.2–3 (2004), pp. 183–200.

[140] Pauli Miettinen. "Dynamic Boolean Matrix Factorizations". In: *ICDM*. 2012, pp. 519–528.

[141] Pauli Miettinen. "Matrix decomposition methods for data mining: Computational complexity and algorithms". PhD thesis. Helsingin yliopisto, 2009.

[142] Pauli Miettinen, Taneli Mielikäinen, Aristides Gionis, Gautam Das, and Heikki Mannila. "The Discrete Basis Problem". In: *IEEE Trans. Knowl. Data Eng.* 20.10 (2008), pp. 1348–1362.

[143] Pauli Miettinen and Stefan Neumann. "Recent Developments in Boolean Matrix Factorization". In: *IJCAI*. 2020. Survey Article. To appear.

[144] Pauli Miettinen and Jilles Vreeken. "MDL4BMF: Minimum Description Length for Boolean Matrix Factorization". In: *ACM Trans. Knowl. Discov. Data* 8.4 (2014), 18:1–18:31.

[145] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. "Network motifs: simple building blocks of complex networks". In: *Science* 298.5594 (2002), pp. 824–827.

[146] Jayadev Misra and David Gries. "Finding Repeated Elements". In: *Sci. Comput. Program.* 2.2 (1982), pp. 143–152.

[147] Pradipta P. Mitra. *A Simple Algorithm for Clustering Mixtures of Discrete Distributions*. Online. `https://sites.google.com/site/ppmitra/invariant.pdf`.

[148] Jeffrey C. Mogul and Lucian Popa. "What we talk about when we talk about cloud network performance". In: *Comput. Commun. Rev.* 42.5 (2012), pp. 44–48.

[149] Jaroslav Nešetřil and Svatopluk Poljak. "On the complexity of the subgraph problem". In: *Commentationes Mathematicae Universitatis Carolinae* 26.2 (1985), pp. 415–419.

[150] Stefan Neumann. "Bipartite Stochastic Block Models with Tiny Clusters". In: *NeurIPS*. 2018, pp. 3871–3881.

[151] Stefan Neumann. "Finding Tiny Clusters in Bipartite Graphs". In: *INFORMATIK*. Session *Best of Data Science Made in Germany, Austria and Switzerland*. 2019.

[152] Stefan Neumann, Rainer Gemulla, and Pauli Miettinen. "What You Will Gain By Rounding: Theory and Algorithms for Rounding Rank". In: *ICDM*. 2016, 25:1–25:14.

[153] Stefan Neumann and Pauli Miettinen. "Biclustering and Boolean Matrix Factorization in Data Streams". In: *PVLDB*. 2020. To appear.

[154] Stefan Neumann and Pauli Miettinen. "Reductions for Frequency-Based Data Mining Problems". In: *ICDM*. 2017, pp. 997–1002.

[155] Stefan Neumann, Julian Ritter, and Kailash Budhathoki. "Ranking the Teams in European Football Leagues with Agony". In: *MLSA@PKDD/ECML*. 2018, pp. 55–66.

[156] Stefan Neumann and Andreas Wiese. "This House Proves That Debating is Harder Than Soccer". In: *FUN*. 2016, 25:1–25:14.

[157] Mark E. J. Newman, Duncan J. Watts, and Steven H. Strogatz. "Random graph models of social networks". In: *Proceedings of the National Academy of Sciences* 99.1 (2002), pp. 2566–2572.

[158] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. "Exploratory Mining and Pruning Optimizations of Constrained Association Rules". In: *SIGMOD*. 1998, pp. 13–24.

[159] Mohammad Noormohammadpour and Cauligi S. Raghavendra. "Datacenter Traffic Control: Understanding Techniques and Trade-offs". In: *IEEE Communications Surveys & Tutorials* (2017).

[160] Neil Olver, Kirk Pruhs, Kevin Schewior, Rene Sitters, and Leen Stougie. "The Itinerant List Update Problem". In: *WAOA* (2018), pp. 310–326.

[161] James Orlin. "Contentment in graph theory: covering graphs with cliques". In: *Indagationes Mathematicae* 80.5 (1977), pp. 406–424.

[162] Petr Osicka and Martin Trnecka. "Boolean Matrix Decomposition by Formal Concept Sampling". In: *CIKM*. 2017, pp. 2243–2246.

[163] Parveen Patel et al. "Ananta: Cloud scale load balancing". In: *SIGCOMM*. Vol. 43. 4. 2013, pp. 207–218.

[164] Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. "Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure". In: *SEA*. 2010, pp. 411–423.

[165] Md. Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. "Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure". In: *IPDPS*. 2012, pp. 827–835.

[166] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830.

[167] Jian Pei, Jiawei Han, and Wei Wang. "Mining sequential patterns with constraints in large databases". In: *CIKM*. 2002, pp. 18–25.

[168] Eric Price. "Optimal Lower Bound for Itemset Frequency Indicator Sketches". In: *CoRR* abs/1410.2640 (2014).

[169] J. Scott Provan and Michael O. Ball. "The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected". In: *SIAM J. Comput.* 12.4 (1983), pp. 777–788.

[170] Luc De Raedt, Tias Guns, and Siegfried Nijssen. "Constraint programming for itemset mining". In: *KDD*. 2008, pp. 204–212.

[171] Satish Rao and Andréa W Richa. "New Approximation Techniques for Some Ordering Problems." In: *SODA*. Vol. 98. 1998, pp. 211–219.

[172] Siamak Ravanbakhsh, Barnabás Póczos, and Russell Greiner. "Boolean Matrix Factorization and Noisy Completion via Message Passing". In: *ICML*. 2016, pp. 945–954.

[173] Zahra S. Razaee, Arash A. Amini, and Jingyi Jessica Li. "Matched Bipartite Block Model with Covariates". In: *J. Mach. Learn. Res.* 20 (2019), 34:1–34:44.

[174] Matteo Riondato and Eli Upfal. "Efficient Discovery of Association Rules and Frequent Itemsets through Sampling with Tight Performance Guarantees". In: *ACM Trans. Knowl. Discov.* 8.4 (2014), 20:1–20:32.

[175] Matteo Riondato and Eli Upfal. "Mining Frequent Itemsets through Progressive Sampling with Rademacher Averages". In: *KDD*. 2015, pp. 1005–1014.

[176] Ronald L. Rivest. "Partial-Match Retrieval Algorithms". In: *SIAM J. Comput.* 5.1 (1976), pp. 19–50.

[177] Liam Roditty and Uri Zwick. "On Dynamic Shortest Paths Problems". In: *Algorithmica* 61.2 (2011), pp. 389–401.

[178] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. "Inside the Social Network's (Datacenter) Network". In: *SIGCOMM*. 2015, pp. 123–137.

[179] Aviad Rubinstein. "Hardness of approximate nearest neighbor search". In: *STOC*. 2018, pp. 1260–1268.

[180] Tammo Rukat, Christopher C. Holmes, Michalis K. Titsias, and Christopher Yau. "Bayesian Boolean Matrix Factorisation". In: *ICML*. 2017, pp. 2969–2978.

[181] Tammo Rukat, Christopher C. Holmes, and Christopher Yau. "Probabilistic Boolean Tensor Decomposition". In: *ICML*. 2018, pp. 4410–4419.

[182] Peter Sanders, Naveen Sivadasan, and Martin Skutella. "Online Scheduling with Bounded Migration". In: *Math. Oper. Res.* 34.2 (2009), pp. 481–498.

[183] Thomas Schank and Dorothea Wagner. "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study". In: *WEA*. 2005, pp. 606–609.

[184] Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. "Optimal Multi-Way Number Partitioning". In: *J. ACM* 65.4 (2018), 24:1–24:61.

[185] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[186] Rob Sherwood et al. "Carving research slices out of your production networks with OpenFlow". In: *Comput. Commun. Rev.* 40.1 (2010), pp. 129–130.

[187] Michael Shindler, Alex Wong, and Adam W Meyerson. "Fast and accurate k-means for large datasets". In: *NeurIPS*. 2011, pp. 2375–2383.

[188] Arjun Singh et al. "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network". In: *Comput. Commun. Rev.* 45.4 (2015), pp. 183–197.

[189] Daniel D. Sleator and Robert E. Tarjan. "Amortized efficiency of list update and paging rules". In: *Commun. ACM* 28.2 (1985), pp. 202–208.

[190] Isabelle Stanton. "Streaming balanced graph partitioning algorithms for random graphs". In: *SODA*. 2014, pp. 1287–1301.

[191] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. "TRIÈST: Counting Local and Global Triangles in Fully Dynamic Streams with Fixed Memory Size". In: *ACM Trans. Knowl. Discov.* 11.4 (2017), 43:1–43:50.

[192] Robert E. Tarjan and Jan van Leeuwen. "Worst-case Analysis of Set Union Algorithms". In: *J. ACM* 31.2 (1984), pp. 245–281.

[193] Seinosuke Toda and Mitsunori Ogiwara. "Counting Classes are at Least as Hard as the Polynomial-Time Hierarchy". In: *SIAM J. Comput.* 21.2 (1992), pp. 316–328.

[194] Hannu Toivonen. "Sampling Large Databases for Association Rules". In: *VLDB*. 1996, pp. 134–145.

[195] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos Faloutsos. "DOULION: counting triangles in massive graphs with a coin". In: *KDD*. 2009, pp. 837–846.

[196] Salil P. Vadhan. "The Complexity of Counting in Sparse, Regular, and Planar Graphs". In: *SIAM J. Comput.* 31.2 (2001), pp. 398–427.

[197] Leslie G. Valiant. "The Complexity of Computing the Permanent". In: *Theor. Comput. Sci.* 8 (1979), pp. 189–201.

[198] Luis M. Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. "Adaptive Partitioning for Large-scale Dynamic Graphs". In: *SoCC*. 2013, 35:1–35:2.

[199] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. "Krimp: mining itemsets that compress". In: *Data Min. Knowl. Discov.* 23.1 (2011), pp. 169–214.

[200] Junhao Wang, Sacha Levy, Ren Wang, Aayushi Kulshrestha, and Reihaneh Rabbany. "SGP: Spotting Groups Polluting the Online Political Discourse". In: *CoRR* abs/1910.07130 (2019).

[201] Duncan J. Watts and Steven H. Strogatz. "Collective dynamics of "small-world" networks". In: *Nature* 393.6684 (1998), pp. 440–442.

[202] Ryan Williams. "A new algorithm for optimal 2-constraint satisfaction and its implications". In: *Theor. Comput. Sci.* 348.2-3 (2005), pp. 357–365.

[203] Virginia Vassilevska Williams. "Multiplying matrices faster than coppersmith-winograd". In: *STOC*. 2012, pp. 887–898.

[204] Virginia Vassilevska Williams. "On some fine-grained questions in algorithms and complexity". In: *Proc. ICM*. 2018.

[205] Virginia Vassilevska Williams and R. Ryan Williams. "Subcubic Equivalences Between Path, Matrix, and Triangle Problems". In: *J. ACM* 65.5 (2018), 27:1–27:38.

[206] Yongqiao Xiao, Jenq-Foung Yao, Zhigang Li, and Margaret H. Dunham. "Efficient Data Mining for Maximal Frequent Subtrees". In: *ICDM*. 2003, pp. 379–386.

[207] Jiaming Xu, Rui Wu, Kai Zhu, Bruce E. Hajek, R. Srikant, and Lei Ying. "Jointly clustering rows and columns of binary matrices: algorithms and trade-offs". In: *SIGMETRICS*. 2014, pp. 29–41.

[208] Xifeng Yan and Jiawei Han. "gSpan: Graph-Based Substructure Pattern Mining". In: *ICDM*. 2002, pp. 721–724.

[209] Guizhen Yang. "Computational aspects of mining maximal frequent patterns". In: *Theor. Comput. Sci.* 362.1-3 (2006), pp. 63–85. Conference version in KDD'04.

[210] Andrew C. Yao. "Probabilistic Computations: Toward a Unified Measure of Complexity". In: *FOCS*. 1977, pp. 222–227.

[211] Esti Yeger-Lotem, Shmuel Sattath, Nadav Kashtan, Shalev Itzkovitz, Ron Milo, Ron Y Pinter, Uri Alon, and Hanah Margalit. "Network motifs in integrated cellular networks of transcription–regulation and protein–protein interaction". In: *Proceedings of the National Academy of Sciences* 101.16 (2004), pp. 5934–5939.

[212] Neal E. Young. "On-line caching as cache size varies". In: *SODA*. 1991, pp. 241–250.

[213] Huacheng Yu. "An improved combinatorial algorithm for Boolean matrix multiplication". In: *Inf. Comput.* 261.Part (2018), pp. 240–247.

[214] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. "Rethinking virtual network embedding: substrate support for path splitting and migration". In: *Comput. Commun. Rev.* 38.2 (2008), pp. 17–29.

[215] Se-Young Yun, Marc Lelarge, and Alexandre Proutière. "Streaming, Memory Limited Algorithms for Community Detection". In: *NeurIPS*. 2014, pp. 3167–3175.

[216] Mohammed Javeed Zaki. "Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications". In: *IEEE Trans. Knowl. Data Eng.* 17.8 (2005), pp. 1021–1035.

[217] Mohammed Javeed Zaki. "SPADE: An Efficient Algorithm for Mining Frequent Sequences". In: *Mach. Learn.* 42.1/2 (2001), pp. 31–60.

[218] Hongyuan Zha, Xiaofeng He, Chris H. Q. Ding, Ming Gu, and Horst D. Simon. "Bipartite Graph Partitioning and Data Clustering". In: *CIKM*. 2001, pp. 25–32.

[219] Zhixin Zhou and Arash A. Amini. "Analysis of spectral clustering algorithms for community detection: the general bipartite setting". In: *J. Mach. Learn. Res.* 20 (2019), 47:1–47:47.

[220] Zhixin Zhou and Arash A. Amini. "Optimal Bipartite Network Clustering". In: *CoRR* abs/1803.06031 (2018).

[221] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. "Improving recommendation lists through topic diversification". In: *WWW*. 2005, pp. 22–32.